МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
МОСКОВСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

А.В.РОЩИН

ФУНКЦИОНАЛЬНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВСТРАИВАЕМЫХ СИСТЕМ

УЧЕБНОЕ ПОСОБИЕ

Москва – 2017

УДК 004.45
ББК 32.973.1
Р81

Рощин А.В. Функциональное программное обеспечение встраиваемых систем. [Электронный ресурс]: Учебное пособие / А.В.Рощин. – М.: Московский технологический университет (МИРЭА), 2017. – 256 с. — 1 электрон. опт. диск (CD-ROM).

В пособии изложены основные этапы 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».

……..

Учебно-методическое пособие издается в авторской редакции.

Рецензенты:

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

Дорри Манучер Хабибуллаевич, д.т.н., профессор, главный научный сотрудник лаборатории № 49 ИПУ им. В.А.Трапезникова РАН

УДК 004.45
ББК 32.973.1

Минимальные системные требования:
Наличие операционной системы Windows, поддерживаемой производителем.
Наличие свободного места в оперативной памяти не менее 128 Мб.
Наличие свободного места в памяти хранения (на жестком диске) не менее 30 Мб.
Наличие интерфейса ввода информации.
Дополнительные программные средства: программа для чтения pdf-файлов (Adobe Reader).

Подписано к использованию по решению Редакционно-издательского совета Московского технологического университета от ___ _____ 2017 г.

Тираж 10

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

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».

Пособие состоит из четырех частей. Основной материал можно условно разбить на две части.

Первая из них посвящена вопросам программирования на языке ассемблера для микропроцессора 8086/88. Дано краткое описание команд микропроцессора, а также основные приемы работы с компилятором TASM.

Во второй части приведены примеры программ, решающих конкретно сформулированные задачи с использованием возможностей, изложенных в первой части.

Третья часть посвящена особенностям программирования на языке ассемблера для 32-разрядныз микропроцессоров. Дано краткое описание дополнительных команд 32-разрядных микропроцессоров, дополнительные возможности, связанные с использованием 32-разрядных операндов, а также с работой в защищенном режиме.

В четвертой части приведены возможности использования 32-разрядной адресации в реальном режиме, а также примеры использования линейной 32-разрядной адресации в реальном режиме.

Особенности программирования встраиваемых систем во многом определили характер и содержание данного учебного пособия. Если еще относительно недавно (15-20 лет назад) под встраиваемыми системами понимали что-то очень специальное – станки с ЧПУ, ракеты и т. д., то сейчас встраиваемые системы окружают нас буквально везде – от стиральных машин, до детских кукол с полной имитацией жизнедеятельности. Такие системы могут быть реализованы и на маленьком и дешевом pic контроллере и на серьезных платформах, основанных на микропроцессорах i386EX. Номенклатура микропроцессоров и/или микроконтроллеров, на которых реализуются встроенные системы, громадна. При этом их функциональные возможности достаточно обширны и продолжают наращиваться. Так, в существующих сериях имеются достаточно дешевые микроконтроллеры, у которых память программ составляет 256 Кбайт и более. Системы, основанные

на i386EX и им подобных, часто снабжаются оперативной памятью в 4-16 Гбайт, дисковыми (часто твердотельными) накопителями и т.п. Практически для всех семейств имеются фирменные средства разработки (с бесплатными базовыми комплектами), позволяющие быстро подготавливать программную «начинку» для таких систем. Многие средства разработки позволяют писать программы на Си и подобных языках. Однако, любая более или менее серьезная система до сих пор программируется на ассемблере, так как именно он дает наиболее быстрый и эффективный код. Что касается операционных систем встроенных микропроцессоров/контроллеров. Хотя сейчас и присутствуют на рынке многозадачные системы реального и «почти реального» времени (LTLinux, QNX, Windows XP Embeded), все же лучше, чем MS-DOS и ее модификации (DR-DOS и другие), до сих пор не придумали. Эта однозадачная операционная система, во-первых, делает в точности то, о чем ее попросишь, и, во-вторых, не делает ничего по собственной инициативе.

Именно эти особенности встроенных систем и определили содержание учебного пособия. Изучаемый ассемблер – это ассемблер x86, а среда – реальный режим и MS-DOS. Выбор ассемблера обоснован его абсолютной доступностью и достаточной полнотой. Ассемблер любого современного микроконтроллера обычно является фактически подмножеством именно такого ассемблера.

1 ПРОГРАММИРОВАНИЕ МИКРОПРОЦЕССОРА 8086/88

1.1 Микропроцессор с точки зрения программиста

Микропроцессор 8086/88 является типичным представителем 16-разрядных микропроцессоров (микропроцессор i8088 отличается от i8086 лишь 8-разрядной внешней мультиплексированной шиной данных). Следует отметить, что все сказанное ниже относится также к микропроцессору 8086, так как для программиста микропроцессоры 8088 и 8086 неразличимы. Все программы, написанные для микропроцессора 8088, могут выполняться на ЭВМ с микропроцессорами 80186, 80286 и т.д., так как система команд младших микропроцессоров этого семейства является подмножеством старших.

Для программиста микропроцессор представляется основным адресным пространством, адресным пространством внешних устройств и программно-доступными регистрами.

Микропроцессор 8088 характеризуется основным адресным пространством объемом 1 (MB) мегабайт, из которого первые 640 KB (килобайт) отведены под основную память (RAM), адресным пространством

ввода/вывода объемом 65536 байтов. Программно-доступными в микропроцессоре 8088 являются четыре регистра общего назначения AX, BX, CX, DX, два индексных регистра SI и DI, два регистра-указателя SP и BP и четыре сегментных регистра CS, DS, SS, ES. Косвенно программно-доступными являются также регистр-указатель команд IP и регистр флагов. Все указанные регистры являются 16-разрядными. Регистры общего назначения могут использоваться также 8-разрядными «половинками», причем младший байт обозначается буквой L, а старший байт – буквой H (например, для регистра AX – регистры AL и AH). Схематическое обозначение регистров микропроцессора приведено на рис. 1.

Использование регистров общего назначения, а также индексных регистров и регистров-указателей поясняется в описании команд. Здесь стоит лишь остановиться на регистре указателя стека. Стек – это память магазинного типа «первым вошел – последним вышел». Содержимое регистра указателя стека содержит адрес вершины стека. Более подробное описание работы со стеком содержится в описании соответствующих команд.

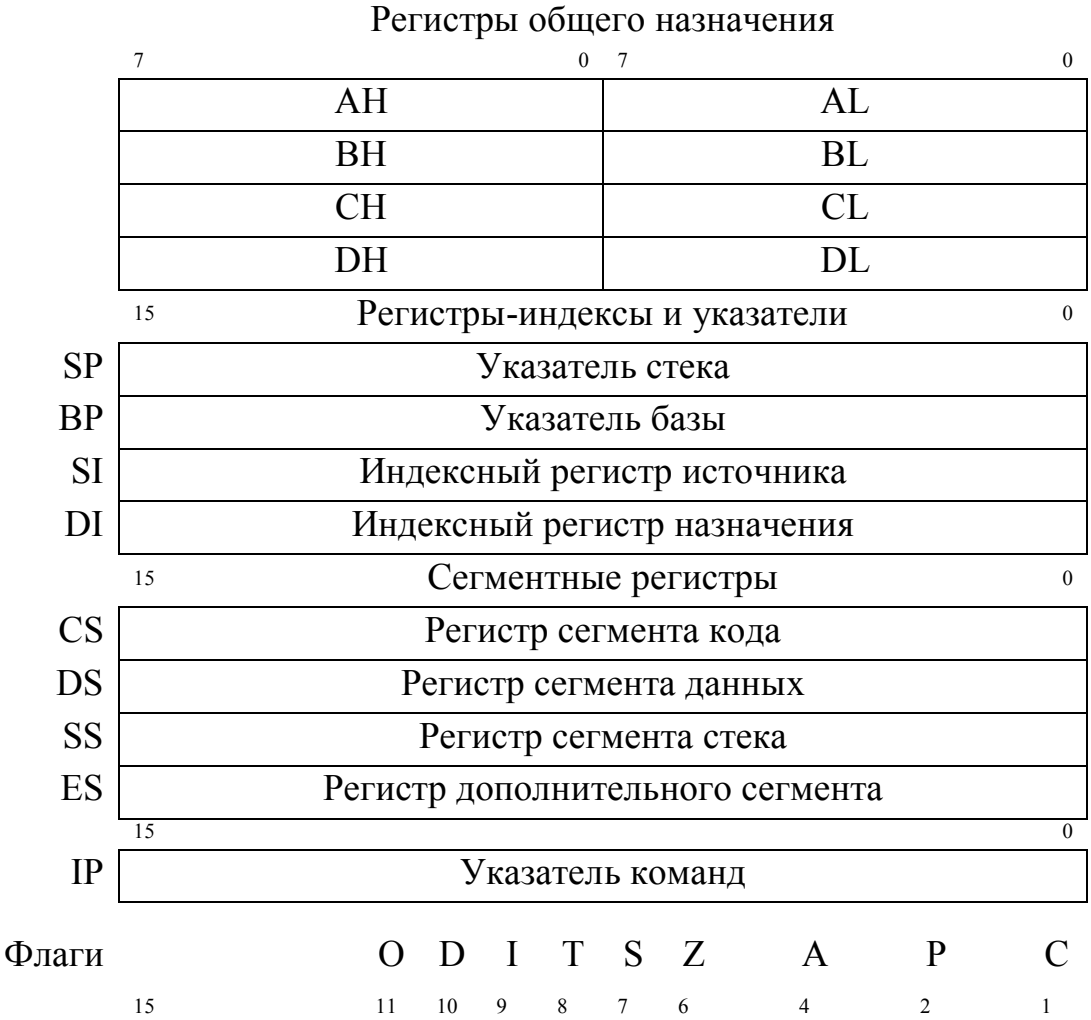


Рисунок 1.1 – Регистры микропроцессора 8086/88

Сегментные регистры используются для организации сегментов памяти. Необходимость в сегментной организации памяти обусловлена несоответствием объема основного адресного пространства микропроцессора (1 MB = 1 048 576 байтов) и размером адресного пространства, адресуемого 16-разрядными регистрами (64 KB = 65 536 байтов). Четыре сегментных регистра позволяют одновременно работать с четырьмя сегментами объемом 64 KB каждый – сегментом кода, сегментом данных, сегментом стека и дополнительным сегментом соответственно.

Сегмент кода служит обычно для размещения кодов программы, сегмент данных – для размещения различных данных, сегмент стека – для размещения стека, дополнительный сегмент – для использования в специальных случаях, а также в случаях, когда невозможно или неудобно использование других сегментов.

Регистр флагов содержит девять флагов:

CF – флаг переноса,

PF – флаг четности (паритета),

AF – флаг дополнительного переноса

ZF – флаг нуля

SF – флаг знака

TF – флаг ловушки

IF – флаг разрешения прерывания

DF – флаг направления

OF – флаг переполнения

Флаг переноса – индицирует перенос единицы из старшего разряда или заема единицы этим разрядом при арифметических операциях над 8- и 16-разрядными числами. При наличии переноса или заема флаг переноса устанавливается в единичное состояние. Этот флаг делает возможной многобайтную и многословную арифметику. Команды циклического сдвига могут изменять значение флага переноса. Имеются команды непосредственной установки (STC) и сброса (CLC) флага переноса.

Флаг четности (паритета) – индицирует четное число единиц в 8-разрядном числе или в младшем байте 16-разрядного. Этот флаг полезен при тестировании памяти и при контроле правильности передачи данных.

Флаг дополнительного переноса – индицирует наличие переноса из младшей тетрады 8-разрядного числа в старшую или заема – из старшей тетрады в младшую. Флаг полезен при использовании десятичной арифметики.

Флаг нуля – получает единичное значение при образовании всех нулевых битов в байте или в слове.

Флаг знака – индицирует единичное значение старшего бита результата одно- или двухбайтовой операции. В стандартном дополнительном коде единица в старшем разряде результата означает получение отрицательного числа.

Флаг ловушки – используется для реализации пошагового режима работы. При установленном флаге T микропроцессор вырабатывает сигнал внутреннего прерывания после выполнения каждой команды.

Флаг разрешения прерывания – используется для разрешения или запрещения внешнего маскируемого прерывания, поступающего по линии INTR. На немаскируемые внешние прерывания и на программные прерывания флаг не влияет. Имеются команды непосредственной установки (STI) и сброса (CLI) флага прерывания.

Флаг направления – используется обычно вместе со строковыми командами. При единичном значении флага изменение адресов в этих командах осуществляется от старших к младшим, при единичном значении – от младших к старшим. Команда STD устанавливает флаг направления в единичное значение, а команда CLD – в нулевое.

1.2 Способы адресации

Генерация физического адреса. Адресная шина микропроцессора 8086/88 является 20-разрядной и позволяет адресовать 1 мегабайт (1 048 576 байтов). В этом микропроцессоре используется сегментная организация памяти, причем каждый сегмент не превышает 64 килобайт (65536 байтов). Каждый сегмент должен начинаться с границы параграфа (1 параграф = 16 байтам). Так как в микропроцессоре 8088 имеется четыре сегментных регистра, микропроцессор одновременно имеет доступ к четырем сегментам – сегменту кода, сегменту данных, сегменту стека и дополнительному сегменту.

Сегменты могут располагаться в памяти произвольно, частично или полностью перекрываясь. Базовые адреса сегментов находятся в сегментных регистрах CS, DS, SS и ES соответственно.Физический 20-разрядный адрес складывается из адреса сегмента и смещения (рисунок 1.2).

Смещение или перемещаемый адрес операнда внутри сегмента образуется в микропроцессоре 8088 множеством различных способов, зависящих от способа адресации. Вычисленное значение этого смещения называется исполнительным адресом (The Effective Address – EA). Способ

адресации определяется вторым байтом команды, состоящим из трех полей (рисунок 1.3).

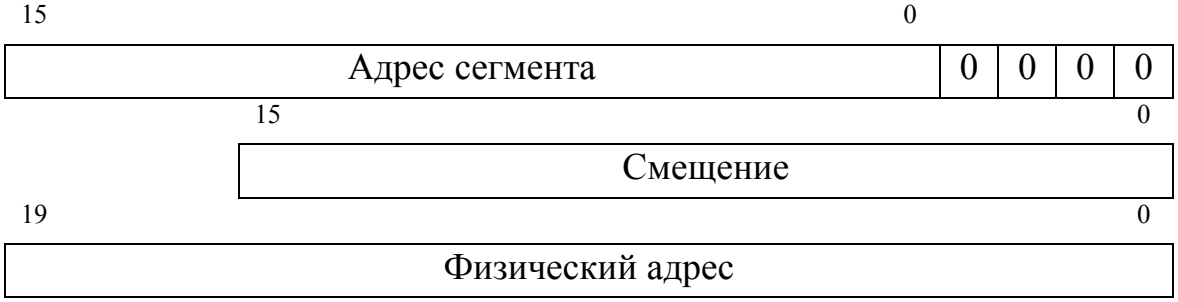


Рисунок 1.2 – Формирование физического адреса

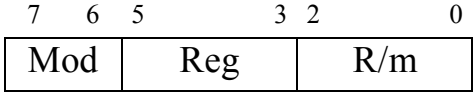


Рисунок 1.3 – Определение способа адресации

Двухразрядное поле **Mod** может принимать следующие значения:

00 – байтов смещения нет,

01 – следом идет один байт смещения со знаком,

10 – следом идут два байта смещения без знака,

11 – в команде используются регистровые операнды.

Трехразрядное поле Reg указывает регистр, содержащий операнд (8-битный при работе с байтами и 16-битный при работе со словами):

000 – AL или AX 100 – AH или SP

001 – CL или CX 101 – CH или BP

010 – DL или DX 110 – DH или SI

011 – BL или BX 111 – BH или DI

Трехразрядное поле R/m вместе с полем Mod определяет тип адресации (рисунок 1.4).

Reg	**Mod=00**	**Mod=01 или 10**	**Mod=11**
000	BX+SI	BX+SI + смещение	AL или AX
001	BX+DI	BX+DI + смещение	CL или CX
010	BP+SI	BP+SI + смещение	DL или DX
011	BP+DI	BP+DI + смещение	BL или BX
100	SI	SI + смещение	AH или SP
101	DI	DI + смещение	CH или BP
110	Прямая	BP + смещение	DH или SI
111	BX	BX + смещение	BH или DI

Рисунок 1.4 – Типы адресации

Прямая адресация. При использовании прямой адресации исполнительный адрес EA берется из поля смещения команды (рисунок 1.5).

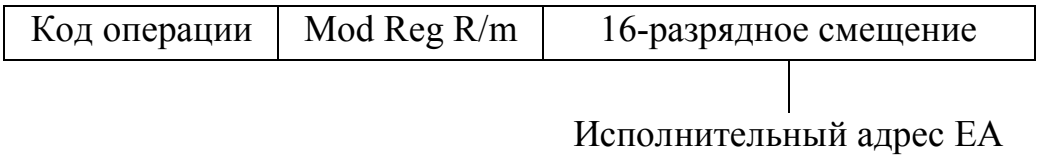


Рисунок 1.5 – Прямая адресация

При относительной адресации смещение определяется 8-битовым числом со знаком. В этом случае исполнительный адрес получается сложением этого смещения с содержимым указателя команд IP (рисунок 1.6).

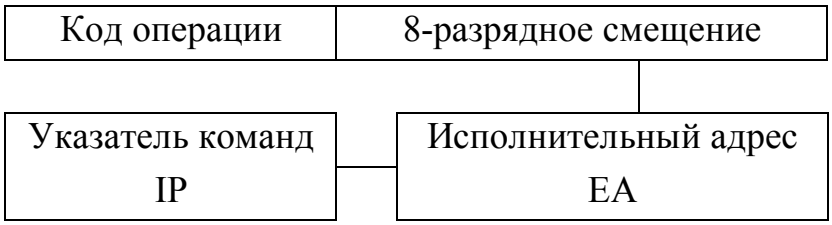


Рисунок 1.6 – Относительная адресация

При абсолютной адресации в команде указывается полный четырехбайтовый адрес, в котором младшее слово определяет смещение, а старшее – сегмент (ритсунок 1.7).

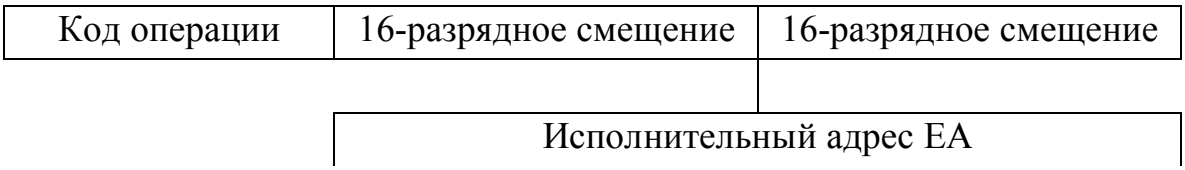


Рисунок 1.7 – Абсолютная адресация

Косвенная регистровая адресация. Исполнительный адрес берется в этом случае непосредственно из регистра BP, BX, SI или DI (рисуглк 1.8).

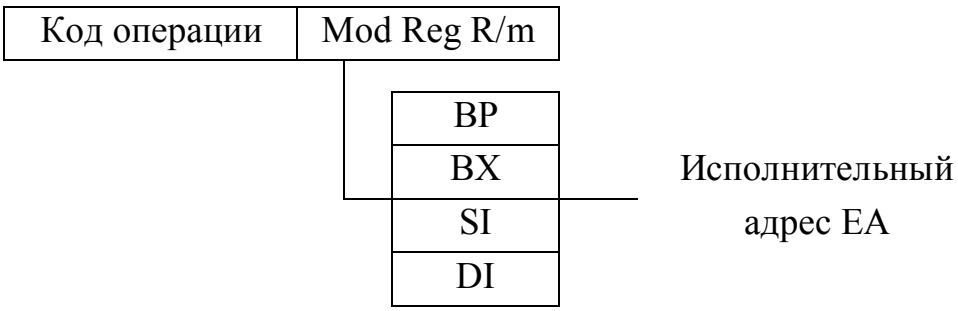


Рисунок 1.8 – Косвенная регистровая адресация

В одной и той же команде можно обрабатывать различные участки памяти, изменяя содержимое индексного регистра или регистра-указателя.

В командах безусловного перехода JMP и вызова процедуры CALL в качестве регистра косвенной адресации может быть использован любой 16-разрядный регистр общего назначения.

Базовая адресация. При базовой адресации эффективный адрес вычисляется как сумма смещения и содержимого регистра BP или BX. При использовании базового регистра BP вычисленный эффективный адрес относится к сегменту стека, если сегмент не был явно переопределен в команде (рисунок 1.9).

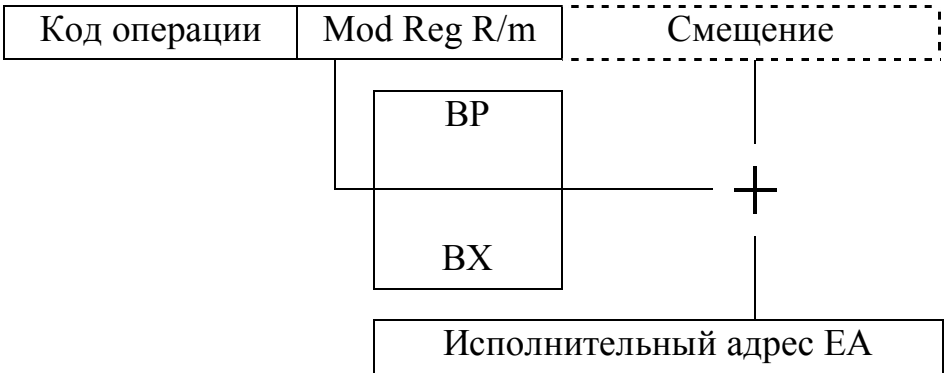
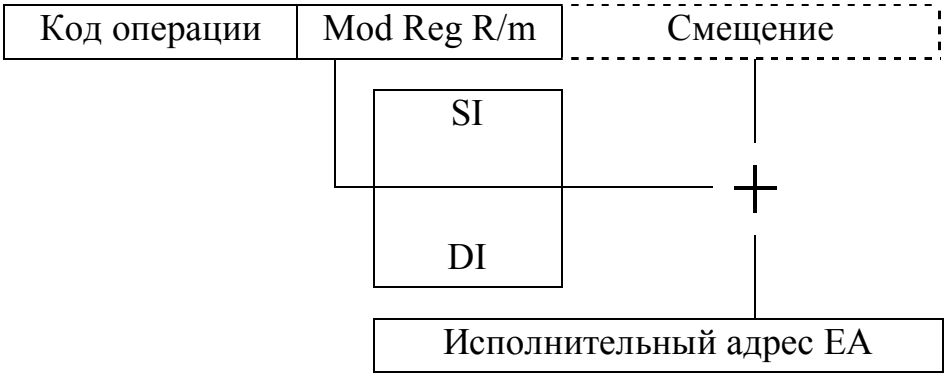


Рисунок 1.9 – Базовая адресация

Индексная адресация. При индексной адресации эффективный адрес вычисляется как сумма смещения и содержимого регистра SI или DI.



Рисуглк 1.10 – Индексеая адресация

Смещение может определять начало некоторого массива в памяти, а содержимое индексного регистра может указывать на конкретный элемент этого массива. Изменяя содержимое индексного регистра можно обращаться к различным элементам массива.

Базово-индексная адресация. При базово-индексной адресации эффективный адрес вычисляется как сумма базового регистра (BP или BX), индексного регистра(SI или DI) и смещения. Таким способом можно обрабатывать двумерные массивы (рисунок 1.11).

При использовании регистра BX эффективный адрес определяется в сегменте данных DS. При использовании регистра BP эффективный адрес определяется в сегменте стека SS.

Адресация строк. Строковые команды используют необычную адресацию операндов в памяти.

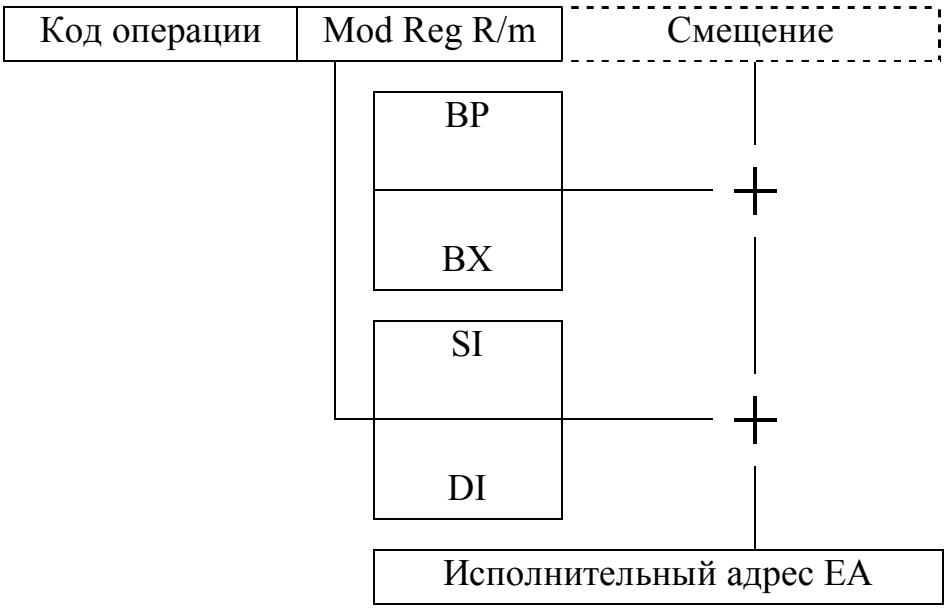


Рисунок 1.11 – Базово-индексная адресация

Индексный регистр SI используется для адресации байта или слова источника, а регистр DI – для адресации байта или слова назначения. При использовании префикса повторения в строковых командах эти регистры определяют начальные адреса байта или слова источника и назначения соответственно (рисунок 1.12).

Рисунок 1.12 – Адресация строк

Адресация портов ввода/вывода. Если порт расположен в адресном пространстве памяти, для его адресации может быть использован любой из описанных выше способов.

Для обращения к порту, расположенному в пространстве ввода/вывода могут использоваться два различных способа адресации. При прямой адресации порта номер порта указывается в непосредственном 8-битовом операнде. Таким образом может быть осуществлен доступ к портам с номерами от 0 до 255. Для косвенной адресации порта может быть использован регистр DX. Таким образом может быть осуществлен доступ к портам с номерами от 0 до 65535 (рисуноку 1.13).

1.3 Система команд

Команды пересылки данных. Команды пересылки данных позволяют пересылать байты, слова и двойные слова между регистрами и памятью, а также из регистра в регистр, из регистра в порт и наоборот.

Рисунок 1.13 – Адресация портов ввода-вывода

В группу пересылки данных включены также команды работы со стеком, команды ввода/вывода, команды пересылки содержимого регистра флагов, а также команды формирования указателей и загрузки сегментных регистров (рисунок 1.14).

Пересылка данных.

MOV (операнд назначения),(операнд-источник)

Команда MOV пересылает байт или слово из операнда-источника в операнд назначения.

PUSH (операнд-источник)

Команда PUSH уменьшает значение указателя SP стека на 2, а затем пересылает слово из операнда источника в стек.

POP (операнд назначения)

Команда POP берет слово с вершины стека и помещает его в операнд назначения, а затем увеличивает значение SP на 2.

XCHG (операнд назначения),(операнд-источник)

Команда XCHG меняет местами содержимое (байты или слова) операнда-источника и операнда назначения.

XLAT

Команда XLAT помещает в регистр AL байт из 256-байтовой таблицы, начальный адрес которой находится в регистре BX, а порядковый номер элемента таблицы – в регистре AL.

Ввод/вывод

IN (аккумулятор),(порт)

Команда IN передает байт или слово из порта ввода с указанным номером в регистр AL или AX. Номер порта может быть определен непосредственно в команде, в этом случае он может иметь номер от 0 до 255. Номер порта может находиться также в регистре DX, в этом случае порт может иметь номер от 0 до 65 535.

Пересылка данных	
MOV	Пересылка байта или слова

PUSH	Помещение слова в стек
POP	Извлечение слова из стека
XCHG	Обмен байтами или словами
XLAT	Выборка из таблицы
Ввод/вывод	
IN	Ввод байта или слова
OUT	Вывод байта или слова
Формирование указателей	
LEA	Загрузка эффективного адреса
LDS	Загрузка указателя с использованием DS
LES	Загрузка указателя с использованием ES
Пересылка содержимого регистра флагов	
LANF	Загрузка регистра AH из регистра флагов
SANF	Запись регистра AH в регистр флагов
PUSHF	Помещение регистра флагов в стек
POPF	Извлечение регистра флагов из стека

Рисунок 1.14 – Команды пересылки данных

OUT (порт),(аккумулятор)

Команда OUT передает байт или слово из регистра AL или AX в порт вывода с указанным номером. Возможности прямого или косвенного указания номера порта такие же, как в команде IN.

Формирование указателей

Эти команды формируют адреса переменных. Они могут быть полезны при обработке списков, массивов и строк.

LEA (операнд назначения),(операнд-источник)

Команда LEA (load effective address – загрузка эффективного адреса) пересылает смещение операнда-источника в операнд назначения. В качестве операнда-источника должен использоваться элемент памяти, а в качестве операнда назначения – 16-разрядный регистр общего назначения. Эта команда не затрагивает флаги. Команда LEA может быть использована, например, для инициализации регистра BX перед использованием команды XLAT.

LDS (операнд назначения),(операнд-источник)

Команда LDS (load pointer using DS – загрузка указателя с использованием DS) пересылает 32-разрядный указатель переменной из операнда-источника, расположенного в памяти, в операнд назначения и регистр DS. Слово смещения указателя пересылается в операнд назначения, который должен быть 16-разрядным регистром общего назначения. Слово сегмента

указателя пересылается в регистр DS. Использование этой команды с указанием в качестве операнда назначения регистра SI позволяет определить строку-источник для последующей строковой команды.

LES (операнд назначения),(операнд-источник)

Команда LES (load pointer using ES – загрузка указателя с использованием ES) пересылает 32-разрядный указатель переменной из операнда-источника, расположенного в памяти, в операнд назначения и регистр ES. Слово смещения указателя пересылается в операнд назначения, который должен быть 16-разрядным регистром общего назначения. Слово сегмента указателя пересылается в регистр ES. Использование этой команды с указанием в качестве операнда назначения регистра DI позволяет определить строку назначения для последующей строковой команды.

Пересылка содержимого регистра флагов

LAHF

Команда LAHF (загрузка регистра AH из регистра флагов) копирует флаги SF, ZF, AF, PF и CF в биты 7, 6, 4, 2 и 0 регистра AH. Содержимое битов 5, 3 и 1 неопределено.

SAHF

Команда SAHF (сохранение регистра AH в регистре флагов) пересылает биты 7, 6, 4, 2 и 0 регистра AH в SF, ZF, AF, PF и CF. Значение флагов OF, DF, IF и TF при этом остаются неизменными.

PUSHF

Команда PUSHF уменьшает значение указателя SP стека на 2, а затем пересылает все флаги в стек.

POPF

Команда POPF берет специальные биты из слова, расположенного на вершине стека и помещает их в регистр флагов, а затем увеличивает значение указателя стека SP на 2.

Арифметические команды показаны нарисунке 1.15. Такие команды могут обрабатывать четыре типа чисел – беззнаковые двоичные, знаковые двоичные, беззнаковые упакованные десятичные и беззнаковые неупакованные десятичные. Двоичные числа могут быть 8- и 16-разрядными. Десятичные упакованные числа содержат в байте две цифры, неупакованные – одну.

Беззнаковые 8-разрядные двоичные числа могут иметь значение от 0 до 255. Для представления беззнаковых чисел в диапазоне от 0 до 65 535 используются 16 разрядов. Над беззнаковыми двоичными числами могут выполняться операции сложения, вычитания, умножения и деления.

Знаковые двоичные числа (целые) также могут быть 8- и 16-разрядными. Самый старший (самый левый) бит знакового числа интерпретируется как знак этого числа: 0 – положительное число, 1 – отрицательное. Отрицательные числа представляются в стандартном двоичном дополнительном коде. Так как старший бит знакового числа используется для обозначения знака, диапазон представления 8-разрядных знаковых чисел от – 128 до + 127. 16-разрядное целое число представляется в диапазоне от – 32 768 до + 32 767. Нуль представляется положительным числом. Для знаковых чисел могут выполняться операции сложения, вычитания, умножения и деления.

Упакованные десятичные числа содержат в каждом байте две десятичных (0 – 9) цифры. В старшем полубайте содержится старшая значащая цифра, в младшем – младшая. Каждая десятичная цифра представляется в двоичном (или, что то же самое, в шестнадцатеричном) коде. Диапазон представления упакованных десятичных чисел в байте 0 – 99. Сложение и вычитание упакованных десятичных чисел осуществляется в два этапа.

Сначала байты складываются или вычитаются как беззнаковые двоичные числа, а затем соответствующая команда коррекции приводит результат к виду правильного упакованного десятичного числа. Команды коррекции для умножения и деления упакованных десятичных чисел отсутствуют.

Неупакованные десятичные числа содержат в байте одну десятичную цифру в младших четырех разрядах. Старшие четыре разряда должны быть нулями.

На рисунке 1.16 показана арифметическая интерпретация 8-разрядных двоичных чисел.

Неупакованное десятичное число легко может быть преобразовано в ASCII-представление соответствующей цифры. Для этого в старший байт неупакованного десятичного числа следует поместить значение 3.

Результаты арифметических команд воздействуют на состояние 6 флагов. Большая часть этих флагов может быть проанализирована после выполнения арифметических команд с помощью команд условного перехода, а также с помощью команды INTO (прерывание по переполнению). Воздействие арифметических команд на флаги описано ниже.

CF (флаг переноса): Если в результате сложения осуществляется перенос из старшего бита, флаг переноса взводится; в противном случае флаг переноса сбрасывается. При вычитании флаг переноса взводится, если осуществляется заем в старший бит результата; при отсутствии заема флаг сбрасывается. Следует иметь в виду, что при возникновении знакового переноса CF = OF

(флаг переполнения). Флаг переноса CF может использоваться для индикации беззнакового переполнения. Команды ADC (сложение с учетом разряда переноса) и SBB (вычитание с учетом разряда переноса) учитывают значение флага переноса, что позволяет реализовывать многобайтовые (например, 32- и 64-битовые) операции.

Сложение	
ADD	Суммирование байта или слова
ADC	Суммирование байта или слова с разр. переноса
INC	Увеличение байта или слова на 1
AAA	Коррекция сложения неупак. десятичных чисел
DAA	Коррекция сложения упак. десятичных чисел
Вычитание	
SUB	Вычитание байта или слова
SBB	Вычитание байта или слова с разр. переноса
DEC	Уменьшение байта или слова на 1
NEG	Инверсия байта или слова
CMP	Сравнение байта или слова
AAS	Коррекция вычитания неупак. десятичных чисел
DAS	Коррекция вычитания упак. десятичных чисел
Умножение	
MUL	Умножение беззнакового байта или слова
IMUL	Целочисленное умножение байта или слова
AAM	Коррекция умножения неупак. десятичных чисел
Деление	
DIV	Деление беззнакового байта или слова
IDIV	Целочисленное деление байта или слова
AAD	Коррекция деления неупак. десятичных чисел
CWB	Преобразование байта в слово
CWD	Преобразование слова в двойное слово

Рисунок 1.15 – Арифметические команды

AF (флаг дополнительного переноса): Если в результате ложения осуществляется перенос из младшего полубайта в старший, флаг дополнительного переноса взводится; в противном случае флаг сбрасывается. При вычитании флаг дополнительного переноса взводится, если осуществляется заем из старшего полубайта в младший; при отсутствии заема флаг сбрасывается. Флаг дополнительного переноса используется при десятичной коррекции операций.

Шестнадца-теричный код	Двоичный код	Беззнаковое двоичное	Знаковое двоичное	Неупакован-ное десятичное	Упакован-ное десятичное
07	00000111	7	+ 7	7	7
89	10001001	137	- 119	недействит.	89
C5	11000101	197	- 59	недействит.	недействит.

Рисунок 1.16 – Интерпретация 8-разрядных чисел

SF (флаг знака): После арифметических и логических операций флаг знака принимает значение старшего (7 или 15) бита результата. Для знаковых двоичных чисел флаг знака принимает значение 0 при положительном результате и 1 при отрицательном (если только не возникло переполнение). Команда условного перехода, выполняемая после операции со знаковыми числами, может использоваться для ветвления программы в зависимости от знака результата.

ZF (флаг нуля): Если в результате арифметической или логической операции получается нулевой результат, флаг нуля взводится; в противном случае флаг нуля сбрасывается. Команда условного перехода могут использоваться для ветвления программы в зависимости от равенства или неравенства нулю результата предыдущей операции.

PF (флаг четности): Если младшие 8 бит результата арифметической или логической операции содержат четное число единичных битов, флаг четности взводится; в противном случае флаг четности сбрасывается. Флаг четности может использоваться для проверки правильности принятого кода при передаче данных по линиям связи.

OF (флаг переполнения): Если в результате операции получается очень большое положительное число или очень маленькое отрицательное, которое не помещается в операнд назначения, флаг переполнения взводится; в противном случае флаг переполнения сбрасывается. Флаг переполнения индицирует знаковое арифметическое переполнение. Состояние этого флага может быть проверено командой условного перехода или командой INTO (прерывание по переполнению). Флаг переполнение может игнорироваться при выполнении операций с беззнаковыми числами.

Сложение

ADD (операнд назначения),(операнд-источник)

Сумма двух операндов, которые могут быть байтами или словами, помещается в операнд назначения. Оба операнда могут быть знаковыми или беззнаковыми числами. Команда ADD изменяет значение флагов AF, CF, OF, PF, SF и ZF.

ADC (операнд назначения),(операнд-источник)

Команда ADC (суммирование с учетом разряда переноса) суммирует операнды, которые могут быть байтами или словами, и добавляет 1, если установлен разряд переноса; результат помещается в операнд назначения. Оба операнда могут быть знаковыми или беззнаковыми числами. Команда ADD изменяет значение флагов AF, CF, OF, PF, SF и ZF. Так как команда ADC учитывает значение разряда переноса от предыдущей операции, это может быть использовано для организации суммирования чисел произвольной разрядности.

INC (операнд назначения)

Команда INC (инкремент) добавляет единицу к операнду назначения. Операнд может быть байтом или словом и трактуется как беззнаковое двоичное число. Команда INC изменяет значение флагов AF, OF, PF, SF и ZF; значение флага CF эта команда не изменяет.

AAA

Команда AAA (коррекция сложения неупакованных десятичных чисел) приводит содержимое регистра AL к виду правильного неупакованного десятичного числа, старший полубайт при этом обнуляется. Команда AAA изменяет значение флагов FC и AC; содержимое флагов OF, PF, SF и ZF после выполнения команды AAA неопределено.

DAA

Команда DAA (десятичная коррекция сложения) приводит содержимое регистра AL к виду правильного упакованного десятичного числа после предшествующей команды сложения. Команда DAA изменяет значение флагов AF, CF, PF, SF и ZF; содержимое флага OF после выполнения команды DAA не определено.

Вычитание

SUB (операнд назначения),(операнд-источник)

Содержимое операнда-источника вычитается из содержимого операнда назначения, и результат помещается в операнд назначения. Операнды могут быть знаковыми или беззнаковыми, двоичными или десятичными (см. команды AAS и DAS), однобайтовыми или двухбайтовыми числами. Команда SUB изменяет значение флагов AF, CF, OF, PF, SF и ZF.

SBB (операнд назначения),(операнд-источник)

Команда SBB (вычитание с учетом заема) вычитает содержимое операнда-источника из содержимого операнда назначения, затем вычитает из результата 1, если был установлен флаг переноса CF. Результат помещается на место операнда назначения.

Операнды могут быть знаковыми или беззнаковыми, двоичными или десятичными (см. команды AAS и DAS), однобайтовыми или двухбайтовыми числами. Команда SBB изменяет значение флагов AF, CF, OF, PF, SF и ZF. Команда SBB может быть использована для организации вычитания многобайтовых чисел.

DEC (операнд назначения)

Команда DEC (декремент) вычитает единицу из операнда назначения, который может быть одно- или двухбайтовым. Команда DEC изменяет содержимое флагов AF, OF, PF, SF и ZF. Содержимое флага CF при этом не изменяется.

NEG (операнд назначения)

Команда NEG (инверсия) вычитает операнд назначения, который может быть байтом или словом из 0 и помещает результат в операнд назначения. Такая форма двоичного дополнения числа пригодна для инверсии знака целых чисел. Если операнд нулевой, его знак не меняется. Попытка применить команду NEG к байтовому числу – 128 или к двухбайтовому числу – 32 768 не приводит к изменению значения операнда, но устанавливает флаг OF. Команда NEG воздействует на флаги AF, CF, OF, PF, SF и ZF. Флаг CF всегда установлен за исключением случая, когда операнд равен нулю, когда этот флаг сброшен.

CMP (операнд назначения),(операнд-источник)

Команда CMP (сравнение) вычитает операнд-источник из операнда назначения, не изменяя при этом значения операндов. Операнды могут быть байтовыми или двухбайтовыми числами. Хотя значения операндов на изменяются, значения флагов обновляются, что может быть учтено в последующих командах условного перехода. Команда CMP воздействует на флаги AF, CF, OF, PF, SF и ZF. При совпадении значений операндов взводится флаг ZF. Флаг переноса взводится, если операнд назначения меньше операнда-источника.

AAS

Команда AAS (коррекция вычитания неупакованных десятичных чисел) корректирует результат предшествующего вычитания двух правильных неупакованных десятичных чисел. Операндом назначения в команде вычитания должен быть регистр AL. Команда AAS приводит значение в AL к виду правильного неупакованного десятичного числа; старший полубайт при этом обнуляется. AAS воздействует на флаги AF и CF; Значение флагов OF, PF, SF и ZF после выполнения команды AAS неопределено.

DAS

Команда DAS (десятичная коррекция вычитания) корректирует результат предшествующего вычитания двух правильных неупакованных десятичных чисел. Операндом назначения в команде вычитания должен быть регистр AL. Команда DAS приводит значение в AL к виду двух правильных упакованных десятичных чисел. Команда DAS воздействует на флаги AF и CF. Значение флагов OF, PF, SF и ZF после выполнения команды DAS неопределено.

Умножение

MUL (операнд-источник)

Команда MUL (умножение) выполняет беззнаковое умножение операнда-источника и содержимого аккумулятора. Если операнд-источник однобайтовый, осуществляется умножение на содержимое регистра AL, а двухбайтовый результат возвращается регистрах AH и AL. Если операнд-источник двухбайтовый, существляется умножение на содержимое регистра AX, а четырехбайтовый результат возвращается в паре регистров DX и AX.

Операнды рассматриваются как беззнаковые двоичные числа. Если старшая половина результата (регистр AH при однобайтовом умножении и DX при двухбайтовом умножении) взводятся флаги CF и OF, в противном случае эти флаги сбрасываются.

Если после выполнения умножения взведены флаги CF и OF, это говорит о наличии значащих цифр результата в регистре AH или DX. Содержимое флагов AF, PF, SF и ZF после выполнения команды умножения неопределено.

IMUL (операнд-источник)

Команда IMUL (целочисленное умножение) выполняет знаковое умножение операнда-источника и содержимого аккумулятора. Если операнд-источник однобайтовый, осуществляется умножение на содержимое регистра AL, а двухбайтовый результат возвращается в регистрах AH и AL. Если операнд-источник двухбайтовый, осуществляется умножение на содержимое регистра AX, а четырехбайтовый результат возвращается в паре регистров DX и AX. Операнды рассматриваются как беззнаковые двоичные числа. Если старшая половина результата (регистр AH при однобайтовом умножении и DX при двухбайтовом умножении) взводятся флаги CF и OF, в противном случае эти флаги сбрасываются. Если после выполнения умножения взведены флаги CF и OF, это говорит о наличии значащих цифр результата в регистре AH или DX. Содержимое флагов AF, PF, SF и ZF после выполнения команды целочисленного умножения неопределено.

AAM

Команда AAM (коррекция умножения неупакованных десятичных чисел) приводит результат предшествующего умножения к двум правильным неупакованным десятичным цифрам. Для получения правильного результата после выполнения коррекции старшие полубайты умножаемых операндов должны быть нулевыми, а младшие должны быть правильными двоично-десятичными цифрами.

Команда AAM воздействует на флаги PF, SF и ZF. Содержимое флагов AF, CF и OF после выполнения команды AAM неопределено.

Деление

DIV (операнд-источник)

Команда DIV (деление) выполняет беззнаковое деление содержимого аккумулятора (и его расширения) на операнд-источник. Если операнд-источник однобайтовый, осуществляется деление двухбайтового делимого, расположенного в регистрах AH и AL. Однобайтовое частное получается в регистре AL, а однобайтовый остаток – в регистре AH. Если операнд-источник двухбайтовый, осуществляется деление четырехбайтового делимого, расположенного в регистрах DX и AX. Двухбайтовое частное при этом получается в регистре AX, а двухбайтовый остаток – в регистре DX. Если значение частного превышает разрядность аккумулятора (0FFh для однобайтового деления и 0FFFFh – для двухбайтового) или выполняется попытка деления на нуль, генерируется прерывание типа 0, а частное и остаток остаются неопределенными. Содержимое флагов AF, CF, OF, PF, SF и ZF после выполнения команды DIV неопределено.

IDIV (операнд-источник)

Команда IDIV (целочисленное деление) выполняет знаковое деление содержимого аккумулятора (и его расширения) на операнд-источник. Если операнд-источник однобайтовый, осуществляется деление двухбайтового делимого, расположенного в регистрах AH и AL. Однобайтовое частное получается в регистре AL, а однобайтовый остаток – в регистре AH. Для бйтового целочисленного деления положительное частное не может быть больше значения +127 (7Fh), а отрицательное не может быть меньше -127 (81h). Если операнд-источник двухбайтовый, осуществляется деление четырехбайтового делимого, расположенного в регистрах DX и AX. Двухбайтовое частное при этом получается в регистре AX, а двухбайтовый остаток – в регистре DX. Для двухбайтового целочисленного деления положительное частное не может быть больше значения +32767 (7FFFh), а отрицательное не может быть меньше значения -32767 (8001h). Если частное

положительное и превышает максимум или отрицательное и меньше минимума, генерируется прерывание типа 0, а частное и остаток остаются неопределенными. Частным случаем такого события является попытка деления на нуль. Содержимое флагов AF, CF, OF, PF, SF и ZF после выполнения команды IDIV неопределено.

AAD

Команда AAD (коррекция деления неупакованных десятичных чисел) модифицирует содержимое регистра AL перед выполнение деления так, чтобы при выполнении деления в частном получилось правильное неупакованное десятичное число. Для получения правильного результата после выполнения деления содержимое регистра AH должно быть нулевым. Команда AAD воздействует на флаги PF, SF и ZF. Содержимое флагов AF, CF и OF после выполнения команды AAD неопределено.

CBW

Команда CBW (преобразование байта в слово) расширяет знак байта в регистре AL на весь регистр AX. Команда CBW не воздействует на флаги. Команда CBW может быть использована для получения двухбайтового делимого из однобайтового перед выполнением команды деления.

CWD

Команда CWD (преобразование слова в двойное слово) расширяет знак слова в регистре AX на пару регистров AX и DX. Команда CWD не воздействует на флаги. Команда CWD может быть использована для получения четырехбайтового делимого из двухбайтового перед выполнением команды деления.

Команды работы с битами могут быть разбиты на три группы: логические команды, команды сдвига и команды циклического сдвига (рисунок 1.17).

Логические операции. К логическим операциям относятся булевы операции «НЕ», «И», «ИЛИ» и «ИСКЛЮЧАЮЩЕЕ ИЛИ». Кроме того к ним относится также команда «ТЕСТ», которая устанавливает флаги, но не изменяет ни одного из операндов.

Команды AND («И»), OR («ИЛИ»), XOR («ИСКЛЮЧАЮЩЕЕ ИЛИ») и TEST («ТЕСТ») воздействуют на флаги следующим образом:

Флаги переполнения (OF) и переноса (CF) после логических операций всегда сброшены, а флаг дополнительного переноса (AF) всегда неопределен.

Логические операции	
NOT	Инверсия байта или слова
AND	Операция “И” над байтами или словами

OR	Операция “ИЛИ” над байтами или словами
XOR	Операция “ИСКЛЮЧ. ИЛИ” над байтами или словами
TEST	Проверка байта или слова
Команды сдвига	
SHL/SAL	Логический/арифметич. сдвиг влево байта или слова
SHR	Логический сдвиг вправо байта или слова
SAR	Арифметический сдвиг вправо байта или слова
Команды циклического сдвига	
ROL	Циклический сдвиг влево байта или слова
ROR	Циклический сдвиг вправо байта или слова
RCL	Цикл. сдвиг влево байта или слова через разряд переноса
RCR	Цикл. сдвиг вправо байта или слова через разр. переноса

Рисунок 1.17 – Команды работы с битами

Флаги знака (SF), нуля (ZF) и четности (PF) всегда отражают результат логической операции и могут быть проверены последующей командой условного перехода. Интерпретация этих флагов такая же, как и после выполнения арифметических операций. Флаг знака (SF) взводится при единичном старшем бите результата и сбрасывается – при нулевом.

Флаг нуля (ZF) взводится при нулевом результате операции и сбрасывается в противном случае. Флаг четности (PF) взводится, если младший байт результата имеет четное число единиц, и сбрасывается при нечетном числе единиц.

Внимание! Операция NOT («НЕ») не влияет на флаги.

NOT (операнд назначения)

Команда NOT («НЕ») инвертирует биты (в форме дополнения до единицы) байта или слова операнда.

AND (операнд назначения),(операнд-источник)

Команда AND выполняет логическую операцию «И» над двумя операндами (байтами или словами), а результат возвращается в операнде назначения. Бит результата устанавливается только в том случае, если соответствующие биты операндов установлены. В противном случае бит результата сбрасывается.

OR (операнд назначения),(операнд-источник)

Команда OR выполняет логическую операцию «ИЛИ» над двумя операндами (байтами или словами), а результат возвращается в операнде назначения. Бит результата устанавливается в том случае, если установлен хотя

бы один соответствующий бит любого из операндов. В противном случае бит результата сбрасывается.

XOR (операнд назначения),(операнд-источник)

Команда XOR выполняет логическую операцию «ИСКЛЮЧАЮЩЕЕ ИЛИ» над двумя операндами (байтами или словами), а результат возвращается в операнде назначения. Бит результата устанавливается в том случае, если установлен соответствующий бит только одного из операндов. В противном случае бит результата сбрасывается.

TEST (операнд назначения),(операнд-источник)

Команда TEST выполняет логическую операцию «И» над двумя операндами (байтами или словами) не меняя при этом значений ни одного из операндов. Если после команды TEST выполняется команда JNZ, переход будет выполнен, если хотя бы одна пара соответствующих битов операндов установлена.

Команды сдвига. Биты в байте или слове могут сдвигаться арифметически или логически. Максимальное может быть выполнено 255 в одной команде. Счетчик сдвига может быть определен в команде либо как константа 1, либо в регистре CL. Арифметические сдвиги могут использоваться для умножения и деления двоичных чисел на степени двойки.

Команды сдвига воздействуют на флаги следующим образом:

- флаг дополнительного переноса AF после операции сдвига всегда неопределен;
- флаги PF, SF и ZF принимают стандартные значение, как после любых логических операций;
- флаг переноса CF всегда содержит последний выдвинутый из операнда назначения бит.

SHL/SAL (операнд назначения)(счетчик)

Команды SHL (логический сдвиг влево) и SAL (арифметический сдвиг влево) полностью идентичны. Байт или слово операнда назначения сдвигается влево на количество разрядов, определяемое операндом-счетчиком. Освобождающиеся младшие биты операнда назначения заполняются нулями. Если знаковый бит во время операции не изменился, флаг OF сбрасывается.

SHR (операнд назначения)(счетчик)

Команда SHR (логический сдвиг вправо) сдвигает байт или слово операнда назначения вправо на количество разрядов, определяемое операндом-счетчиком. Освобождающиеся старшие биты операнда назначения заполняются

нулями. Если знаковый бит во время операции не изменился, флаг OF сбрасывается.

SAR (операнд назначения)(счетчик)

Команда SAR (арифметический сдвиг вправо) сдвигает байт или слово операнда назначения вправо на количество разрядов, определяемое операндом-счетчиком. Освобождающиеся старшие биты операнда назначения заполняются исходным значением знакового (старшего значащего бита) операнда. Следует отметить, что эта команда не эквивалентна команде целочисленного деления (IDIV) на соответствующую степень двойки. Так, например, при сдвиге вправо на 1 разряд значения – 5 в результате получается значение – 3, в то время, как при делении должно было получиться -2.

Команды циклического сдвига. Биты в байте или в слове могут сдвигаться также циклически. При циклическом сдвиге «выдвигаемые» из операнда биты не теряются, как при простом сдвиге, в «вдвигаются» в операнд с другого его конца. Как и в командах простого сдвига количество разрядов сдвига определяется операндом-счетчиком, который может определяться непосредственной константой 1 или регистром CL. Команда циклического сдвига воздействуют только на флаги переноса CF и переполнения OF. Флаг переноса CF всегда содержит значение последнего «выдвинутого» бита. Значение флага переполнения OF при многобитовом сдвиге неопределено, при однобитовом сдвиге OF устанавливается, если старший (знаковый) бит изменяется во время операции. Если старший бит не меняется, флаг переполнения сбрасывается.

ROL (операнд назначения)(счетчик)

Команда ROL (циклический сдвиг влево) циклически сдвигает содержимое операнда назначение влево на количество разрядов, определенное в операнде-счетчике.

ROR (операнд назначения)(счетчик)

Команда ROR (циклический сдвиг вправо) циклически сдвигает содержимое операнда назначение вправо на количество разрядов, определенное в операнде-счетчике.

RCL (операнд назначения)(счетчик)

Команда RCL (циклический сдвиг влево через разряд переноса) циклически сдвигает содержимое операнда назначение влево на количество разрядов, определенное в операнде-счетчике. При этом флаг переноса является частью операнда назначения, то есть, значение флага переноса CF при сдвиге

переносится в младший значащий бит операнда, а сам флаг принимает значение старшего значащего бита байта или слова.

RCR (операнд назначения)(счетчик)

Команда RCR (циклический сдвиг вправо через разряд переноса) в точности соответствует команде RCL, лишь с той разницей, что сдвиг производится вправо.

Строковые команды (рисунок 1.18). Базовые строковые команды осуществляют элементарную операцию со строками байтов или слов, выполняя каждый раз действие только с одним элементом. При помощи этих команд могут быть обработаны строки длиной до 128 Кбайт. Сводная таблица строковых команд приведена ниже.

Строковая команда может иметь операнд-источник, операнд назначения или оба. Операнд-источник по умолчанию всегда находится в текущем сегменте данных. При использовании сегментного префикса это назначение может быть изменено.

Операнд назначения всегда должен находиться в текущем дополнительном сегменте. Ниже в таблице показано использование регистров процессора в строковых командах.

Строковые команды автоматически изменяют значение регистров SI и/или DI так, чтобы каждый из этих регистров указывал на очередной элемент строки. Значение флага направления DF определяет направление изменения содержимого регистров SI и DI. Если флаг направления сброшен (DF = 0) автоматически выполняется инкремент регистров SI и DI. При установленном флаге направления (DF = 1) автоматически выполняется декремент этих регистров. Если в команде используется префикс повторения, значение регистра CX уменьшается на 1 после выполнения каждой строковой команды.

REP/REPE/REPZ/REPNE/REPNZ

Префикс повторить/повторить пока равно/повторить пока нуль/повторить пока не равно/повторить пока не нуль определяет условие повторения строковой команды.

Префикс REP используется в сочетании с командами MOVS (переслать строку) и STOS (сохранить строку). При этом он трактуется «Повторять, пока не кончилась строка» (CX не нуль).

Префиксы REPE и REPZ совершенно идентичны. Они используются с командами CMPS (сравнить строку) и SCAS (просканировать строку) и действуют точно так же, как префикс REP, однако перед каждым следующим

повторением проверяется состояние флага нуля (FZ). Если флаг взведен, команда повторяется, если флаг сброшен, повторения не будет.

Строковые операции	
REP	Повторение
REPE/ REPZ	Повторение, пока равно/пока нуль
REPNE/ REPNZ	Повторение, пока не равно/пока не нуль
MOVS	Строковая пересылка байта или слова
MOVSB/ MOVSW	Строковая пересылка байта/слова
CMPS	Строковое сравнение байта или слова
SCANS	Строковое сканирование байта или слова
STOS	Строковая загрузка байта или слова
TEST	Строковое сохранение байта или слова
Использование регистров в строковых командах	
SI	Индекс (смещение) строки-источника
DI	Индекс (смещение) строки назначения
CX	Счетчик повторений
AL/AX	Значение сканирования Регистр назначения для LODS Регистр-источник для STOS
Флаги	
DF	0 – автоинкремент SI, DI 0 – автодекремент SI, DI
ZF	Завершение сканирования/сравнения

Рисунок 1.18 – Строковые команды

Префиксы REPNE и REPNZ также идентичны. Они совпадают с префиксами REPE и REPZ, однако действие флага FZ на них прямо противоположно.

MOVS (строка назначения),(строка-источник)

Команда MOVS (переслать строку) пересылает байт или слово из операнда-источника (адресуемого регистром SI) в операнд назначения (адресуемый регистром DI) и изменяет содержимое этих регистров так, чтобы каждый указывал на следующий элемент строки. При использовании этой команды вместе с префиксом повторения осуществляется блочная пересылка

память-память. Операнды, указанные в команде определяют только формат пересылаемой единицы данных (байт или слово).

MOVSB, MOVSW

Команды MOVSB (переслать строку байтов) и MOVSW (переслать строку слов) полностью соответствуют команде MOVS, но не имеют операндов, так как формат пересылаемых данных явно указан в мнемонике команды (MOVSB – байты, MOVSW – слова).

CMPS (строка назначения),(строка-источник)

Команда CMPS (сравнить строки) вычитает байт или слово источника (адресуемого регистром SI) из байта или слова назначения (адресуемого регистром DI). Сами операнды при этом не меняются. Команда CMPS воздействует на флаги AF, CF, OF, PF, SF и ZF, а также изменяет содержимое регистров SI и DI так, чтобы каждый из них указывал на следующий элемент соответствующей строки. Действие команды CMPS на флаги аналогично действию команды CMP. Если команде CMPS предшествует префикс REPE или REPZ, операция трактуется так: «производить сравнение пока не достигнут конец строки (CX не равен нулю) и пока элементы строк равны». Если команде предшествует префикс REPNE или REPNZ – трактовка такова: «производить сравнение пока не достигнут конец строки и элементы строк различны».

SCAS (строка назначения)

Команда SCAS (сканировать строку) вычитает элемент строки назначения (байт или слово), адресуемый регистром DI, из содержимого регистра AL (если строка состоит из байтов) или AX (если строка состоит из слов). Содержимое элемента строки назначения и аккумулятора при этом остается неизменным. Команда SCAS воздействует на флаги AF, CF, OF, PF, SF и ZF, а также изменяет содержимое регистра DI так, чтобы он указывал на следующий элемент строки. Если команде SCAS предшествует префикс REPE или REPZ, операция трактуется так: «производить сравнение пока не достигнут конец строки (CX не равен нулю) и пока элементы строки равны содержимому аккумулятора». Если команде предшествует префикс REPNE или REPNZ трактовка такова: «производить сравнение пока не достигнут конец строки и элементы строки не равны содержимому аккумулятора».

LODS (строка-источник)

Команда LODS (загрузка строки) пересылает элемент строки (байт или слово), адресуемый регистром SI в регистр AL или AX, и изменяет содержимое регистра SI так, чтобы он указывал на следующий элемент строки.

STOS (строка назначения)

Команда STOS (сохранение строки) пересылает (байт или слово) из регистра AL или AX в элемент строки, адресуемый регистром DI, и изменяет содержимое регистра DI так, чтобы он указывал на следующий элемент строки. Эта команда удобна для инициализации некоторой области памяти какой-либо константой.

Команды передачи управления. Порядок выполнения команд в процессорах 80x86 и 80x88 определяется содержимым регистра сегмента кода (CS) и счетчика команд (IP). Регистр CS содержит базовый адрес текущего сегмента кода, т.е. 64-килобайтного фрагмента памяти, из которого в данный момент извлекаются коды команд. Содержимое счетчика команд IP используется как смещение относительно начала текущего сегмента кода. Содержимое CS и IP однозначно определяет то место в памяти, из которого будет извлечена следующая команда. Команда передачи управления (команды переходов) изменяют содержимое регистров CS и IP. Всего имеется четыре группы команд передачи управления, которые приведены в таблице.

Команды безусловного перехода (рисунок 1.19). Команда безусловного перехода передает управление указанной команде в том же сегменте (внутрисегментный переход) или за его пределы (межсегментный переход). Внутрисегментный переход обычно называется ближним (NEAR), а межсегментный – дальним (FAR).

CALL	Вызов процедуры
RET	Возврат из процедуры
JMP	Переход

Рисунок 1.19 – Команды безусловного перехода

CALL (имя процедуры)

Команда CALL передает управление внешней процедуре, предварительно сохранив в стеке информацию для последующего возврата в вызывающую процедуру при помощи команды RET. Команда CALL имеет различную форму записи в зависимости от типа вызываемой процедуры (дальняя или ближняя). Команда RET, которой завершается вызываемая процедура, должна иметь тот же тип (дальний или ближний), что и вызывающая процедуру команда CALL. Адрес вызываемой процедуры может быть задан непосредственно в команде CALL, в памяти или в регистре.

При внутрисегментной непосредственной команде CALL в стеке сохраняется текущее содержимое счетчика команд (IP), который указывает на первый байт следующей за CALL командой. Относительное смещение

вызываемой процедуры содержится в самой команде (диапазон плюс – минус 32К).

При внутрисегментной косвенной команде CALL в стеке сохраняется текущее содержимое счетчика команд (IP), который указывает на первый байт следующей за CALL командой. Относительное смещение вызываемой процедуры может содержаться в слове памяти или в 16-разрядном регистре.

При межсегментной непосредственной команде CALL в стеке сохраняется текущее содержимое регистра CS, в регистр CS помещается значение сегмента из команды CALL, затем в стеке сохраняется текущее содержимое регистра IP, и в него записывается значение смещения из команды.

При межсегментной косвенной команде CALL происходит то же, что описано выше, но значения сегмента и смещения берутся из памяти или из регистров, причем первое слово содержит смещение, а второе – сегмент вызываемой процедуры.

RET (необязательное значение)

Команда RET возвращает управление из вызванной процедуры команде, следующей за командой CALL. Если возврат осуществляется из ближней процедуры, возврат является внутрисегментным (содержимое регистра CS остается неизменным). При возврате из дальней процедуры возврат является межсегментным (из стека восстанавливаются значения CS и IP). Если в команде задано необязательное значение, команда RET добавляет это значение к указателю стека SP. Это позволяет пропускать параметры, передаваемые через стек перед командой CALL.

JMP (цель)

Команда JMP осуществляет безусловную передачу управления на указанный адрес. В отличие от команды CALL команда JMP не сохраняет в стеке информацию об адресе возврата. Так же, как в команде CALL адрес целевого операнда может быть указан непосредственно в команде (непосредственная команда JMP), а также в памяти или в регистре (косвенная команда JMP).

При внутрисегментной непосредственной команде JMP к счетчику команд IP добавляется смещение, указанное в команде. Если компилятор обнаруживает, что целевой адрес находится на расстоянии меньшем, чем 127 байтов от команды, он автоматически генерирует двухбайтовый вариант команды, называемый коротким переходом (SHORT JMP), в противном случае генерируется ближний переход (NEAR JMP), в котором диапазон перехода составляет плюс-минус 32К.

При внутрисегментной косвенной команде JMP смещение, добавляемое к регистру IP, может быть указано в памяти или в 16-разрядном регистре. В последнем случае значение смещение берется из регистра, указанного в команде.

При межсегментной непосредственной команде JMP значения IP и CS заменяются значениями, указанными в команде.

При межсегментной косвенной команде JMP значения IP и CS могут быть заменены только значениями расположенными в памяти. При этом первое слово двойного слова содержит смещение, второе слово – сегмент.

Команда JMP осуществляет безусловную передачу управления на указанный адрес. В отличие от команды CALL команда JMP не сохраняет в стеке информацию об адресе возврата. Так же, как в команде CALL адрес целевого операнда может быть указан непосредственно в команде (непосредственная команда JMP), а также в памяти или в регистре (косвенная команда JMP).

Команды управления циклами позволяют организовать простые или сложные циклы (рисунок 1.20).

LOOP	Цикл
LOOPE/ LOOPZ	Цикл если равно/если нуль
LOOPNE/ LOOPNZ	Цико если не равно/если не нуль
JCXZ	Переход если содержимое регистра CX равно нулю

Рисунок 1.20 – Команды управления циклами

LOOP (цель)

У этой команды один операнд — имя метки (цель), на которую осуществляется переход. В качестве счётчика цикла используется регистр CX. Команда LOOP выполняет декремент CX, а затем проверяет его значение. Если содержимое CX не равно нулю, то осуществляется переход на метку, иначе управление переходит к следующей после LOOP команде.

Содержимое CX интерпретируется командой как число без знака. В CX нужно помещать число, равное требуемому количеству повторений цикла. Понятно, что максимально может быть 65535 повторений. Ещё одно ограничение связано с дальность перехода. Метка должна находиться в диапазоне -127…+128 байт от команды LOOP

LOOPE/LOOPZ (цель)

Выполняется так же, как **LOOP**, но дополнительным условием перехода на метку является взрведенный флаг нуля.

LOOPNE/LOOPNZ (цель)

Выполняется так же, как **LOOP**, но дополнительным условием перехода на метку является сброшенный флаг нуля.

JCXZ (цель)

Переход выполняется, если содержимое регистра CX равно нулю. Эта команда часто используется для выхода из цикла по условию.

Команды условного перехода (рисунок 1.21). Команды условного перехода выполняют или не выполняют передачу управления на указанный адрес в зависимости от состояния флагов процессора не момент выполнения команды. Эти команды проверяют различные комбинации флагов и условий (рисунок 1.22). Если условие истинно, осуществляется передача управления на указанный адрес. Если условие неверно, управление передается команде, следующей за командой условного перехода.

JA/JNBE	Переход если выше/если не ниже или равно
JAE/JNB	Переход если выше или равно/если не ниже
JB/JNAE	Переход если ниже/если не выше или равно
JBE/JNA	Переход если ниже или равно/если не выше
JC	Переход если перенос
JE/JZ	Переход если равно/если нуль
JG/JNLE	Переход если больше/если не меньше или равно
JGE/JNL	переход если больше или равно/если не меньше
JL/JNGE	Переход если меньше/если не больше или равно
JLE/JNG	Переход если меньше или равно/если не больше
JNC	Переход если нет переноса
JNE/JNZ	Переход если не равно/если не нуль
JNO	Переход если не переполнение
JNP/JPO	Переход если нечетно
JNS	Переход если не знак
JO	Переход если переполнение
JP/JPE	переход если четно
JS	Переход если знак

Рисунок 1.21 – Команды условного перехода

Все команды условного переходя являются короткими (SHORT), так что диапазон переходов в этих командах лежит в диапазоне от -128 до +127 байтов.

При этом следует иметь в виду, что команде JMP 00h соответствует переход на следующую команду.

Мнемон.	Проверяемое условие	Переход, если ...
JA/JNBE	(CF or ZF) = 0	выше/не ниже или равно
JAE/JNB	CF = 0	выше или равно/не ниже
JB/JNAE	CF = 1	ниже/не выше или равно
JBE/JNA	(CF or ZF) = 1	ниже или равно/не выше
JC	CF = 1	перенос
JE/JZ	ZF = 1	равно/нуль
JG/JNLE	[(SF xor OF) or ZF] = 0	больше/не меньше или равно
JGE/JNL	(SF xor OF) = 0	больше или равно/не меньше
JL/JNGE	(SF xor OF) = 1	меньше/не больше или равно
JLE/JNG	[(SF xor OF) or ZF] = 1	меньше или равно/не больше
JNC	CF = 0	нет переноса
JNE/JNZ	ZF = 0	не равно/не нуль
JNO	OF = 0	не переполнение
JNP/JPO	PF = 0	нечетно
JNS	SF = 0	не знак
JO	OF = 1	переполнение
JP/JPE	PF = 1	четно
JS	SF = 1	знак

Рисунок 1.22 – Флаги, проверяемые командами условного перехода

Команды прерывания (рисунок 1.23). Команды прерывания позволяют вызывать процедуры обслуживания прерываний из программ так же как это сделало бы устройство. Программные прерывания имитируют действие аппаратных прерываний.

Прерывания	
INT	Прерывание
INTO	Прерывание если переполнение
IRET	Возврат из прерывания

Рисунок 1.23 – Команды прерывания

INT (тип прерывания)

Команда INT (прерывание) инициирует выполнение процедуры обработки прерывания, определенного в операнде «тип прерывания». Эта команда сохраняет в стеке регистр флагов, очищает флаги TF и IF для запрещения пошагового выполнения и маскируемых прерываний. Флаги сохраняются в том же формате, что и в команде PUSHF. Затем в стеке

сохраняется текущее содержимое регистра сегмента кода CS, вычисляется адрес вектора прерывания путем умножения «типа прерывания» на четыре, и второе слово этого вектора помещается в регистр сегмента кода CS. Далее в стеке сохраняется текущее содержимое счетчика команд IP, и в этот регистр записывается первое слово вычисленного вектора прерывания.

INTO

Команда INTO (прерывание при переполнении) генерирует программное прерывание, если установлен флаг переполнения (OF), в противном случае управление передается следующей команде. Вектор прерывания INTO расположен по адресу 10h. Действие этой команды аналогично действию команды INT.

IRET

Команда IRET (возврат из прерывания) возвращает управление в точку, откуда прерывание было вызвано, заполняя из стека регистры IP, CS и регистр флагов. Команда IRET используется для выхода из процедур обработки как программных, так и аппаратных прерываний.

Прочие команды. К прочим командам относятся команды, приведенные ниже на р сунке 1.24.

Прочие команды	
STC	Установка флага переноса
CLC	Сброс флага переноса
CMC	Инверсия флага переноса
STD	Установка флага направления
CLD	Сброс флага направления
STI	Установка флага разрешения прерываний
CLI	Сброс флага разрешения прерываний
HLT	Останов процессора
NOP	Отсутствие операции

CLC

Команда CLC (очистка флага переноса) обнуляет флаг переноса CF и не воздействует нина какие другие флаги.

CMC

Команда CMC (инверсия флага переноса) изменяет значение флага переноса CF на противоположное и не воздействует ни на какие другие флаги.

STC

Команда STC (установка флага переноса) устанавливает флаг переноса CF в 1 и не воздействует нина какие другие флаги.

CLD

Команда CLD (сброс флага направления) сбрасывает флаг переноса DF в 0 и не воздействует нина какие другие флаги.

При сброшенном флаге направления выполнение строковых команд сопровождается автоинкрементом регистра SI и/или DI.

STD

Команда STD (установка флага направления) устанавливает флаг переноса DF в 1 и не воздействует нина какие другие флаги. При установленном флаге направления выполнение строковых команд сопровождается автодекрементом регистра SI и/или DI.

CLI

Команда CLI (сброс флага разрешения прерываний) сбрасывает флаг переноса IF в 0 и не воздействует нина какие другие флаги. При сброшенном флаге разрешения прерывания процессор не реагирует на внешние запросы прерывания, поступающие на вход INTR. Этот запрет не касается немаскируемого прерывания NMI и программных прерываний.

STI

Команда STI (установка флага разрешения прерываний) устанавливает флаг переноса IF в 1 и не воздействует нина какие другие флаги. При установленном флаге разрешения прерывания разрешены все прерывания.

HLT

Команда HLT (останов) переводит процессор в состояние останова, из которого он может быть выведен только сбросом или сигналом запроса прерывания. Команда HLT не воздействует ни на какие флаги.

NOP

Команда NOP (нет операции) не влияет ни на работу процессора ни на флаги.

1.4 Вопросы для самопроверки

1. Микропроцессор 80x86, 80x88
2. Распределение памяти в IBM-совместимых ЭВМ
3. Регистры микропроцессора 80x86
4. Способы адресации в микропроцессоре 80x86
5. Формирование исполнительного алреса в 80x86
6. Команды пересылки в 80x86

7. Арифметические команды в 80x86
8. Логические команды в 80x86
9. Команды передачи управления в 80x86
10. Процедуры в ассемблере
11. Векторы прерывания
12. Дальние, ближние и короткие переходы
13. Текстовый видеобуфер
14. Строковые команды в 80x86
15. Графический видеобуфер
16. Префиксы повторения
17. Префиксы-указатели
18. Аппаратные прерывания
19. Команды десятичной арифметики
20. Умножение в 80x86
21. Деление в 80x86
22. Условные переходы в 80x86
23. Команды ввода-вывода в 80x86
24. Команды возврата из процедуры
25. Работа с регистром флагов.
26. Вывод динамических объектов на экран.
27. Обработка клавиатуры в программе пользователя.
28. Экранная область памяти.
29. Чем отличается графический режим от текстового ?
30. Этапы реализации движения изображения.
31. Десятичная коррекция.
32. Порты ввода-вывода.
33. Аппаратные и программные прерывания.
34. Подпрограммы работы с экраном.
35. Код возврата программы.
36. Запуск программы типа .com.
37. Что нужно для создания файла?
38. Запуск программы типа .exe.
39. Строковые команды
40. Базово-индексная адресация
41. Файловые функции DOS
42. Формирование эффективного адреса
43. Вывод данных в порт

2 ФУНКЦИИ DOS И BIOS

В этой и следующей главах рассмотрены наиболее часто используемые функции DOS и BIOS, вызываемые из прикладных программ командами int n. Рассмотрены некоторые особенности этих функций.

2.1 Функции BIOS

int 05h: Печать экрана

Прерывание int 5 используется в PC для вызова программы ROM BIOS, печатающей экран. Это прерывание вызывается обработчиком прерывания int 9 при распознавании клавиши PrtSc.

Оно может также вызываться из прикладной программы. Команда DOS «Graphics» заменяет эту программу своей, которая выдает графический экран (в точечном представлении) на IBM-совместимый графический принтер.

int 08h: Прерывание от таймера

Это аппаратно генерируемое прерывание (IRQ 0) вызывается по каждому тику часов реального времени PC. Часы тикают каждые 55мс, или около 18.2 раз в секунду. Подпрограмма обработки этого прерывания обновляет значение часов на 0:046 сек. Эта же подпрограмма выключает двигатели гибких дисков по истечении примерно двух секунд без операций ввода/вывода.

int 09h: Прерывание от клавиатуры

Это аппаратно генерируемое прерывание (IRQ 1) выполняется при каждом нажатии и отпускании клавиши. Подпрограмма BIOS интерпретирует это событие, сохраняя значения в буфере клавиатуры по адресу 0:041e. Она обрабатывает также специальные случаи клавиш PrtSc и SysReq, и отслеживает состояние клавиш Shift и различных Lock.

int 10H: Видео сервис

В таблице 2.1 приведен перечень функций видеосервиса, предоставляемых подпрограммами BIOS. Номер функции определяется значением регистра ah при вызове int 10.

Таблица 2.1 – Функции видеосервиса

ah	Функция
00h	Установить видеорежим
01h	Установить размер и форму курсора
02h	Установить позицию курсора
03h	Читать позицию курсора
04h	Читать световое перо
05h	Выбрать активную страницу дисплея
06h	Скроллинг окна вверх (или очистка его)
07h	Скроллинг окна вниз (или очистка его)
08h	Читать символ/атрибут
09h	Вывести символ/атрибут
0ah	Вывести символ
0bh	Выбрать палитру/цвет рамки
0ch	Вывести графическую точку
0dh	Читать графическую точку
0fh	Вывести символ в режиме TTY
10h	Читать видео режим
11h	EGA установить палитру
12h	EGA специальные функции
13h	Писать строку (только AT + EGA)

2.2 Подробное описание видеосервиса

Здесь детализированы функции INT 10H стандартного видеосервиса ROM-BIOS.

AH = 00H Установка видеорежима.

Замечание: для «EGA» и «Jr» можно добавить 80H к AL, чтобы инициализировать видео режим без очистки экрана.

AH = 01H Установка размера/формы курсора (текст). Курсор, если он видим, всегда мерцает.

Вход:CH = начальная строка (0-1fH; 20H=подавить курсор)

CL = конечная строка (0-1fH)

Вход: AL = видеорежим (см. таблицу 2.2).

Таблица 2.2 – Таблица видеорежимов

AL	Тип	Формат	Цветов	Адаптер	Адрес
0	текст	40 x 25	16/8	CGA,EGA	b800

1	текст	40 x 25	16/8	CGA,EGA	b800
2	текст	80 x 25	16/8	CGA,EGA	b800
3	текст	80 x 25	16/8	CGA,EGA	b800
4	графика	320 x 200	4	CGA,EGA	b800
5	графика	320 x 200	4	CGA,EGA	b800
6	графика	640 x 200	2	CGA,EGA	b800
7	текст	80 x 25	3	MA,EGA	b000

Продолжение таблицы 2.2

0dh	графика	320 x 200	16	EGA	a000
0eh	графика	640 x 200	16	EGA	a000
0fh	графика	640 x 350	3	EGA	a000
10h	графика	640 x 350	4/16	EGA	a000
0bh,0ch		(резервируется для EGA BIOS)			

AH = 02H Установка позиции курсора. Установка на строку 25 делает курсор невидимым.

Вход: BH = видеостраница

DH,DL = строка, колонка (считая от 0)

AH = 03H читать позицию и размер курсора

Вход: BH = видеостраница

Выход: DH,DL = текущие строка, колонка курсора

CH,CL = текущие начальная, конечная строки курсора (см. функцию 01H)

AH = 05H выбрать активную страницу дисплея

Вход: AL = номер страницы (большинство программ использует нулевую страницу)

AH = 06H прокрутить окно вверх (или очистить). Прокрутка на 1 или более строк вверх.

Вход: CH,CL = строка, колонка верхнего левого угла окна (считая от 0)

DH,DL = строка, колонка нижнего правого угла окна (считая от 0)

AL = число пустых строк, вдвигаемых снизу (0 = очистить все окно)

BH = видео атрибут, используемый для пустых строк

AH = 07H прокрутить окно вниз (вдвинуть пустые строки в верхнюю часть окна)

Вход: (аналогично функции 06H)

AH = 08H читать символ/атрибут в текущей позиции курсора

Вход: BH = номер видеостраницы

Выход: AL = прочитанный символ

AH = прочитанный атрибут (только для текстовых режимов)

AH = 09H писать символ/атрибут в текущей позиции курсора

Вход: BH = номер видеостраницы

AL = записываемый символ

CX = счетчик (количество выводимых символов)

BL = атрибут (текст) или цвет (графика) (в графических режимах +80H означает XOR с символом на экране)

AH = 0aH писать символ в текущей позиции курсора

Вход: BH = номер видеостраницы

AL = записываемый символ

CX = счетчик (количество выводимых символов)

AH = 0bH выбрать цвет палитры/рамка (CGA-совместимые режимы)

Вход: BH = 0: (текст) выбрать цвет рамки

BL = цвет рамки (0-1fH; 10H – 1fH – интенсивные)

BH = 1: (графика) выбрать палитру

BL = 0: палитра green/red/brown

BL = 1: палитра cyan/magenta/white

AH = 0cH писать графическую точку (слишком медленно для большинства приложений!)

Вход: BH = номер видеостраницы

DX,CX = строка, колонка

AL = значение цвета (+80H означает XOR с точкой на экране)

AH = 0dH читать графическую точку (очень медленная функция!)

Вход: BH = номер видеостраницы

DX,CX = строка, колонка

Выход: AL = прочитанное значение цвета

AH = 0eH писать символ на активную страницу (в режиме телетайпа)

Вход: AL = записываемый символ (использует существующий атрибут)

BL = цвет переднего плана (для графических режимов)

AH = 0fH читать текущий видеорежим

Вход: нет

Выход: AL = текущий режим (см. функцию 00H)

AH = число текстовых колонок на экране

BH = текущий номер активной страницы дисплея

AH = 13H писать строку. Выдает строку в позиции курсора. Символы 0dH (Возврат каретки), 0aH (перевод строки), 08H (backspace) и 07H (гудок) трактуются как команды управления и не выводятся на экран.

Вход: ES:BP адрес строки вывода (специальный формат для AL=2 и AL=3)

CX = длина строки (подсчитываются только символы)

DH,DL = строка, колонка начала вывода

BH = номер страницы

AL = код подфункции:

0 = использовать атрибут в BL; не трогать курсор

1 = использовать атрибут в BL; курсор – в конец строки

2 = формат строки: char,attr, char,attr…; не трогать курсор

3 = формат строки: char,attr, char,attr…; передвинуть курсор

2.3 Прочие функции BIOS

int 13H: Дисковый ввод-вывод

Это программное прерывание предоставляет прямой доступ к дискетам и жесткому диску. Там, где это возможно, рекомендуется использовать программные прерывания int 25H и int 26H, чтобы предоставить драйверам устройств DOS выполнять всю низкоуровневую обработку. Разумеется, для таких операций, как форматирование диска или установка защиты от копирования, int 13H может оказаться единственной альтернативой. Ниже дано описание некоторых функций этого прерывания.

AH = 00H Сброс устройства. Выполняется полный сброс контроллера. Если значение DL равно 80H или 81H, выполняется сброс контроллера жесткого диска, иначе сбрасывается контроллер гибких дисков.

AH = 02H Читать секторы

Вход:DL = номер диска (0=диск A…; 80H = жесткий диск C, 81H = жесткий диск D и т.д.)

DH = номер головки чтения/записи

CH = номер дорожки (цилиндра)

CL = номер сектора

AL = число секторов (в сумме не больше чем один цилиндр)

ES:BX => адрес буфера вызывающей программы

0:0078 => таблица параметров гибкого диска

0:0104 => таблица параметров жесткого диска

Выход: Флаг переноса = 1 при ошибке и код ошибки в AH.

ES:BX буфер содержит данные, прочитанные с диска

Замечание: на сектор и цилиндр отводится соответственно 6 и 10 бит:

- биты 0 – 5 CX – номер сектора
- биты 6, 7 CX – старшие биты номера цилиндра
- биты 8 – 15 CX – младшие биты номера цилиндра

AH = 03H Писать секторы

Вход:(аналогично функции 02H)

ES:BX => данные, записываемые на диск.

Выход: Флаг переноса = 1 при ошибке и код ошибки в AH.

AH = 04H Проверить секторы. Проверяет контрольные суммы для указанных секторов.

Вход:(аналогично функции 02H. ES:BX также желательно указать)

Выход: Флаг переноса = 1 при ошибке и код ошибки в AH.

AH = 05H Форматировать дорожку. Данные на дорожке, если они есть, разрушаются.

Вход:DL,DH,CH = диск, головка, дорожка (см. функцию 02H)

ES:BX => дескрипторы секторов (необходим 512-байтовый буфер)

Выход: Флаг переноса = 1 при ошибке и код ошибки в AH.

AH = 17H установить тип дискеты (используется перед операцией форматирования)

Вход:DL = номер устройства диска (0 или 1)

AL = тип носителя диска:

- 0 = не используется
- 1 = 360K дискета в 360K устройстве
- 2 = 360K дискета в 1.2M устройстве
- 3 = 1.2M дискета в 1.2M устройстве

int 16H: Сервис клавиатуры

Это программное прерывание предоставляет интерфейс прикладного уровня с клавиатурой. Нажатия клавиш на самом деле обрабатываются асинхронно на заднем плане. Когда клавиша получена от клавиатуры, она обрабатывается прерыванием int 09H и помещается в циклическую очередь.

AH = 00H читать (ожидать) следующую нажатую клавишу

Выход: AL = ASCII символ (если AL=0, AH содержит Расширенный код ASCII)

AH = Сканкод или Расширенный код ASCII

AH = 01H Проверить готовность символа (и показать его, если он есть)

Выход: ZF = 1 если символ не готов.

ZF = 0 если символ готов.

AX = как для функции 00H (но символ здесь не удаляется из очереди).

AH = 02H Читать состояние shift-клавиш. Определить, какие shift-клавиши нажаты.

int 1cH: Пользовательское прерывание по таймеру

Это прерывание возникает по каждому тику аппаратных часов (каждые 55 миллисекунд; приблизительно 18.2 раз в секунду). Первоначально этот вектор указывает на IRET, но может быть изменен прикладной программой, чтобы адресовать фоновую программу пользователя, использующую прерывание по таймеру. Поскольку программа int 1cH выполняется во время низкоуровневого аппаратного прерывания, вы должны помнить, что система еще не сбросила контроллер прерываний и потому другие аппаратные прерывания, в том числе прерывание от клавиатуры, не будут происходить при работе INT 1cH (т.е. вы не сможете обработать ввод пользователя).

2.4 Функции DOS

Функции DOS – это функции, выполняемые при вызове прерывания int 21h с обозначением функции в регистре AH и подфункции (если это необходимо) в регистре AL.

Функция 00h: Завершить программу

Вход AH = 00h

CS = сегмент PSP завершаемого процесса

Описание: Передает управление на вектор завершения в PSP (выходит в родительский процесс). Идентична функции int 20h Terminate. Регистр CS должен указывать на PSP. Восстанавливает векторы прерываний DOS 22h-24h (Завершение, Ctrl-Break и Критическая ошибка), устанавливая значения, сохраненные в родительском PSP. Выполняет сброс файловых буферов. Должны быть закрыты файлы с измененной длиной).

Замечание: Проще и корректнее использовать функцию 4ch Exit.

Функция 01h: Ввод с клавиатуры

Вход AH = 01h

Выход AL = символ, полученный из устройства стандартного ввода

Описание: Считывает (ожидает) символ со стандартного устройства ввода. Отображает этот символ на стандартное устройство вывода (эхо). При распознавании Ctrl-Break выполняется int 23h.

Замечание: Ввод расширенных клавиш ASCII (F1-F12, PgUp, курсор и т. п.) требует двух обращений к этой функции. Первый вызов возвращает AL = 0. Второй вызов возвращает в AL расширенный код ASCII.

Функция 02h: Вывод на дисплей

Вход AH = 02h

DL = символ, выводимый на устройство стандартного вывода

Описание: Посылает символ из DL на устройство стандартного вывода. Обрабатывает символ Backspace (ASCII 8), перемещая курсор влево на одну позицию и оставляя его в новой позиции. При обнаружении Ctrl-Break выполняется int 23h.

Функция 06h: Консольный ввод/вывод

Вход AH = 06h

DL = символ (от 0 до 0feh), посылаемый на устройство стандартного вывода, DL = 0ffh запрос ввода с устройства стандартного ввода

Выход При запросе ввода (т.е. при DL=0ffh):

ZF сброшен (NZ), если символ готов

AL = Считанный символ, если ZF сброшен

Описание: При DL = 0ffh выполняет ввод с консоли «без ожидания», возвращая взведенный флаг нуля (ZF), если на консоли нет готового символа. Если символ готов, сбрасывает флаг ZF (NZ) и возвращает считанный символ в AL. Если DL не равен 0ffh, то DL направляется на стандартный вывод.

Замечание: Не проверяет Ctrl-Break. Для расширенного ASCII функцию следует вызывать дважды.

Функция 07h: Нефильтрованный консольный ввод без эха

Вход AH = 07h

Выход AL = символ, полученный из устройства стандартного ввода

Описание: Считывает (ожидает) символ из стандартного устройства ввода и возвращает этот символ в AL. Не фильтрует, т.е. не проверяет на Ctrl-Break, backspace и т. п.

Замечания: Вызывайте дважды для ввода расширенного символа ASCII. Используйте функцию 0bh для проверки статуса (если не хотите ожидать нажатия клавиши).

Функция 08h: Консольный ввод без эха

Вход AH = 08h

Выход AL = символ, полученный из устройства стандартного ввода

Описание: Считывает (ожидает) символ со стандартного устройства ввода и возвращает этот символ в AL. При обнаружении Ctrl-Break выполняется прерывание int 23h.

Замечание: Вызывайте дважды для ввода расширенного кода ASCII.

Функция 09h: Вывести строку на дисплей

Вход AH = 09h

DS:DX = адрес строки, заканчивающейся символом '$' (ASCII 24h)

Описание: Строка, исключая завершающий ее символ '$', посылается на устройство стандартного вывода. Символы Backspace обрабатываются как в функции 02h. Обычно, чтобы перейти на новую строку, включают в текст пару CR/LF (ASCII 13h и ASCII 0ah).

Функция 0ah: Ввод строки в буфер

Вход AH = 0ah

DS:DX = адрес входного буфера (смотри ниже)

Выход Буфер содержит ввод, заканчивающийся символом CR (ASCII 0dh)

Описание: При обращении буфер по адресу DS:DX должен содержать значение максимально допустимой длины ввода. На выходе функции в следующем байте содержится действительная длина ввода, затем введенный текст, завершающийся символом возврата каретки (0dh). Символы считываются с устройства стандартного ввода вплоть до CR (ASCII 0dh) или до достижения длины MAX-1. Если достигнут MAX-1, включается консольный звонок для каждого очередного символа, пока не будет введен возврат каретки CR (нажатие Enter). Второй байт буфера заполняется действительной длиной введенной строки, не считая завершающего CR. Последний символ в буфере – всегда CR (который не засчитан в байте длины). Символы в буфере (включая LEN) в момент вызова используются как «шаблон». В процессе ввода действительны обычные клавиши редактирования: Esc выдает «\» и начинает с начала, F3 выдает буфер до конца шаблона, F5 выдает «@» и сохраняет текущую строку как шаблон, и т. д. Большинство расширенных кодов ASCII игнорируются. При распознавании Ctrl-Break выполняется прерывание int 23h (буфер остается неизменным).

Функция 0bh: Проверить статус ввода

Вход AH = 0bh

Выход AL = 0ffh, если символ доступен со стандартного ввода

AL = 0, если нет доступного символа

Описание: Проверяет состояние стандартного ввода. При распознавании Ctrl-Break выполняется int 23h.

Замечания: Используйте перед функциями 01h 07h и 08h, чтобы избежать ожидания нажатия клавиши. Эта функция дает простой неразрушающий способ проверки Ctrl-Break в процессе длинных вычислений или другой обработки, обычно не требующей ввода. Это позволяет вам снимать счет по нажатию Ctrl-Break.

Функция 0ch: Ввод с очисткой

Ввод AH = 0ch

AL = номер функции ввода DOS (01h, 06h, 07h, 08h или 0ah)

Описание: Очищает буфер опережающего ввода стандартного ввода, а затем вызывает функцию ввода, указанную в AL. Это заставляет систему ожидать ввода очередного символа. В AL допустимы следующие значения:

- 01h Ввод с клавиатуры
- 06h Ввод с консоли
- 07h Нефильтрованный ввод без эха
- 08h Ввод без эха
- 0ah Буферизованный ввод

Функция 0dh: Сброс диска

Вход AH = 0dh

Описание: Сбрасывает (записывает на диск) все буферы файлов. Если размер файла изменился, такой файл должен быть предварительно закрыт (при помощи функций 10h или 3eh).

Функция 0eh: Установить текущий диск DOS

Вход AH = 0eh

DL = номер диска (0=A, 1=B и т.д.), который становится текущим

Выход AL = общее число дисководов в системе

Описание: Диск, указанный в DL, становится текущим (диском по умолчанию) в DOS. Для проверки используйте функцию 19h «Дать текущий диск». В регистре AL возвращается число дисководов всех типов, включая жесткие и логические диски.

Функция 19h: Дать текущий диск DOS

Вход AH = 19h

Выход AL = Номер текущего диска (0 = A, 1 = B, и т.д.)

Описание: Возвращает номер дисковода текущего диска DOS.

Функция 1ah: Установить адрес DTA

Вход AH = 1ah

DS:DX = адрес области передачи данных (DTA)

Описание: Устанавливает адрес DTA. Все FCB-ориентированные операции работают с DTA. DOS не позволяет операциям вода/вывода пересекать границу сегмента. Функции поиска: 11h 12h 4eh и 4fh помещают данные в DTA. DTA глобальна, поэтому будьте осторожны, назначая ее в рекурсивной или реентерабельной процедуре. При запуске программы ее DTA устанавливается по смещению 80h относительно PSP.

Функция 1bh: Дать информацию FAT (текущий диск)

Вход AH = 1bh

Выход DS:BX = адрес байта FAT ID (отражающего тип диска)

DX = всего кластеров на диске

AL = секторов в кластере

CX = байтов в секторе

Описание: Возвращает информацию о размере и типе текущего диска. Размер диска в байтах = (DX * AL * CX). Для определения свободного места на диске используйте функции 36h Disk Free или 32h Disk Info.

Внимание: Эта функция изменяет содержимое регистра DS.

Функция 1ch: Дать информацию FAT (любой диск)

Вход AH = 1ch

DL = номер диска (0 = текущий, 1 = A, и т.д.)

Выход DS:BX = адрес байта FAT ID (отражающего тип диска)

DX = всего кластеров

AL = секторов в кластере

CX = байтов в секторе

Описание: Аналогична функции 1bh Get FAT Cur, с той разницей, что регистр DL указывает диск, для которого вы хотите получить информацию.

Функция 25h: Установить вектор прерывания

Вход AH = 25h

AL = номер прерывания

DS:DX = вектор прерывания: адрес программы обработки прерывания

Описание: Устанавливает значение элемента таблицы векторов прерываний для прерывания с номером AL равным DS:DX. Это равносильно записи 4-байтового адреса в 0000:(AL*4), но, в отличие от прямой записи, DOS здесь знает, что вы делаете, и гарантирует, что в момент записи прерывания будут заблокированы.

Внимание: Не забудьте восстановить DS (если необходимо) после этого вызова.

Функция 26h: Построить PSP

Вход AH = 26h

DX = адрес сегмента (параграфа) для нового PSP

CS = сегмент PSP, используемого как шаблон для нового PSP

Описание: Устанавливает PSP для порождаемого процесса по адресу DX:0000. Текущий PSP (100h байт, начиная с CS:0), копируется в DX:0. Соответственно корректируется поле MemTop. Векторы Terminate, Ctrl-Break и Critical Error копируются в PSP из векторов прерываний int 22h, int 23h и int 24h.

Функция 2ah: Дать системную дату

Вход AH = 2ah

Выход AL = день недели (0 = воскресенье, 1 = Понедельник,…)

CX = год (от 1980 до 2099)

DH = месяц (от 1 до 12)

DL = день (от 1 до 31)

Описание: Возвращает текущую системную дату.

Функция 2bh: Установить системную дату

Вход AH = 2bh

CX = год (от 1980 до 2099)

DH = месяц (от 1 до 12)

DL = день (от 1 до 31)

Выход AL = 0, если дата корректна

AL = 0ffh если дата некорректна

Описание: Устанавливает системную дату DOS.

Функция 2ch: Дать системное время

Вход AH = 2ch

Выход CH = часы (от 0 до 23)

CL = минуты (от 0 до 59)

DH = секунды (от 0 до 59)

DL = сотые доли секунды (от 0 до 99)

Описание: Возвращает текущее системное время.

Замечание: Поскольку системные часы имеют частоту 18.2 тиков в секунду (интервал 55мс), DL имеет точность 0.04 сек.

Функция 2dh: Установить системное время

Вход AH = 2dh

CH = часы (от 0 до 23)

CL = минуты (от 0 до 59)

DH = секунды (от 0 до 59)

DL = сотые доли секунды (от 0 до 99)

Выход AL = 0, если время корректно

AL = 0ffh если время некорректно

Описание: Устанавливает системное время.

Функция 2eh: Установить/сбросить переключатель верификации

Вход AH = 2eh

AL = 0 отключить верификацию

AL = 1 включить верификацию

Описание: Устанавливает, должна ли DOS проверять каждый сектор, записываемый на диск. Это замедляет операции записи на диск, но дает некоторую гарантию при записи. Функция 56h Get Verify возвращает текущий статус верификации DOS.

Функция 2fh: Дать адрес текущей DTA

Вход AH = 2fh

Выход ES:BX = адрес начала текущей DTA

Описание: Возвращает адрес начала области передачи данных (DTA). Поскольку DTA глобальна для всех процессов, в рекурсивной процедуре (например, при проходе по дереву оглавления) может потребоваться сохранить адрес DTA, а впоследствии восстановить его посредством функции 1ah «Установить DTA».

Замечание: Эта функция изменяет сегментный регистр ES.

Функция 30h: Дать номер версии DOS

Вход AH = 30h

Выход AL = часть номера версии до точки

AH = часть номера версии после точки

BX,CX = 0000h DOS 3.0+

Описание: Возвращает в AX значение текущего номера версии DOS. Например, для DOS 3.2, в AL возвращается 3, в Ah – 2.

Замечание: Если в AL возвращается 0, можно предполагать, что работает DOS более ранней версии, чем DOS 2.0.

Функция 31h: Завершиться и остаться резидентным (KEEP)

Вход AH = 31h

AL = код возврата

DX = объем оставляемой резидентной части в параграфах

Описание: Выходит в родительский процесс, сохраняя код возврата в AL. Код возврата можно получить через функцию 4dh Wait. DOS устанавливает начальное распределение памяти, как специфицировано в DX, и возвращает управление родительскому процессу, оставляя указанную часть резидентной (число байтов = DX*16). Эта функция перекрывает функцию int 27h, которая не возвращает код возврата и неспособна установить резидентную программу больше сегмента.

Функция 33h: Установить/опросить статус Ctrl-Break

Вход AH = 33h

AL = 0 чтобы опросить текущий статус контроля Ctrl-Break

AL = 1 чтобы установить статус контроля Ctrl-Break

DL = требуемый статус (0=OFF, 1=ON) (только при AL=1)

Выход DL = текущий статус (0 = OFF, 1 = ON)

Описание: Если AL=0, в DL возвращается текущий статус контроля Ctrl-Break. Если AL=1, в DL возвращается новый текущий статус. Когда статус ON, DOS проверяет на Ctrl-Break с консоли для большинства функций (исключая 06h и 07h). При обнаружении, выполняется int 23h (если оно не перехватывается, то это снимает процесс). Когда статус OFF, DOS проверяет на Ctrl-Break лишь при операциях стандартного в/в, стандартной печати и стандартных операциях AUX.

Функция 35h: Дать вектор прерывания

Вход AH = 35h

AL = номер прерывания (от 00h до 0ffh)

Выход ES:BX = адрес обработчика прерывания

Описание: Возвращает значение вектора прерывания для int (AL); то есть, загружает в BX 0000:[AL*4], а в ES – 0000:[(AL*4)+2].

Внимание: Эта функция изменяет сегментный регистр ES.

Функция 36h: Дать свободную память диска

Вход AH = 36h

DL = номер диска (0=текущий, 1=A, и т.д.)

Выход AX = 0ffffh, если AL содержал неверный номер диска

AX = число секторов на кластер, если нет ошибок

BX = доступных кластеров

CX = байт на сектор

DX = всего кластеров на диске

Описание: Возвращает данные для подсчета общей и доступной дисковой памяти. Если в AX возвращено 0ffffh, значит, вы задали неверный диск. Иначе, свободная память в байтах = (AX * BX * CX) всего памяти в байтах = (AX * CX * DX).

Функция 39h: Создать новый каталог (MKDIR)

Вход AH = 39h

DS:DX = адрес строки ASCIIZ с именем оглавления

Выход AX = код ошибки если CF установлен

Описание: DS:DX указывает на строку ASCIIZ в формате: «d:\путь\имя_каталога»,0. Если диск и/или путь опущены, то берется каталог, принятый по умолчанию. Подкаталог создается и связывается с существующим деревом. Если при возврате, установлен флаг CF, то AX содержит код ошибки, и каталог не создается.

Функция 3ah: Удалить каталог (RMDIR)

Вход AH = 3ah

DS:DX = адрес строки ASCIIZ с именем оглавления

Выход AX = код ошибки, если установлен CF

Описание: DS:DX указывает на строку ASCIIZ в формате: «d:\путь\имя_каталога»,0. Если диск и/или путь опущены, то берется каталог, принятый по умолчанию. Подкаталог удаляется из указанного каталога. Если при возврате, установлен флаг CF, то AX содержит код ошибки, и каталог не удаляется.

Замечание: Каталог не должен содержать файлов и подкаталогов, а также и не должен быть связан с возможными ограничениями DOS (например, каталог не должен быть задействован в активных командах JOIN или SUBST).

Функция 3bh: Установить текущий каталог DOS (CHDIR)

Вход AH = 3bh

DS:DX = адрес строки ASCIIZ с именем каталога

Выход AX = код ошибки, если установлен CF

Описание: DS:DX указывает на строку ASCIIZ в формате: «d:\путь\имя_каталога»,0. Если диск и/или путь опущены, то берется каталог, принятый по умолчанию. Указанный подкаталог для указанного диска становится текущим каталогом DOS для этого (или текущего) диска. Если при возврате установлен флаг CF, то AX содержит код ошибки, и текущий каталог для выбранного диска не изменяется.

Функция 3ch: Создать файл через дескриптор

Вход AH = 3ch

DS:DX = адрес строки ASCIIZ с именем файла

CX = атрибут файла

Выход AX = код ошибки, если CF установлен

AX = дескриптор файла, если ошибки нет

Описание: DS:DX указывает на строку ASCIIZ в формате: «d:\путь\имяфайла»,0. Если диск и/или путь опущены, они принимаются по умолчанию. файл создается в указанном (или текущем) каталоге файл открывается в режиме доступа чтение/запись вы должны сохранить дескриптор (handle) для последующих операций, если файл уже существует:

при открытии файл усекается до нулевой длины

если атрибут файла – только чтение, открытие отвергается (атрибут можно изменить функцией 43h Изменить Атрибут)

CONFIG.SYS определяет число доступных дескрипторов в системе

Используйте функцию 5bh Создать Новый Файл, если вы не хотите испортить существующий файл.

Функция 3dh: Открыть дескриптор файла

Вход AH = 3dh

DS:DX = адрес строки ASCIIZ с именем файла

AL = Режим открытия

Выход AX = код ошибки, если CF установлен

AX = дескриптор файла, если нет ошибки

Описание: DS:DX указывает на строку ASCIIZ в формате: d:\путь\имя_файла»,0. Если диск и/или путь опущены, они принимаются по умолчанию. При этом:

файл должен существовать. См. функцию 3ch (Создать Файл). Файл открывается в выбранном Режиме Доступа/Режиме Открытия

AL = 0 открыть для чтения

AL = 1 открыть для записи

AL = 2 открыть для чтения и записи

указатель чтения/записи устанавливается в 0. См. 42h (LSEEK)

вы должны сохранить дескриптор (handle) для последующих операций

если запрашивается открытие в одном из режимов разделения, должно быть активно разделение файлов (команда DOS SHARE).

CONFIG.SYS определяет число доступных дескрипторов файлов.

Функция 3eh: Закрыть дескриптор файла

Вход AH = 3eh

BX = дескриптор файла

Выход AX = код ошибки, если CF установлен

Описание: BX содержит дескриптор файла (handle), возвращенный при открытии. Файл, представленный этим дескриптором, закрывается, его буферы сбрасываются, и запись в каталоге обновляется корректными размером, временем и датой. Из-за нехватки дескрипторов файлов (максимум 20, по умолчанию 8), вам может понадобиться закрыть часть стандартных дескрипторов, например, дескриптор 3 (стандартный AUX).

Функция 3fh: Читать файл через дескриптор

Вход AH = 3fh

BX = дескриптор файла

DS:DX = адрес буфера для чтения данных

CX = число считываемых байтов

Выход AX = код ошибки, если CF установлен

AX = число действительно прочитанных байтов

Описание: CX байтов данных считываются из файла или устройства с дескриптором, указанным в BX. Данные читаются с текущей позиции указателя чтения/записи файла и помещаются в буфер вызывающей программы, адресуемый через DS:DX. Используйте функцию 42h LSEEK, чтобы установить указатель файла, если необходимо (OPEN сбрасывает указатель в 0). Модифицирует указатель чтения/записи файла, подготавливая его к последующим операциям чтения или записи. Вы должны всегда сравнивать возвращаемое значение AX (число прочитанных байтов) с CX (запрошенное число байтов):

если AX = CX, (и CF сброшен) – чтение было корректным без ошибок

если AX = 0, достигнут конец файла (EOF)

если AX < CX (но ненулевой):

при чтении с устройства – входная строка имеет длину AX байт

при чтении из файла – в процессе чтения достигнут EOF

Замечания: Эта функция превосходит сложные и неудобные FCB-функции. Она эффективно сочетает произвольный и последовательный доступ, позволяя пользователю выполнять свое собственное блокирование. Удобно использовать эту функцию для чтения стандартных дескрипторов, таких как дескрипторы стандартного ввода/вывода, взамен многочисленных буферизующих и посимвольных FCB-функций ввода. Когда вы читаете с устройства, AX возвращает длину считанной строки с учетом завершающего возврата каретки CR (ASCII 0dh).

Функция 40h: Писать в файл через дескриптор

Вход AH = 40h

BX = дескриптор файла

DS:DX = адрес буфера, содержащего записываемые данные

CX = число записываемых байтов

Выход AX = код ошибки, если CF установлен

AL = число реально считанных байтов

Описание: CX байт данных записывается в файл или на устройство с дескриптором, заданным в BX. Данные берутся из буфера, адресуемого через DS:DX. Данные записываются, начиная с текущей позиции указателя чтения/записи файла. Используйте функцию 42h LSEEK, чтобы установить указатель файла, если необходимо (OPEN сбрасывает указатель в 0). Обновляет указатель чтения/записи файла, чтобы подготовиться к последующим операциям последовательного чтения или записи. Вы должны всегда

сравнивать возвращаемое значение AX (число записанных байтов) с CX (запрошенное число байтов для записи). При этом:

если AX = CX, запись была успешной, если AX < CX, встретилась ошибка (вероятно, переполнение).

Замечание: Эта функция превосходит сложные и неудобные FCB-функции. Она эффективно сочетает произвольный и последовательный доступ, позволяя пользователю осуществлять собственное блокирование. Удобно использовать эту функцию для вывода на стандартные устройства, такие как стандартный вывод, взамен использования различных функций вывода текста.

Функция 41h: Удалить файл

Вход AH = 41h

DS:DX = адрес строки ASCIIZ с именем файла

Выход AX = код ошибки, если CF установлен

Описание: DS:DX указывает на строку ASCIIZ в формате: d:\путь\имяфайла»,0. Если диск и/или путь опущены, они принимаются по умолчанию. Имя файла не может содержать символы замены ('?' и '*'). Файл удаляется из заданного каталога заданного диска. Если файл имеет атрибут только чтение, то перед удалением необходимо изменить этот атрибут при помощи функции 43h CHMOD.

Функция 42h: Установить указатель файла (LSEEK)

Вход AH = 42h

BX = дескриптор файла

CX:DX = величина сдвига указателя: (CX*65536)+DX

AL = 0 переместить к началу файла +CX:DX

AL = 1 переместить к текущей позиции +CX:DX

AL = 2 переместить к концу файла +CX:DX

Выход AX = код ошибки, если CF установлен

DX:AX = новая позиция указателя файла (если нет ошибки)

Описание: Перемещает логический указатель чтения/записи к нужному адресу. Очередная операция чтения или записи начнется с этого адреса.

Замечание: Вызов с AL=2, CX=0, DX=0 возвращает длину файла в DX:AX. DX здесь – значащее слово: действительная длина (DX * 65536) + AX.

Функция 43h: Установить/опросить атрибут файла (CHMOD)

Вход AH = 43h

DS:DX = адрес строки ASCIIZ с именем файла

AL = Код подфункции:

AL = 0 – извлечь текущий атрибут файла

AL = 1-= установить атрибут файла

CX = новый атрибут файла (для подфункции 01h)

Выход AX = код ошибки, если CF установлен

CX = текущий атрибут файла (для подфункции 00h)

Описание: DS:DX указывает на строку ASCIIZ в формате: «d:\путь\имяфайла»,0. Если диск и/или путь опущены, они принимаются по умолчанию. Атрибут файла извлекается или устанавливается, согласно коду в AL.

Замечание: Чтобы спрятать каталог, используйте CX=02h (а не 12h, как вы, возможно, ожидали).

Функция 44h: Управление устройством ввода/вывода (IOCTL)

Вход AH = 44h

AL = Код подфункции:

00h = дать информацию устройства

01h = уст. информацию устройства

02h = читать с символьного устройства

03h = писать на символьное устройство

04h = читать с блочного устройства

05h = писать на блочное устройство

06h = дать статус ввода

07h = дать статус вывода

08h = запрос съемного носителя

09h = запрос локального/удаленного устройства

0ah = запрос локального/удаленного дескриптора

0bh = счет повторов разделения

0ch (зарезервировано)

0dh = общий IOCTL DOS 3.2+

0eh = дать логическое устройство 3.2+

0fh = уст логическое устройство 3.2+

Выход AX = код ошибки, если CF установлен или иное значение (в зависимости от подфункции)

Описание: IOCTL предоставляет собой метод взаимодействия с устройствами и получения информации о файлах. Входные параметры и выходные значения варьируются в зависимости от кода подфункции в регистре AL.

Подф. 00h: Запросить флаги информации об устройстве

Вход BX = дескриптор файла (устройство или дисковый файл)

Выход DX= IOCTL Информация об устройстве

Подф. 01h: Установить флаги информации об устройстве

Вход BX= дескриптор файла (устройство или дисковый файл)

DX= IOCTL Информация об устройстве (DH должен быть нулевым)

Выход DX= IOCTL Информация об устройстве

Подф. 02-03: Читать (AL=02h) или Писать (AL=03h) строку IOCTL на СИМВОЛЬНОЕ устройство

Вход DS:DX=> адрес буфера (чтение) или данных (запись)

CX = число передаваемых байтов

BX = дескриптор файла (только устройство, но не файл!)

Выход AX= код ошибки, если CF установлен

Подф. 04-05: Читать (AL=04h) или Писать (AL=05h) строку IOCTL на БЛОЧНОЕ устройство

Вход DS:DX=> адрес буфера (чтение) или данных (запись)

CX= число передаваемых байтов

BL= ID диска (0 = текущий, 1 = A, и т. д.)

Выход AX = код ошибки, если CF установлен AX = действительное число переданных байтов (если CF=NC=0)

Подф. 06-07: Дать статус ввода (AL=06h) или статус вывода (AL=07h)

Вход BX = дескриптор файла (устройство или дисковый файл)

Выход AL = 0ffh – не конец файла;

AL = 0 – EOF (для дисковых дескрипторов)

AL = 0ffh – готово;

AL = 0 – не готово (для устройств)

Подф. 08h: Использует ли блочное устройство съемный носитель?

Вход BL = ID диска (0 = текущий, 1=A, и т. д.)

Выход AX = 00h – съемный носитель (т. е. гибкий диск)

AX = 01h – несъемный (жесткий диск или RAM-диск)

AX = 0fh код ошибки, если BL содержит неверный диск

Подф. 09h: Является ли устройство съемным в сети?

Вход BL= ID диска (0 = текущий, 1=A, и т.д.)

Выход DX= атрибут устройства для диска. Если бит 12=1 (т.е., DX & 1000h = 1000h), то устройство съемное.

Подф. 0ah: Принадлежит ли дескриптор файла локальному или удаленному устройству в сети?

Вход BX = дескриптор файла (только устройство, но не файл)

Выход DX = атрибут устройства для диска. Если бит 15=1 (т.е., DX & 8000h = 8000h), то устройство удаленное.

Подф. 0bh: Контроль повтора/задержки при разделении и блокировке файлов.

Вход DX= число попыток перед вызовом int 24h «Критическая Ошибка»

CX= счетчик цикла между попытками

Выход AX= код ошибки, если CF установлен

Замечание: По умолчанию – 3 попытки и счетчик цикла 1.

Подф. 0dh: Общий вызов IOCTL отрабатывает разнообразные функции управления. Начиная с DOS 3.2, можно создавать драйверы устройств, работающие на уровне дорожек (форматирование, чтение/запись). DOS 3.2+ Код действия в регистре CL определяет «подподфункцию»:

Вход CL= код действия

40h = Установить параметры устройства

60h = Дать параметры устройства

41h = Писать дорожку логического устройства

61h = Читать дорожку логического устройства

42h = Форматировать дорожку с верификацией

62h = Верифицировать дорожку логического устройства

DS:DX=> адрес пакета данных IOCTL

По поводу структуры пакетов данных IOCTL см. Общий IOCTL 40h/60h, Общий IOCTL 41h/61h, Общий IOCTL 42h/62h.

Выход AX= код ошибки, если CF установлен

DS:DX=> пакет данных может содержать информацию возврата.

Подф. 0eh: Выяснить, назначил ли драйвер устройства несколько логических устройств одному физическому устройству.

Вход BL = ID диска (0 = текущий, 1=A, и т.д.)

Выход AX = код ошибки, если CF установлен

AL = 0 если ровно одна буква диска назначена устройству BL

AL = (1 = A, 2 = B и т.д.) если назначено несколько логических устройств, AL содержит ID текущего назначенного диска

Подф. 0fh: Сообщить драйверу блочного устройства ID устройства для обработки. Когда с физическим устройством ассоциируется несколько логических, DOS выдает сообщение «Insert diskette for drive X:…». Эта функция позволяет вам сообщать DOS, что диск с указанным ID уже установлен, тем самым обходя сообщение.

Вход BL= ID диска (0 = текущий, 1=A, и т.д.)

Выход AX= код ошибки, если CF установлен

AL = 0 если ровно один ID назначен устройству BL

AL = (1=A, 2=B и т.д.) ID выбранного устройства, которое будет использоваться в последующих операциях вода/вывода.

Замечание: Эта функция должна вызываться перед любой операцией вода/вывода на логическом устройстве. Иначе DOS может выдать сообщение.

Функция 45h: Дублировать дескриптор файла (DUP)

Вход AH = 45h

BX = существующий дескриптор файла

Выход AX = новый дескриптор файла, дублирующий оригинал

AX = код ошибки, если CF установлен

Описание: Функция создает дополнительный дескриптор файла, ссылающийся на тот же поток вода/вывода, что и существующий дескриптор. Любое продвижение указателя чтения/записи для одного дескриптора действует на его дубликат, включая любые операции чтения, записи или перемещения указателя посредством функции 42h LSEEK. Новый дескриптор наследует ограничения Режима Открытия оригинала. Эта функция используется с одной главной целью: вы можете закрыть дескриптор, заставляя DOS записать файловые буферы. Такой способ DUP/CLOSE быстрее, чем закрытие и повторное открытие файла.

Функция 46h: Переназначить дескриптор (FORCDUP)

Вход AH = 46h

BX = целевой дескриптор файла (уже должен существовать)

CX = исходный дескриптор файла (уже должен уже существовать)

Выход AX = код ошибки, если CF установлен

Описание: Заставляет дескриптор файла (handle) ссылаться на другой файл или устройство. Дескриптор в CX (источник) закрывается, если он открыт, а затем становится дубликатом дескриптора в BX (назначения). Иными словами, дескрипторы в CX и BX будут ссылаться на один и тот же физический файл или устройство. Используется для переназначения стандартного устройства ввода/вывода. Пример: Откроем файл «C:\STDOUT.TXT» через функцию 3dh Open File и получим дескриптор (например, 05). Установим BX=05, CX=01 и вызовем эту функцию. (Замечание: дескриптор 01 – это предопределенный дескриптор «стандартного выходного устройства»). Теперь можно вызвать функцию 3eh Close File и закрыть handle 05. Можно обращаться к файлу STDOUT.TXT через дескриптор 01. Стало быть, дисковый файл «C:\STDOUT.TXT» будет отныне получать весь вывод, создаваемый всеми

процессами (текущим и порожденными), через любую функцию символьного ввода/вывода DOS, так же, как и любой вывод в дескриптор файла 01 через функцию DOS 40h. Когда вы выходите в COMMAND.COM, предопределенные дескрипторы устанавливаются на обычные устройства (например, дескриптор 01 устанавливается на «CON»).

Функция 47h: Дать текущий каталог DOS

Вход AH = 47h

DS:SI = адрес буфера для указания пути (64 байта)

DL = номер диска (0 = текущий, 1 = A, и т. д.)

Выход AX = код ошибки, если CF установлен

Описание: В пользовательский буфер по адресу DS:SI помещается в форме ASCIIZ путь текущего каталога для диска, указанного в DL. Путь возвращается в формате: «путь\каталог»,0. Впереди не подставляется буквенное обозначение диска, а сзади не подставляется символ «\». Например, если текущим является корневой каталог, эта функция вернет вам пустую строку (DS:[SI] = 0).

Функция 48h: Распределить память (дать размер памяти)

Вход AX = 48h

BX = размер запрашиваемой памяти в параграфах

Выход AX = сегментный адрес распределенного блока (если нет ошибок). При ошибке: AX = код ошибки, если установлен CF

BX = размер доступной памяти в параграфах (если памяти не хватает)

Описание: Распределяет блок памяти размером в BX параграфов, возвращая сегментный адрес этого блока в AX (блок начинается с AX:0000). Если распределение неудачно, взводится флаг переноса CF, а в AX возвращается код ошибки; BX при этом содержит максимальный размер доступной для распределения памяти (в параграфах). Чтобы определить наибольший доступный фрагмент, обычно перед вызовом устанавливают BX=0ffffh. Функция завершится с ошибкой, возвратив размер максимального доступного блока памяти в BX.

Замечание: Когда процесс получает управление через функцию 4bh EXEC, вся доступная память уже распределена ему.

Функция 49h: Освободить распределенный блок памяти

Вход AH = 49h

ES = сегментный адрес (параграф) освобождаемого блока памяти

Выход AX = код ошибки, если CF установлен

Описание: Освобождает блок памяти, начинающийся с адреса ES:0000. Этот блок становится доступным для других запросов системы. Вообще говоря, вы должны освобождать лишь те блоки памяти, которые вы получили через функцию 48h Распределить Память. Родитель отвечает за освобождение памяти порожденных процессов. Тем не менее, ничто не препятствует вам освобождать память чужих процессов.

Функция 4ah: Сжать или расширить блок памяти

Вход AH = 4ah

ES = сегмент распределенного блока памяти

BX = желаемый размер блока в 16-байтовых параграфах

Выход AX = код ошибки, если CF установлен

BX = наибольший доступный блок памяти (если расширение неудачно)

Описание: Изменяет размер существующего блока памяти. Когда программа получает управление, функция 4bh EXEC уже распределила блок памяти, начинающийся с PSP и содержащий всю доступную память. Чтобы освободить память для запуска порождаемых процессов, необходимо сначала сжать блок памяти, начинающийся с PSP.

Замечание: Функция 31h (KEEP) и int 27h (TSR) сжимают блок по адресу PSP.

Функция 4bh: Выполнить или загрузить программу (EXEC)

Вход AH = 4bh

DS:DX = адрес строки ASCIIZ с именем файла, содержащего программу

ES:BX = адрес EPB (EXEC Parameter Block – блока параметров EXEC)

AL = 0 Загрузить и выполнить

AL = 3 Загрузить программный оверлей

Выход AX = код ошибки, если CF установлен

Описание: Функция предоставляет средства одной программе (процессу-родителю) вызвать другую программу (порожденный процесс), которая по завершению возвратит управление процессу родителю. Адрес DS:DX указывает на строку ASCIIZ в форме: «d:\путь\имя_файла»,0. Если диск или путь опущены, они подразумеваются по умолчанию. ES:BX указывает на блок памяти, подготовленный как EPB, формат которого зависит от запрошенной подфункции в AL. AL=0 EXEC: Так как родительская программа первоначально получает всю доступную память в свое распоряжение, вы должны освободить часть памяти через функцию 4ah до вызова EXEC (AL=0). Обычная последовательность:

Вызовите функцию 4ah с ES=сегменту PSP и BX=минимальному объему памяти, требуемой вашей программе (в параграфах).

Подготовьте строку ASCIIZ со спецификацией вызываемого программного файла и установите DS:DX на первый символ этой строки.

Подготовьте Блок Параметров EXEC со всеми необходимыми полями.

Сохраните текущие значения SS, SP, DS, ES и DTA в переменных, адресуемых через регистр CS (CS – это единственная точка для ссылок после того, как EXEC вернет управление от порожденного процесса).

Выдайте вызов EXEC с AL=0.

Восстановите локальные значения SS и SP.

Проверьте флаг CF, чтобы узнать, не было ли ошибки при EXEC.

Восстановите DS, ES и локальную DTA, если это необходимо.

Проверьте код выхода через функцию 4dh WAIT (если надо).

Все открытые файлы дублируются, так что порожденный процесс может обрабатывать данные как через дескрипторы файлов, так и через стандартный ввод/вывод. Режимы доступа дескрипторов дублируются, но любые активные блокировки файлов не будут относиться к порожденному процессу (см. функцию 5ch). После возврата из порожденного процесса, в векторах int 22h Terminate, int 23h CtrlBreak и int 24h Critical Error восстанавливаются их исходные значения. AL=3 LOAD: Эта подфункция используется для загрузки «оверлея». DS:DX указывает на ASCIIZ имя файла, а ES:BX указывает на «LOAD»-версию Блока Параметров EXEC. Главная особенность этой подфункции заключается в том, что она считывает Заголовок EXE и выполняет необходимые перемещения сегментов, как это требуется для программ типа.EXE.

Замечания: Эта функция использует программу-загрузчик из COMMAND.COM, который транзитен в DOS 2.x (и, возможно, уже перекрыт программой). В этом случае возникнет ошибка, если DOS не найдет файл COMMAND.COM. Перед вызовом этой функции следует обеспечить корректную строку COMSPEC= в окружении. Вместо разбора собственных FCB (как требуется для EPB), вы можете загрузить и выполнить вторичную копию файла COMMAND.COM, используя опцию /C. Например, чтобы выполнить программу FORMAT.COM, установите DS:DX на адрес строки ASCIIZ: «\command.com»,0 и установите EPB+2 на сегмент и смещение следующей строки команд: 0eh,»/c format a:/s/4»,0dh. Такой вторичный интерпретатор команд использует очень мало памяти (около 4K). Вы можете

поискать в Окружении DOS строку COMSPEC=, чтобы установить точное местоположение файла COMMAND.COM.

Функция 4ch: Завершить программу (EXIT)

Вход AH = 4ch

AL = код возврата

Описание: Возвращает управление от порожденного процесса его родителю, устанавливая код возврата, который можно опросить функцией 4dh WAIT. Управление передается по адресу завершения в PSP завершаемой программы. В векторах Ctrl-Break и Critical Error восстанавливаются старые значения, сохраненные в родительском PSP.

Замечание: Значение ERRORLEVEL (используемое в пакетных файлах DOS) можно использовать для проверки кода возврата самой последней программы.

Функция 4dh: Дать код возврата программы (WAIT)

Вход AH = 4dh

Выход AL = код возврата последнего завершившегося процесса

AH = 0 – нормальное завершение

AH = 1 – завершение через Ctrl-Break int 23h

AH = 2 – завершение по критической ошибке устройства int 24h

AH = 3 – завершение через функцию 31h KEEP

Описание: Возвращает код возврата последнего из завершившихся процессов. Эта функция возвращает правильную информацию только однажды для каждого завершившегося процесса.

Функция 4eh: Найти 1-й совпадающий файл

Вход AH = 4fh

DS:DX = адрес строки ASCIIZ с именем файла (допускаются ? и *)

CX = атрибут файла для сравнения

Выход AX = код ошибки, если CF установлен, DTA заполнена данными (если не было ошибки)

Описание: DS:DX указывает на строку ASCIIZ в форме: «d:\путь\имя_файла»,0. Если диск и/или путь опущены, они подразумеваются по умолчанию. Символы замены * и ? допускаются в имени файла и расширении. DOS находит имя первого файла в каталоге, которое совпадает с заданным именем и атрибутом, и помещает найденное имя и другую информацию в DTA, как показано в таблице 2.3.

Таблица 2.3 – Содержимое DTA

DTA			
Смещение	Длина	Содержимое	Примечания
+0	15h	Резерв	Используется в последующих вызовах 4fh Find Next
+15h	1	Атрибут	Атрибут искомого файла
+16h	2	Время	Время создания/модификации в формате filetime
+18h	2	Дата	Дата создания/модификации в формате filedate
+1ah	4	Младший, старший	Размер файла в байтах в формате DWORD
+1eh	0dh	Имя файла	13-байтовое ASCIIZ имя: “filename.ext”,0
+2ch	Требуемый размер буфера		

Типичная последовательность, используемая для поиска всех подходящих файлов: используйте вызов 1ah, чтобы установить DTA на локальный буфер (или используйте текущую DTA в PSP по смещению 80h); установите CX=атрибут, DS:DX => ASCIIZ диск, путь, обобщенное имя; вызовите функцию 4eh (Найти 1-й); если флаг CF указывает ошибку, вы закончили (нет совпадений); установите DS:DX => DTA (или на данные, которые вы скопировали из DTA после вызова функции 4eh); повторять поиск последующих файлов с использованием функции 4fh (Найти следующий), пока установленный флаг переноса (CF) не покажет, что совпадений больше нет.

Функция 4fh: Найти следующий совпадающий файл

Вход AH = 4fh

DS:DX = адрес данных, возвращенных предыдущей функцией 4eh Найти 1-й файл

Выход AX = код ошибки, если CF установлен DTA заполнена данными

Описание: DS:DX указывает на буфер размером 2bh байтов с информацией, возвращенной функцией 4eh <<Найти 1-й файл>> (либо DTA, либо буфер, скопированный из DTA). Используйте эту функцию после вызова 4eh. Следующее имя файла, совпадающее по обобщенному имени и атрибуту файла, копируется в буфер по адресу DS:DX вместе с другой информацией (см. функцию 4eh о структуре файловой информации в буфере, заполняемом DOS).

Функция 54h: Дать переключатель верификации DOS

Вход AH = 54h

Выход AL = 0, если верификация выключена (OFF)

AL = 1 если верификация включена (ON)

Описание: Возвращает текущий статус верификации записи DOS. Если в AL возвращается 1, то DOS считывает обратно каждый сектор, записываемый на диск, чтобы проверить правильность записи. Функция DOS 2eh позволяет установить/изменить режим верификации.

Функция 56h: Переименовать/переместить файл

Вход AH = 56h

DS:DX = адрес старого ASCIIZ имени (путь/имя существующего файла)

ES:DI = адрес нового ASCIIZ имени (новые путь/имя)

Выход AX = код ошибки, если CF установлен

Описание: DS:DX и ES:DI указывают на строки ASCIIZ: «d:\путь\имя_файла»,0. Старое имя DS:DX должно соответствовать существующему файлу и не может содержать символов замены. Диск и путь необязательны (если опущены, они принимаются по умолчанию). Новое имя ES:DI должно описывать НЕ существующий файл. Если указан диск, он должен быть тем же, что и в старом имени. Если диск или путь опущены, принимаются текущие значения. Если старое и новое имя содержат разные пути (явно или принятые по умолчанию), то элемент каталога для файла ПЕРЕМЕЩАЕТСЯ в каталог, указанный в новом имени.

Замечание: Если ID диска в старом имени отличается от текущего диска DOS, не забывайте указывать такой же ID диска в новом имени.

Функция 57h: Установить/опросить дату/время файла

Вход AH = 57h

AL = 0 чтобы получить дату/время файла

AL = 1 чтобы установить дату/время файла

BX = дескриптор файла (handle)

CX = (если AL=1) новая отметка времени в формате <<время файла>>

DX = (если AL=1) новая отметка даты в формате <<дата файла>>

Выход AX = код ошибки, если CF установлен

CX = отметка времени файла в формате <<время файла>>

DX = отметка даты файла в формате <<дата файла>>

Описание: Регистр BX должен содержать дескриптор открытого файла (см. 3ch или 3dh). Укажите подфункцию, 0 или 1, в регистре AL. DX и CX задаются в формате памяти; например, младшие 8 бит даты находятся в DH.

Функция 59h: Дать расширенную информацию об ошибке

Вход AH = 59h DOS 3.0+

BX = 0000h (номер версии: 0000h для DOS 3.0, 3.1 и 3.2)

Выход AX = расширенный код ошибки (0, если не было ошибки)

BH = класс ошибки

BL = предлагаемое действие

CH = сфера (где произошла ошибка)

Описание: Используйте эту функцию, чтобы уточнить, что предпринять после сбоя функции DOS по ошибке (только DOS 3.0+).

Вызывайте ее:

в обработчике критических ошибок int 24h;

после любой функции int 21h , возвратившей взведенный флаг переноса (CF);

после вызова FCB-функции, возвратившей AL=0ffh.

См. <<Коды ошибок DOS>> на предмет полного списка кодов ошибок, их классов, предлагаемых действий и сфер возникновения, которые могут быть возвращены этой функцией.

Функция 5ah: Создать уникальный временный файл

Вход AH = 5ah

DS:DX = адрес строки ASCIIZ с диском и путем (заканчивается \)

CX = атрибут файла

Выход AX = код ошибки, если CF установлен

AX = дескриптор файла (если нет ошибки)

DS:DX = (не изменяется) становится полным ASCIIZ-именем нового файла

Описание: Открывает (создает) файл с уникальным именем в каталоге, указанном строкой ASCIIZ, на которую указывает DS:DX. COMMAND.COM использует эту функцию, когда создает временные «канальные» файлы, используемые при переназначении ввода/вывода. Описание пути должно быть готово к присоединению в его конец имени файла. Вы должны обеспечить минимум 12 байт в конце строки. Сама строка должна быть содержать один из вариантов указания пути:

«d:\путь\»,0 (указаны диск и путь),

«d:»,0 (текущий каталог диска),

«d:\»,0 (корневой каталог диска),

«»,0 (текущие диск и каталог)

После возврата строка DS:DX будет дополнена именем файла.

Замечания: DOS создает имя файла из шестнадцатеричных цифр, получаемых из текущих даты и времени. Если имя файла уже существует, DOS продолжает создавать новые имена, пока не получит уникальное имя. Создаваемые таким способом файлы – по существу НЕ ВРЕМЕННЫЕ, и их следует удалять посредством функции DOS 41h , когда они не нужны.

Версии: Доступна, начиная с DOS 3.0

Функция 5bh: Создать новый файл

Вход AH = 5bh DOS 3.0+

DS:DX = адрес строки ASCIIZ с именем файла

CX = атрибут файла

Выход AX = код ошибки, если CF установлен

AX = дескриптор файла, если ошибок нет

Описание: DS:DX указывает на строку ASCIIZ в форме: «d:\путь\имя_файла»,0. Если диск и/или путь опущены, они принимаются по умолчанию. Этот вызов идентичен функции DOS 3ch CREATE, с тем исключением, что он вернет ошибку, если файл с заданным именем уже существует. Файл открывается для чтения/записи в совместимом Режиме Доступа.

Функция 5ch: Блокировать/разблокировать доступ к файлу

Вход AH = 5ch

AL = Подфункция:

0 – заблокировать область файла

1 – разблокировать ранее захваченную область

BX = дескриптор файла (handle)

CX:DX = смещение ((CX * 65536) + DX) от начала файла

SI:DI = длина блокируемой области ((SI * 65536) + DI) байтов

Выход AX = код ошибки, если CF установлен

Описание: Блокирует или освобождает доступ к участку файла, идентифицируемого дескриптором в BX. Область файла, начинающаяся по логическому смещению CX:DX и имеющая длину SI:DI, блокируется (захватывается) или разблокируется (освобождается). Смещение и длина обязательны. Разделение файлов Должно быть активизировано (командой SHARE), иначе функция вернет код ошибки «неверный номер функции» Блокировка действует на операции чтения, записи и открытия со стороны порожденного или конкурирующего процесса. При попытке такого доступа (и Режиме Доступа, определенном при OPEN как «режим разделения», который запрещает такой доступ), DOS отвергает операцию через вызов int 24h

(обработчик критических ошибок) после трех попыток. DOS при этом выдает сообщение «Abort, Retry, Ignore». Рекомендуемое действие – НЕ пытаться читать файл и ожидать кода ошибки. Вместо этого попытайтесь заблокировать область и действуйте в соответствии с кодом возврата. Это позволяет избежать довольно неустойчивого состояния DOS, связанного с выполнением int 24h. Блокировка за концом файла не является ошибкой. Вы можете захватить весь файл, задав CX=0, DX=0, SI=0ffffh, DI=0ffffh и AL=0. При освобождении смещение и длина участка должны точно совпадать со смещением и длиной захваченного участка.

Замечания: Дублирование дескриптора через 45h или 46h дублирует и блокировки со смещением и длиной захваченного участка. Даже если во время OPEN выбран Режим Доступа Inherit, механизм блокировки не даст никаких привилегий доступа порожденным процессам, созданным функцией 4bh EXEC (они трактуются как отдельные). Важно, чтобы все блокировки файла были сняты до завершения программы. Если вы используете блокировку, особо отслеживайте вызовы int 23h (выход Ctrl-Break) и int 24h (выход по критической ошибке), чтобы снять блокировки до действительного завершения программы. Рекомендуется освобождать блокировки как можно скорее. Всегда блокируйте, обрабатывайте файл и освобождайте блокировку одной операцией. Версии: Доступна, начиная с DOS 3.0

Функция 62h: Дать адрес PSP

Вход AH = 62h DOS 3.0+

Выход BX = сегментный адрес PSP выполняемой программы

Описание: Эта функция возвращает в BX адрес PSP текущей программы. Используется, чтобы получить адрес параметров командной строки, адрес окружения DOS и другой полезной информации в PSP. Версии: Доступна, начиная с DOS 3.0

2.5 Коды ошибок DOS

Коды ошибок DOS показаны в таблице 2.4.

Таблица 2.4 – Коды ошибок DOS

Ошибка		Значение
Hex	Dec	
1	1	Неверный номер функции
2	2	Файл не найден
3	3	Путь не найден
4	4	Слишком много открытых файлов
5	5	Доступ отвергнут
6	6	Неверный дескриптор (handle)
7	7	Разрушены блоки управления памятью
8	8	Недостаточно памяти
9	9	Неверный адрес блока памяти
0ah	10	Неверное окружение
0bh	11	Неверный формат
0ch	12	Неверный код доступа
0dh	13	Неверная дата
0eh	14	Не используется
0fh	15	Задан неверный диск
10h	16	Нельзя удалить текущий каталог
11h	17	не то же самое устройство
12h	18	Больше нет искомых файлов

Дополнительные коды ошибок (начиная с версии DOS 3.0, выдаются функцией 59h) показаны в таблицу 2.5.

Коды 13h – 1fh соответствуют ошибкам 0 – 0ср, передаваемых в регистре DI обработчику критических ошибок int 24h. Они также совпадают с кодами ошибок в AL для int 25h, int26h

Класс ошибки. Эти коды предоставляют дополнительную информацию, чтобы помочь вам обработать ошибку. Функция 59h возвращает значение класса ошибки в регистре BH (таблица 2.6).

Таблица 2.5 – Дополнительные коды ошибок DOS

Ошибка		Значение
Hex	Dec	
0	0	Нет ошибок
13h	19	Попытка записи на защищенный диск
14h	20	Неизвестный идентификатор устройства
15h	21	Дисковод не готов
16h	22	Неизвестная команда
17h	23	Ошибка данных диска (ошибка контрольной суммы)
18h	24	Неверная длина структуры запросов
19h	25	Ошибка поиска на диске
1ah	26	Неизвестный тип носителя диска
1bh	27	Сектор не найден
1ch	28	Конец бумаги на принтере
1dh	29	Ошибка записи
1eh	30	Ошибка чтения
1fh	31	Общая ошибка
20h	32	Нарушение разделения файла
21h	33	Нарушение блокировки
22h	34	Неверная замена диска
23h	35	FCB недоступен (слишком много открытых FCB)
24h-49h		Резерв
50h	80	Файл уже существует
51h	81	Резерв
52h	82	Неизвестно что
53h	83	int 24h – сбой при обработке прерывания по критической ошибке

Таблица 2.6 – Класс ошибки

Ошибка		Значение (возвращается в BH функцией 59h)
Hex	Dec	
1	1	Нет ресурсов: не хватает FCB, памяти, каналов, дескрипторов файлов и т. п.
2	2	Временная ситуация: исчезнет со временем (например, блокировка файла)
3	3	Проблема авторизации: Вы должны иметь более высокие полномочия
4	4	Внутренняя ошибка: сбой DOS
5	5	Ошибка оборудования
6	6	Системная ошибка: сбой DOS
7	7	Ошибка приложения: некорректный запрос, неверные параметры и т. п.
8	8	Не найден запрошенный файл/элемент
9	9	Неверный формат: испорчен файл EXE, плохой диск и т.п.
0ah	10	Блокировка: файл/элемент захвачен другим пользователем
0bh	11	Ошибка носителя: неверный диск, ошибка контроля четности и т. п.
0ch	12	Уже существует файл/элемент
0dh	13	Неизвестный класс: классификация не определена или на проходит

Предлагаемое действие. Эти коды отражают подходящее действие по устранению ошибки. Идея состоит в том, чтобы сэкономить ваш код, не заставляя вас проверять коды ошибок в приложении. Вместо этого вы достигнете совместимости вверх, выполняя предлагаемые ниже действия. Функция 59h возвращает эти коды в регистре BL (таблица 2.7).

Таблица 2.7 – Предлагаемые действия для устранения ошибки

Ошибка		Значение (возвращается в BL функцией 59h)
Hex	Dec	
1	1	Повторить: повторите операцию несколько раз. Если ошибка повторяется, запросите пользователя, продолжить или закончить работу.
2	2	Задержать повтор: подождите немного и повторите операцию. Если ошибка повторяется, запросите пользователя, продолжить или закончить работу.
3	3	Ввод пользователя: если данные для DOS были введены пользователем, предложите ему повторить ввод (может быть неверный идентификатор диска или путь).
4	4	Снять: снимите приложение. Можно выполнить операции завершения, какие, как закрытие файлов, обновление индексов, освобождение памяти и т. п.
5	5	Немедленный выход: снимайте немедленно без попытки завершения. Система в подозрительном состоянии, и немедленный выход – лучшее продолжение.
6	6	Игнорировать: ошибка ни на что не влияет
7	7	Повторить после действия пользователя: требуется вмешательство пользователя (например, установка дискеты). После этого повторите операцию.

Сфера ошибки. Эти коды служат для того, чтобы помочь вам определить место ошибки. Функция 59h возвращает эти коды в регистре CH (таблица 2.8).

Таблица 2.8 – Указание места ошибки

Ошибка		Значение (возвращается в CH функцией 59h)
Hex	Dec	
1	1	Неизвестно: не существует определенной области для привязки ошибки
2	2	Блочное устройство: ошибка дискового или ленточного устройства
3	3	Резерв
4	4	Символьное устройство
5	5	Память

Обработка ошибок совместима снизу вверх для всех версий DOS. Применимы следующие общие правила:

- DOS 1.x: индицирует некоторые ошибки, помещая в AL при возврате 0ffh.
- DOS 2.x: новые вызовы 2.x индицируют ошибки, устанавливая флаг CF=1 и помещая код ошибки в AX.
- DOS 3.x: вызовы 3.x по-прежнему помещают код ошибки в AX при CF=1, но нет гарантии, что будущие версии будут поступать так же. Советуем использовать функцию 59h для получения информации об ошибке.

2.6 Вопросы для самопроверки

1. Функции BIOS
2. Функции DOS
3. Вывод строки на экран
4. Создание файла
5. Получение даты
6. Получение времени
7. Установки видеорежима
8. Текстовые видеорежимы
9. Установка и изменение знакогенератора
10. Ввод с клавиатуры
11. Взятие вектора прерывания
12. Установка вектора прерывания
13. Определение текущего дисковода
14. Чтение из файла
15. Запись в файл
16. Открытие файла
17. Физическое чтение диска.
18. Установка текущего дисковода.
19. Работа с блоками памяти.
20. Запуск программы из прикладной программы.
21. Системное время.
22. Выделение памяти из программы пользователя.
23. Функции работы с временем
24. Работа DOS с памятью
25. Функции работы с датой

3 ПРИМЕРЫ ПРОГРАММ

3.1 Ввод-вывод символьной информации

В этой главе рассмотрены различные аспекты ввода-вывода символьной информации с использованием функций DOS и BIOS, а также непосредственный вывод информации в область видеопамяти. Все приемы программирования представлены в виде примеров программ, решающих конкретные задачи.

Задача 3.1.1. Вывести на середину пустого экрана мигающее слово «ТЕКСТ» в текстовом режиме CGA 80 символов * 25 строк.

а) Используя функции DOS.

б) Используя функции BIOS.

в) Помещая символы непосредственно в экранную область.

а. Использование функций DOS

```
Assume CS: Code, DS: Code
Code  SEGMENT
        org    100h
Start: mov    ax,cs
        mov    ds,ax
 ; Установка видеорежима 03
        mov    ah,0    ; Функция 0
        mov    al,3    ; Режим 3
        int    10h
 ; Вывод текста (мигание в ДОС нельзя)
        mov    ah,9    ; Функция 9
        lea    dx,text ; Смещение текста (DS)
        int    21h
 ; Ожидание ввода клавиши пробела
loop1: mov  ah,7    ; Функция 7 (нефильтр. ввод без эха)
        int    21h
        cmp    al,' '   ; Пробел ?
        jnz    loop1 ; Нет !
 ; Выход из программы
        int    20h
text    db     25 dup(0ah),0dh ; Очистка экрана
        db     38 dup(20h),'ТЕКСТ' ; Вывод слова на середину
        db     12 dup(0ah),0dh,'$' ; На середину экрана
Code ENDS
```

```
        END  Start
```

б. Использование функций BIOS

```
Assume CS: Code, DS: Code
Code  SEGMENT
        org    100h
Start: mov   ax,cs
        mov   ds,ax
        mov   ah,0   ; Функция 0; Установка видеорежима 3
        mov   al,3   ; Режим 3
        int     10h
; Очистка экрана и задание атрибута мигания
        mov   ah,6   ; Иниц. или прокрутка окна вверх
        mov   al,0   ; Очистка всего окна
        mov   bh,87h ; Установка атрибута мерцания
        mov   ch,0   ; Y лев. верхн. угла
        mov   cl,0   ; X лев. верхн. угла
        mov   dh,24 ; Y прав. нижн. угла
        mov   dl,89  ; X прав. нижн. угла
        int     10h
; Установка позиции курсора
        mov   ah,2   ; Функция 2
        mov   bh,0   ; Страница 0
        mov   dh,12 ; Строка 12
        mov   dl,38  ; Колонка 38
        int     10h
; Вывод текста
        mov   ah,0Eh ; Функция 0Eh
        lea     si,text          ; Смещение текста (DS)
        mov   bl,7
loop2: cmp  byte ptr [si],0 ; Проверка на конец текста
        jz      loop1 ; Выход
        mov   al,[si]
        int     10h
        inc     si
        jmp    short loop2
; Ожидание ввода клавиши пробела
loop1: mov  ah,0   ; Функция 0
```

```
        int     16h
        cmp   al,' '    ; Пробел ?
        jnz     loop1 ; Нет !
         int     20h    ; Выход из программы
text    db      'ТЕКСТ',0 ; Вывод слова на середину
Code ENDS
        END Start
```

в. Непосредственный вывод в экранную область

```
Assume CS: Code, DS: Code
Code  SEGMENT
        org     100h
Start: mov   ax,cs
        mov   ds,ax
; Установка видеорежима 3
        mov   ah,0    ; Функция 0
        mov   al,3    ; Режим 3
        int     10h
; Очистка экрана
        mov   ax,0b800h
        mov   es,ax  ; ES = B800h
        xor     si,si    ; Обнуление регистра-источника
        xor     di,di   ; Обнуление регистра назначения
        mov   cx,2000 ; Инициализация счетчика
        mov   ax,0700h ;
        rep     stosw
; Вывод текста
        lea     si,text         ; Смещение текста (DS)
        mov   di,2000         ; Начальная ячейка экр. памяти
        mov   ah,8fh          ; Атрибут
loop2: cmp  byte ptr [si],0 ; Проверка на конец текста
        jz      loop1 ; Выход
        mov   al,[si]
        mov   es:[di],ax     ; На экран
        inc      si
        inc     di
        inc     di
        jmp    short loop2
```

```
; Ожидание ввода клавиши пробела
loop1: mov  ah,0   ; Функция 0
       int    16h
       cmp    al,' '   ; Пробел ?
       jnz    loop1 ; Нет !
; Выход из программы
       int    20h
text   db     'ТЕКСТ',0   ; Вывод слова на середину
Code ENDS
       END  Start
```

Задача 3.1.2. Используя различные функции прерывания 10 BIOS, перепрограммировать символ знакогенератора с кодом 41H в какой-нибудь псевдографический символ (рисунок 3.1), заполнить весь экран этим символом, инициализировать окно, определяемое координатами 2,2 – 10,32, и вывести на всю первую строку этого окна символ с кодом 42H с повышенной яркостью и с миганием. Выход из программы должен осуществляться по нажатию клавиши пробел.

Символ 41h

1			x	x	x	x			3ch
2		x					x		42h
3	x							x	81h
4	x		x			x		x	0a5h
5	x							x	81h
6	x			x	x			x	99h
7	x			x	x			x	99h
8	x							x	99h
9	x							x	81h
10	x		x			x		x	0a5h
11	x			x	x			x	99h
12		x					x		42h
13			x	x	x	x			3ch
14									0

Символ 42h

1			x	x	x	x			3ch
2		x					x		42h
3	x							x	81h
4	x		x			x		x	0a5h
5	x							x	81h
6	x			x	x			x	99h
7	x			x	x			x	99h
8	x							x	99h
9	x							x	81h
10	x			x	x			x	99h
11	x		x			x		x	0a5h
12		x					x		42h
13			x	x	x	x			3cg
14									0

Рисунок 3.1 – Перепрограммирование знакогенератора

```
Assume  CS: Code, DS: Code
Code  SEGMENT
       org    100h
Start:                    ; Установка видеорежима 3
```

```
        mov   ah,0   ; Функция 0
        mov   al,3   ; Режим 3
        int     10h
; Перепрограммирование буквы А (код 41H)
        mov   ah,11h        ; Функция 11h подфункция 12h
        mov   al,12h        ; Загрузка шрифта ПЗУ 8x8
        mov   bl,0          ; Блок знакогенератора
        int     10h
        mov   bh,8          ; Высота символа в точках
        mov   bl,0          ; Блок знакогенератора
        mov   cx,1          ; Количество символов, описанных в таблице
        mov   dx,41h         ; Код, соотв. первому символу таблицы
        mov   ax,cs
        mov   es,ax         ; ES:BP – адрес таблицы
        mov   bp,offset tabl
        mov    ah,11h       ; Функция 11h подфункция 10h
        mov   al,10h        ; Загрузка шрифта пользователя
        int     10h
; Установка позиции курсора 0,0
        mov   ah,2          ; Функция 2
        mov   bh,0   ; Страница 0
        mov   dh,0   ; Строка 0
        mov   dl,0   ; Столбец 0
        int     10h
; Запись символа в позицию курсора
        mov   ah,0ah        ; Функция 0Ah
        mov   al,41h        ; Символ 41h
        mov   bh,0          ; Страница 0
        mov   cx,2000       ; Коэффициент повторения
        int     10h
; Инициализация или прокрутка окна вниз (2,2 – 10,32)
        mov   ah,6h ; Функция 6
        mov   al,0          ; Очистка окна
        mov   bh,8ch        ; Атрибут (бит 7 – мигание, бит
                            ; 3 – интенсивность,
                            ; биты 0-2 – цвет переднего плана,
                            ; биты 4-6 – цвет заднего плана
```

```
        mov   cx,202h       ; Верхний левый угол
        mov   dx,0a20h      ; Правый нижний угол
        int   10h
; Установка позиции курсора 2,2
        mov   ah,2   ; Функция 2
        mov   bh,0   ; Страница 0
        mov   dh,2   ; Строка 2
        mov   dl,2   ; Столбец 2
        int   10h
; Запись символа в позицию курсора
        mov   ah,0ah        ; Функция 0Ah
        mov   al,42h        ; Символ 42h
        mov   bh,0          ; Страница 0
         mov  cx,30         ; Коэффициент повторения
        int   10h
; Ожидание ввода клавиши пробела
loop1: mov  ah,0   ; Функция 0
        int   16h
        cmp   al,' '   ; Пробел ?
        jnz   loop1 ; Нет !
; Выход из программы
        int   20h
; Таблица перепрограммирования знакогенератора
; (здесь могут быть другие коды)
tabl   db    0ffh,82h,84h,88h,90h,0a0h,0c0h,0ffh
Code ENDS
        END  Start
```

Задача 3.1.3. Усовершенствовать предыдущую программу так, чтобы после перепрограммирования символов и вывода сообщения нажатие пробела приводило к выходу из программы, а нажатие любой другой клавиши – циклически – к возврату исходного изображения символов и перепрограммированного. Повторный вывод сообщения следует блокировать.

```
Assume CS: Code;, DS: Code
Code  SEGMENT
        org   100h
Start: jmp    start1
flag   db     0       ; Флаг вывода сообщения
```

```
 ; Установка видеорежима 3
start1: mov  ah,0   ; Функция 0
        mov  al,3   ; Режим 3
        int  10h
; Перепрограммирование буквы A (код 41H)
        mov  ah,11h       ; Функция 11h подфункция 11h
        mov  al,11h       ; Загрузка шрифта ПЗУ 8x14
        mov  bl,0         ; Блок знакогенератора
        int  10h
loop2: mov  bh,14 ; Высота символа в точках
        mov  bl,0         ; Блок знакогенератора
        mov  cx,2         ; Количество символов, описанных в таблице
        mov  dx,41h       ; Код, соотв. первому символу таблицы
        mov  ax,cs
        mov  es,ax        ; ES:BP – адрес таблицы
        mov  bp,offset tabl
        mov  ah,11h       ; Функция 11h подфункция 10h
        mov  al,10h       ; Загрузка шрифта пользователя
        int 10h           ; Вывод сообщения
        test  byte ptr flag,0ffh ; Проверка флага печати
        jnz   m1          ; Пропуск печати, если флаг взведен
        mov  ah,9         ; Функция вывода строки
        lea  dx,text      ; Смещение текста
        int  21h
        mov  byte ptr flag,1
; Ожидание ввода клавиши пробела
m1:   mov  ah,0   ; Функция 0
        int  16h
        cmp  al,' '       ; Пробел ?
        jz   loop1 ; Да !
        mov  ah,11h       ; Функция 11h подфункция 11h
        mov  al,11h
; Загрузка шрифта ПЗУ 8x14
        mov  bl,0   ; Блок знакогенератора
        int  10h
        mov  ah,0   ; Функция 0
        int  16h
```

```
        cmp   al,' '   ; Пробел ?
        jz    loop1 ; Да !
        jmp   loop2 ; Выход из программы
loop1: int    20h
; Таблица перепрограммирования знакогенератора
; (здесь могут быть другие коды)
tabl   db     3ch,42h,81h,0a5h,81h,99h,99h,99h,81h,0a5h,99h,42h,3ch,0
       db     3ch,42h,81h,0a5h,81h,99h,99h,99h,81h,99h,0a5h,42h,3ch,0
text   db     0dh,0ah,0ah,'Замена символов произведена:'
       db     0dh,0ah,'41h = A',0dh,0ah,'42h = B',0dh,0ah,'$'
Code ENDS
       END  Start
```

Задача 3.1.4. Написать программу, которая поместила бы процедуру, выводящую сообщение о своем местонахождении, по адресу 8800h. Выход из пересланной процедуры должен осуществляться по клавише ENTER. После возврата в основную программу должно выводиться сообщение об успешном завершении программы.

Выход из основной программы должен осуществляться по нажатию клавиши пробела. После переписывания дальней процедуры в область 8800h следует стереть ее текст в основной программе для демонстрации того, что на исходном месте она не может быть выполнена.

Для выполнения дальнего перехода следует использовать команду RETF, поместив предварительно в стек сегмент и смещение желаемого места перехода. Перед переходом в дальнюю процедуру следует позаботиться о правильном значении сегментного регистра DS. О том же следует позаботиться и при возвращении в основную программу. Попробуйте выполнить эту программу с другими адресами места назначения, в частности, в экранной области: B800h, BC00h, B900h.

а) Использование команды CALL dword ptr

```
Assume CS: Code;, DS: Code
Code  SEGMENT
        org   100h
Start: jmp    start1
len    equ    dend-dal     ; Длина пересылаемой процедуры
s_dal equ     8800h        ; Сегмент места пересылки процедуры
addr   dw     0,s_dal      ; Дальний адрес места назначения
start1: cld
```

```
        mov   ax,cs
        mov   ds,ax          ; DS = CS
; Перенос процедуры DAL в область 8800h
        les   di,dword ptr addr ; ES:DI = s_dal:0000
        lea   si,dal         ; SI = offset DAL
        mov   cx,len         ; CX = длине пересылаемой проц.
        rep   stosb          ; DS:SI □ ES:DI
; Стирание процедуры в тексте
        mov   ax,cs
        mov   es,ax          ; ES = CS
        lea   di,dal         ; DI = offset DAL
        mov   cx,len         ; CX = длине пересылаемой проц.
        xor   ax,ax          ; AX=0 для стирания процедуры
        rep   movsb          ; AX(0) □ ES:DI
; Вычисление значения DS для дальней процедуры
        push  ds             ; DS □ ctack
        lea   ax,dal         ; offset DAL
        xor   dx,dx          ; Мы знаем, что это смещение
        mov   cx,16          ; нацело делится на 16
        div   cx             ; AX = (offset DAL)/16
        mov   dx,ax          ; DX = (offset DAL)/16
        mov   ax,s_dal
        sub   ax,dx          ; AX = s_dal – (offset DAL)/16
        mov   ds,ax          ; DS = s_dal – (offset dal)/16
;***** Переменная часть *** Переход на дальнюю процедуру
         call  dword ptr addr      ; Переход на s_dal:0000
;***** Конец переменной части **********************
; Точка возврата – печать сообщения о завершении
l:      pop   ds      ; Восстановление местного DS
        mov   ah,9    ; Функция 9
        lea   dx,text        ; DS:DX – смещение текста сообщения
        int   21h            ; Вызов функции DOS
; Ожидание ввода клавиши пробела
loop1: mov  ah,0             ; Функция 0
        int   16h            ; Клавиатурное прерывание
        cmp   al,' '         ; Пробел ?
        jnz   loop1 ; Нет !
```

```
        int     20h
text    db      0dh,0ah,0ah
        db      'Программа выполнена',0dh,0ah,'$'
; Выравнивание начала засылаемой процедуры на начало параграфа
        if ($-start)mod 16 (если не 0)          ; Псевдокоманда
                org $+(16-($-start)mod 16)      ; условного
        endif                                   ; ассемблирования
; Процедура, которая пересылается в область s_dal.
; Она начинается здесь на границе параграфа.
; Когда эта процедура находится в области s_dal,
; значение DS = s_dal – (offset dal)/16
dal     proc    far     ; Дальняя процедура (т.е. будет дальний возврат – retf)
        mov     ah,9    ; Функция 9
        lea     dx,text1        ; DS:DX – смещение текста сообщения
        int     21h             ; Вызов функции DOS
; Ожидание ввода клавиши ENTER
dal1:   mov     ah,0            ; Функция 0
        int     16h             ; Клавиатурное прерывание
        cmp     al,0dh          ; ENTER ?
        jnz     dal1            ; Нет !
        ret                      ; Возврат дальний !!!
text1   db      0dh,0ah,0ah,'Я нахожусь по адресу S_DAL',0dh,0ah,'$'
dal     endp
dend: ; Метка для определения конца пересылаемой процедуры
Code ENDS ; Конец сегмента (кодового)
        END  Start ; Указание точки входа в программу
```

б) Использование команды RETF

```
Assume CS: Code;, DS: Code
Code  SEGMENT
        org     100h
len     equ     dend-dal        ; Длина пересылаемой процедуры
s_dal   equ     8800h           ; Сегмент места пересылки процедуры
Start:
        .386
        cld
        mov     ax,cs
        mov     ds,ax           ; DS = CS
```

```
; Перенос процедуры DAL в область 8800h
        mov   ax,s_dal
        mov   es,ax         ; ES = s_dal
        lea   si,dal        ; SI = offset DAL
        xor   di,di         ; DI = 0
        mov   cx,len        ; CX = длине пересылаемой проц.
        rep   stosb         ; DS:SI □ ES:DI
; Стирание процедуры в тексте
        mov   ax,cs
        mov   es,ax         ; ES = CS
        lea   di,dal        ; DI = offset DAL
        mov   cx,len        ; CX = длине пересылаемой проц.
        xor   ax,ax         ; AX=0 для стирания процедуры
        rep   movsb         ; AX(0) □ ES:DI
; Вычисление значения DS для дальней процедуры
        push  ds            ; DS □ ctack
        lea   ax,dal        ; offset DAL
        xor   dx,dx         ; Мы знаем, что это смещение
        mov   cx,16         ; нацело делится на 16
        div   cx            ; AX = (offset DAL)/16
        mov   dx,ax         ; DX = (offset DAL)/16
        mov   ax,s_dal
        sub   ax,dx         ; AX = s_dal – (offset DAL)/16
        mov   ds,ax         ; DS = s_dal – (offset dal)/16
;**** Переменная часть  Подготовка возврата в стеке
        mov   ax,cs
        push  ax            ; CS □ stack
        lea   ax,l          ; CS:(offset L) – точка возврата
        push  ax            ; offset L □ stack
; Подготовка перехода в стеке
        mov   ax,s_dal
        push  ax            ; s_dal □ stack
        xor   ax,ax         ; s_dal:0000 – Начало проц. в памяти
        push  ax            ; 0 □ stack
; Переход на дальнюю процедуру
        retf                ; Переход на s_dal:0000
;*************** Конец переменной части *****************
```

```
; Точка возврата – печать сообщения о завершении
l:      pop   ds      ; Восстановление местного DS
        mov   ah,9    ; Функция 9
        lea   dx,text       ; DS:DX – смещение текста сообщения
        int   21h           ; Вызов функции DOS
; Ожидание ввода клавиши пробела
loop1: mov  ah,0            ; Функция 0
        int   16h           ; Клавиатурное прерывание
        cmp   al,' '  ; Пробел ?
        jnz   loop1 ; Нет !
        int   20h
text    db    0dh,0ah,0ah,'Программа выполнена',0dh,0ah,'$'
; Выравнивание начала засылаемой процедуры на начало параграфа
        if ($-start)mod 16 (если не 0)          ; Псевдокоманда
                org $+(16-($-start)mod 16)      ; условного
        endif                                   ; ассемблирования
; Процедура, которая пересылается в область s_dal.
; Она начинается здесь на границе параграфа.
; Когда эта процедура находится в области s_dal,
; значение DS = s_dal – (offset dal)/16
dal     proc  far     ; Дальняя процедура (т.е. будет дальний возврат – retf)
        mov   ah,9    ; Функция 9
        lea   dx,text1      ; DS:DX – смещение текста сообщения
        int   21h           ; Вызов функции DOS
; Ожидание ввода клавиши ENTER
dal1:  mov  ah,0            ; Функция 0
        int   16h           ; Клавиатурное прерывание
        cmp   al,0dh        ; ENTER ?
        jnz   dal1          ; Нет !
        ret                 ; Возврат дальний !!!
text1  db    0dh,0ah,0ah,'Я нахожусь по адресу S_DAL',0dh,0ah,'$'
dal     endp
dend: ; Метка для определения конца пересылаемой процедуры
Code ENDS ; Конец сегмента (кодового)
        END  Start ; Указание точки входа в программу
```

в) Использование команды JMP dword ptr

```
Assume CS: Code;, DS: Code
```

```
Code SEGMENT
        org   100h
Start: jmp    start1
len    equ    dend-dal      ; Длина пересылаемой процедуры
s_dal  equ    8800h         ; Сегмент места пересылки процедуры
addr   dw     0,s_dal       ; Дальний адрес места назначения
start1: cld
        mov   ax,cs
        mov   ds,ax         ; DS = CS
; Перенос процедуры DAL в область 8800h
        les   di,dword ptr addr   ; ES:DI = s_dal:0000
        lea   si,dal        ; SI = offset DAL
        mov   cx,len        ; CX = длине пересылаемой проц.
        rep   stosb         ; DS:SI □ ES:DI
; Стирание процедуры в тексте
        mov   ax,cs
        mov   es,ax         ; ES = CS
        lea   di,dal        ; DI = offset DAL
        mov   cx,len        ; CX = длине пересылаемой проц.
        xor   ax,ax         ; AX=0 для стирания процедуры
        rep   movsb         ; AX(0) □ ES:DI
; Вычисление значения DS для дальней процедуры
        push  ds            ; DS □ ctack
        lea   ax,dal        ; offset DAL
        xor   dx,dx         ; Мы знаем, что это смещение
        mov   cx,16         ; нацело делится на 16
        div   cx            ; AX = (offset DAL)/16
        mov   dx,ax         ; DX = (offset DAL)/16
        mov   ax,s_dal
        sub   ax,dx         ; AX = s_dal – (offset DAL)/16
        mov   ds,ax         ; DS = s_dal – (offset dal)/16
;**** Переменная часть  Подготовка возврата в стеке
        mov   ax,cs
        push  ax            ; CS □ stack
        lea   ax,l          ; CS:(offset L) – точка возврата
        push  ax            ; offset L □ stack
; Переход на дальнюю процедуру
```

```
        jmp   dword ptr addr      ; Переход на s_dal:0000
;*************** Конец переменной части *****************
; Точка возврата – печать сообщения о завершении
l:      pop   ds                  ; Восстановление местного DS
        mov   ah,9                ; Функция 9
        lea   dx,text             ; DS:DX – смещение текста сообщения
        int   21h                 ; Вызов функции DOS
; Ожидание ввода клавиши пробела
loop1: mov  ah,0                  ; Функция 0
        int   16h                 ; Клавиатурное прерывание
        cmp   al,' '              ; Пробел ?
        jnz   loop1 ; Нет !
        int   20h
text    db    0dh,0ah,0ah,'Программа выполнена',0dh,0ah,'$'
; Выравнивание начала засылаемой процедуры на начало параграфа
        if ($-start)mod 16 (если не 0)          ; Псевдокоманда
              org $+(16-($-start)mod 16)        ; условного
        endif                                   ; ассемблирования
; Процедура, которая пересылается в область s_dal.
; Она начинается здесь на границе параграфа.
; Когда эта процедура находится в области s_dal,
; значение DS = s_dal – (offset dal)/16
dal     proc  far     ; Дальняя процедура (т.е. будет дальний возврат – retf)
        mov   ah,9    ; Функция 9
        lea   dx,text1        ; DS:DX – смещение текста сообщения
        int   21h             ; Вызов функции DOS
; Ожидание ввода клавиши ENTER
dal1:  mov   ah,0             ; Функция 0
        int   16h             ; Клавиатурное прерывание
        cmp   al,0dh          ; ENTER ?
        jnz   dal1            ; Нет !
        ret                   ; Возврат дальний !!!
text1  db    0dh,0ah,0ah,'Я нахожусь по адресу S_DAL',0dh,0ah,'$'
dal     endp
dend: ; Метка для определения конца пересылаемой процедуры
Code ENDS ; Конец сегмента (кодового)
        END  Start ; Указание точки входа в программу
```

3.2 Работа с файлами

Задача 3.2.1. Используя функции прерывания 21h DOS, написать программу, которая

- создает в текущем каталоге новый файл с именем, соот ветствующим фамилии студента
- записывает в созданный файл текст длиной не менее 40 символов
- изменяет дату создания файла на 11 ноября 1991 года (при этом время создания должно остаться неизменным)

При возникновении ошибки во время выполнения какой-либо функции должно выводиться сообщение о возникновении ошибки, и программа должна завершаться. Классифицировать возникшую ошибку не надо.

```
Assume CS: Code, DS: Code
Code  SEGMENT
        org   100h
Start   proc  near
        mov   ax,cs
        mov   ds,ax         ; DS = CS
; Создание нового файла
        mov   ah,5bh        ; Функция создания нового файла
        mov   cx,2          ; Атрибут "скрытый"
        lea   dx,file       ; Адрес спецификации файла
        int   21h           ; Функция DOS
        jnc   m1            ; Переход, если ошибки нет
; Вывод сообщения при возникновении ошибки
err1:   mov   ah,9          ; Функция вывода строки на экран
        lea   dx,meserr     ; Адрес сообщения об ошибке
        int   21h
        int   20h
m1:     mov   handle,ax     ; Сохранение дескриптора файла
; Запись текста в файл
        mov   ah,40h        ; Функция записи в файл
        mov   bx,handle     ; Дескриптор файла
        mov   cx,40         ; Длина записываемого текста
        lea   dx,text       ; Адрес записываемого текста
        int   21h
        jc    err1          ; Переход на вывод сообщения при ошибке
```

```
; Получение даты и времени создания файла
        mov   ah,57h        ; Функция работы с датой и временем
        mov   al,0          ; Подфункция получения даты и времени
        mov   bx,handle     ; Дескриптор файла
        int   21h
        jc    err1          ; Переход на вывод сообщения при ошибке
; Изменение даты создания файла
        mov   al,1    ; Подфункция установки даты и времени
        mov   dx,0001011101101011b ; Год, месяц и день
        int   21h
        jc    err1    ; Переход на вывод сообщения при ошибке
        int   20h     ; Выход из программы
file    db    'roschin.ie4',0
meserr db     'Error during program execution$'
text    db    '*** This file was created by Roschin *** '
handle dw     ? ; Место для дескриптора файла
Start   endp
Code ENDS
        END   Start
```

Задача 3.2.2. Используя функции прерывания 21h DOS, написать программу, которая

- создает в текущем каталоге новый файл с именем, соответствующим фамилии студента
- записывает в созданный файл строку, вводимую с клавиатуры (только ее, и ничего лишнего) в каталог ...\PROGRAM\FILES

При возникновении ошибки во время выполнения какой-либо функции должно выводиться сообщение о возникновении ошибки, и программа должна завершаться. Классифицировать возникшую ошибку не надо.

```
Assume CS: Code, DS: Code
Code  SEGMENT
        org   100h
Start   proc  near
        mov   ax,cs
        mov   ds,ax  ; DS = CS Далее идет ввод текста с клавиатуры
        mov   ah,0ah        ; Функция ввода с клавиатуры
        lea   dx,buf        ; Адрес буфера
        int   21h
```

```
; Создание нового файла
        mov   ah,5bh        ; Функция создания нового файла
        mov   cx,2          ; Атрибут "скрытый"
        lea   dx,file       ; Адрес спецификации файла
        int   21h           ; Функция DOS
        jc    err1          ; Переход, если ошибки нет
        mov   handle,ax     ; Сохранение дескриптора файла
; Запись текста в файл
        mov   ah,40h        ; Функция записи в файл
        mov   bx,handle     ; Дескриптор файла
        xor   ch,ch
        mov   cl,buf+1      ; Длина записываемого текста
        lea   dx,buf+2      ; Адрес записываемого текста
        int   21h
        jc    err1          ; Переход на вывод сообщения при ошибке
; Закрытие файла
        mov   ah,3eh        ; Функция закрытия файла
        mov   bx,handle     ; Дескриптор файла
        int   21h
        jc    err1
        int   20h           ; Выход из программы
; Вывод сообщения при возникновении ошибки
err1:   mov   ah,9          ; Функция вывода строки на экран
        lea   dx,meserr     ; Адрес сообщения об ошибке
        int   21h
        int   20h
file    db    'roschin.ie4',0
buf     db    255,255 dup(0)
meserr db     'Error during program execution$'
handle dw     ? ; Место для дескриптора файла
Start   endp
Code ENDS
        END   Start
```

3.3 Работа с графикой

Задача 3.3.1. Написать программу, которая переключает ЭВМ в графический режим CGA (4 цвета, 320 * 200 точек) и заполняет экранную область (8000h байтов, начиная с адреса B800h) заданным значением (color), выводит на середину экрана изображение мяча (4*4) точек и обеспечивает его движение с отражением от краев экрана.

```
Assume  CS: Code, DS: Code
screen_size  equ   8000h        ; Размер экрана в байтах
color          equ  0       ; Цвет: 4 точки в байте, 55р – голубой,
Code  SEGMENT          ;0aah – сиреневый, 0ffh – белый
      org   100h
Start  proc   near
      mov   ah,0   ; Функция установки видеорежима
      mov   al,4   ; Графический режим 4 цвета 320 * 200
      int    10h
      call   lab1
      call   lab2
      int    20h
Start  endp
lab2   proc   near
      mov   dl,40  ; Координата X (0 – 79)
      mov   dh,50 ; Координата Y (0 – 99)
      mov   cl,1    ; Дельта X
      mov   ch,1    ; Дельта Y
lab20: call   ball1  ; Построение светлого мяча
      push  cx
      mov   cx,5000       ; Задержка
lab21: loop  lab21
      pop    cx
      call   ball0          ; Стирание мяча
      cmp   dl,79
      jnz    lab22
      neg    cl
      jmp    lab23
lab22: cmp  dl,0
      jnz    lab23
      neg    cl
```

```
lab23: add   dl,cl
       cmp   dh,99
       jnz   lab24
       neg   ch
       jmp   lab25
lab24: cmp   dh,0
       jnz   lab25
       neg   ch
lab25: add   dh,ch
       jmp   lab20
       ret
lab2   endp
ball1  proc  near   ; Рисование мяча
       mov   ax,0b800h
       mov   es,ax
       mov   ax,80
       mul   dh             ; 80 * Y
       xor   bh,bh
       mov   bl,dl
       add   bx,ax          ; 80 * Y + X
       mov   es:byte ptr [bx],3ch
       add   bx,2000h
       mov   es:byte ptr [bx],0ffh
       sub   bx,2000h-80
       mov   es:byte ptr [bx],0ffh
       add   bx,2000h
       mov   es:byte ptr [bx],3ch
       ret
ball1  endp
ball0  proc  near   ; Стирание мяча
       mov   ax,0b800h
       mov   es,ax
       mov   ax,80
       mul   dh     ; 80 * Y
       xor   bh,bh
       mov   bl,dl
       add   bx,ax ; 80 * Y + X
```

```
        mov   es:byte ptr [bx],0
        add   bx,2000h
        mov   es:byte ptr [bx],0
        sub   bx,2000h-80
        mov   es:byte ptr [bx],0
        add   bx,2000h
        mov   es:byte ptr [bx],0
        ret
ball0   endp
lab1    proc  near
        mov   ax,cs
        mov   ds,ax          ; DS = CS
        mov   ax,0b800h
        mov   es,ax          ; ES = B800h
        xor   si,si          ; Обнуление регистра-источника
        xor   di,di          ; Обнуление регистра назначения
        mov   cx,screen_size/2 ; Инициализация счетчика
        mov   al,color       ; Цвет
    rep stosb
        ret
lab1    endp
Code    ENDS
        END   Start
```

3.4 Работа со звуком

В простейшем случае в IBM PC для генерации звука используется микросхема интегрального таймера 8253 или 8254. Эта микросхема имеет три независимых канала, каждый из которых может программироваться для работы в режиме делителя частоты или генератора одиночных импульсов (рисунок 3.2). Каждый канал содержит 16-разрядный счетчик, в который записывается значение делителя частоты или коэффициента пересчета (в зависимости от режима работы). Каждый канал имеет вход частоты (clk) и вход разрешения (gate). На вход частоты всех каналов подается импульсный сигнал частотой 1,19 МГц. Канал 0 микросхемы таймера используется для выработки сигнала прерывания по таймеру (частотой 18,2 Гц). Канал 1 работает в режима генерации одиночных импульсов через каждые 15 мкс. Этот сигнал используется для регенерации динамической памяти ЭВМ.

Канал 2 микросхемы исходно программируется для работы в режиме делителя частоты. Выход канала используется для генерации звука через встроенный динамик. Для управления звуком используются биты 0 и 1 системного порта B (микросхема 8255). Бит 0 используется для разрешения прохождения сигнала на выход канала 2 таймера. Сигнал с выхода канала 2 подается на схему "И", на второй вход который подается сигнал бита 1 системного порта B. Этот сигнал может разрешать или запрещать прохождения сигнала с выхода канала 2 таймера, а при закрытом канале 2 (битом 0 порта B) сигнал бита 1 порта B может использоваться для непосредственной генерации звука в динамике.

Адрес системного порта B – 61h, адреса каналов таймера – 40h, 41h, 42h, 43h – для каналов 0, 1, 2 и управляющего регистра соответственно. Ниже рассмотрены примеры генерации звука с помощью сигнала бита 1 системного порта B, а также с помощью таймера. Рассмотрен случай извлечения звука с использованием прерывания.

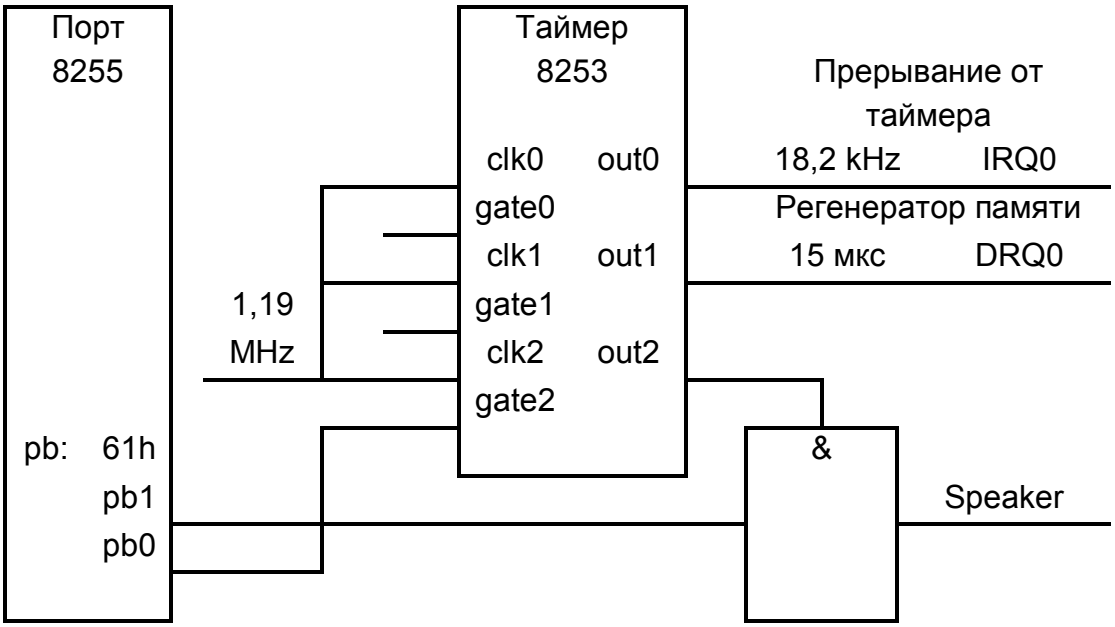


Рисунок 3.2 – Схема ггенерации звука в IBM PC

Примеры генерации звука

Задача 3.4.2.1. Написать программу, издающую различные звуки при нажатии на клавиши '1' и '2'. Для генерации звука следует использовать сигнал бита 1 системного порта B. Выход из программы должен осуществляться по нажатию клавиши 'q'.

```
Assume  CS: Code, DS: Code
Code  SEGMENT
        org    100h
frequency1          equ    300    ; Задержка переключения 1
```

```
frequency2      equ   500   ; Задержка переключения 2
number_cycles1  equ   1000  ; Количество циклов (длит.)
number_cycles2  equ   600   ; Количество циклов (длит.)
port_b          equ   61h   ; Адрес системного порта B
        .286
Start   proc  near
        mov   ax,cs
        mov   ds,ax ; DS = CS
beg1:   call  kbin  ; Опрос клавиатуры
        cmp   al,'1'  ; = '1' ?
        jnz   beg2  ; Нет
        call  ton1  ; Звук высоты 1
        jmp   beg1  ; Переход на начало цикла
beg2:   cmp   al,'2'  ; = '2' ?
        jnz   beg3  ; Нет
        call  ton2  ; Звук высоты 2
        jmp   beg1  ; Переход на начало цикла
beg3:   cmp   al,'q'  ; = 'q' ?
        jnz   beg1  ; Нет
        int   20h   ; Выход из программы
start   endp
ton2    proc  near  ; Процедура генерации звука 2
        mov   dx,number_cycles2       ; Длительность 2
        mov   di,frequency2           ; Задержка 2
        jmp   ton0  ; Переход на универсальную процедуру генерации звука
ton1    proc  near  ; Процедура генерации звука 1
        mov   dx,number_cycles1       ; Длительность 1
        mov   di,frequency1           ; Задержка 1
; Универсальная процедура генерации звука
; DX – количество циклов, DI – задержка
ton0    proc  near
        cli                           ; Запрещение прерываний
        in    al,port_b               ; Чтение сост. системн. порта B
        and   al,11111110b            ; Отк. динамика от таймера
ton01:  or    al,00000010b            ; Включение динамика
        out   port_b,al               ; Запись в системный порт B
        mov   cx,di                   ; Счетчик цикла задержки
```

```
        loop  $       ; Задержка
; Выключение звука
        and   al,11111101b      ; Выключение динамика
        out   port_b,al         ; Запись в системный порт B
        mov   cx,di             ; Счетчик цикла задержки
        loop  $                 ; Задержка
        dec   dx                ; Декремент счетчика колич. циклов
        jnz   ton01         ; Переход на начало нового периода
        sti             ; Разрешение прерываний
        ret             ; Выход из процедуры
ton0    endp            ; Конец универсальной процедуры
ton1    endp            ; Конец процедуры генерации звука 1
ton2    endp            ; Конец процедуры генерации звука 2
kbin    proc  near      ; Ввод с клавиатуры с ожиданием
        mov   ah,0      ; Функция 0
        int   16h       ; клавиатурного прерывания
        ret             ; Выход из процедуры
kbin    endp            ; Конец процедуры ввода с клавиатуры
code    ends            ; Конец сегмента (кодового)
        END   Start     ; Указание на точку входа
```

Задача 3.4.2.2. Написать программу, издающую различные звуки при нажатии на клавиши '1' и '2'. Для генерации звука следует выход канала 2 таймера. Выход из программы должен осуществляться по нажатию клавиши 'q'.

```
Assume  CS: Code, DS: Code
Code    SEGMENT
        org   100h
frequency1  equ   1000        ; Коэффиц. деления 1
frequency2  equ   3000        ; Коэффиц. деления 2
duration    equ   50000       ; Длительность
port_b      equ   61h         ; Адрес системного порта B
        .286
Start   proc  near          ; Основная процедура
        mov   ax,cs
        mov   ds,ax         ; DS = CS
beg1:   call  kbin ;                Опрос клавиатуры
        cmp   al,'1'        ; = '1' ?
        jnz   beg2          ; Нет
```

```
        call    ton1            ; Звук высоты 1
        jmp     beg1            ; Переход на начало цикла
beg2:   cmp     al,'2'          ; = '2' ?
        jnz     beg3            ; Нет
        call    ton2            ; Звук высоты 2
        jmp     beg1            ; Переход на начало цикла
beg3:   cmp     al,'q'          ; = 'q' ?
        jnz     beg1            ; Нет
        int     20h             ; Выход из программы
start   endp            ; Конец основной процедуры
ton2    proc    near    ; Процедура генерации звука 2
        mov     dx,duration  ; Длительность
        mov     di,frequency2 ; Коэффициент деления 2
        jmp     ton0            ; Переход на универсальную процедуру
ton1    proc    near            ; Процедура генерации звука 1
        mov     dx,duration  ; Длительность
        mov     di,frequency1 ; Коэффициент деления 2
; Универсальная процедура генерации звука
; DX – длительность, DI – коэффиц. деления
ton0    proc    near
        cli             ; Запрещение прерываний
; Включение динамика и таймера
        in      al,61h          ; Чтение состояния системного порта B
        or      al,3            ; Разрешение звучания (биты 0 и 1)
        out     61h,al          ; Запись в системный порт B
; Программирование делителя частоты 2 канала
        mov     ax,di           ; Делитель частоты
        out     42h,al          ; Мл.байт частоты □ канал 2 таймера
        xchg    al,ah           ; AH □ AL
        out     42h,al          ; Ст.байт частоты □ канал 2 таймера
; Формирование задержки
        mov     cx,dx           ; Счетчик цикла задержки
ton01:  push cx         ; Команды, используемые только для
        pop     cx      ; увеличения длит. цикла задержки
        loop    ton01 ; Задержка
; Выключение звука
        in      al,61h ; Чтение состояния системного порта B
```

```
        and   al,0fch ; Запрещение звучания (биты 0 и 1)
        out   61h,al  ; Запись в системный порт B
        sti           ; Разрешение прерываний
        ret           ; Выход из процедуры
ton0    endp          ; Конец универсальной процедуры
ton1    endp          ; Конец процедуры генерации звука 1
ton2    endp          ; Конец процедуры генерации звука 2
kbin    proc  near    ; Ввод с клавиатуры и проверка на выбор игры
; Процедура совпадает с одноименной в задаче 2.4.2.1.
kbin    endp
code    ends          ; Конец сегмента (кодового)
        END   Start   ; Указание на точку входа
```

Задача 3.4.2.3. Написать программу, издающую различные звуки при нажатии на клавиши '1' и '2'. Для генерации звука следует выход канала 2 таймера. Выход из программы должен осуществляться по нажатию клавиши 'q'. Использовать прерывание от таймера.

```
Assume  CS: Code, DS: Code
Code    SEGMENT
        org   100h
        .286
Start   proc  near   ; Основная процедура
        mov   ax,cs
        mov   ds,ax  ; DS = CS
        jmp   beg    ; "Перескок" через переменные
frequency       equ   0500h       ; Коэффициент деления
iniflag         db    0           ; Флаг звучания
old_int1c_off   dw    0           ; Смещение старого вектора
old_int1c_seg   dw    0           ; Сегмент старого вектора
beg:            ; Сохранение старого вектора прерывания 1Ch
        mov   ax,35h        ; Функция взятия вектора
        mov   al,1ch        ; Вектор 1Ch
        int   21h           ; Вызов функции DOS
        mov   cs:old_int1c_off,bx       ; Запись смещения
        mov   cs:old_int1c_seg,es       ; Запись сегмента
; Установка в вектор прерывания адреса новой
; программы обработки прерывания
        lea   dx,new_int1c      ; Запись нового вектора 1c
```

```
        mov   ah,25h              ; Функция установки вектора прерыв.
        mov   al,1ch              ; Номер вектора прерывания
        int   21h          ; DS:DX – адрес новой программы обр.
beg1: call    kbin         ; Опрос клавиатуры
        cmp   al,'1'       ; = '1' ?
        jnz   beg2         ; Нет
        mov   byte ptr iniflag,1   ; Взведение флага звуч.
        jmp   beg1         ; Переход на начало цикла
beg2: cmp   al,'2'         ; = '2' ?
        jnz   beg3         ; Нет
        mov   byte ptr iniflag,0   ; Сброс флага звуч.
        jmp   beg1         ; Переход на начало цикла
beg3: cmp   al,'q'         ; = 'q' ?
        jnz   beg1         ; Нет
; Восстановление старого вектора 1c и выход
        mov   dx,old_int1c_off   ; Смещение старого вектора
        mov   ax,old_int1c_seg   ; Сегмент старого вектора
        mov   ds,ax              ; DS:DX – адрес устанавл. вектора
        mov   ax,251ch           ; Установка старого вектора 1ch
        int   21h                ; Вызов функции DOS
        int   20h                ; Выход из программы
start   endp          ; Конец основной процедуры
; Новый обработчик прерывания 1ch
new_int1c   proc   far     ; Дальняя процедура
            pusha          ; Сохранение всех регистров (для Intel286)
            call   muz     ; Вызов процедура извлечения звука
            popa           ; Восстановление всех регистров
            iret           ; Возврат из программы обработки прерывания
new_int1c   endp           ; Конец нового обработчика прерывания 1ch
muz   proc   near   ; Процедура генерации звука
        test   byte ptr cs:iniflag,0ffh     ; Проверка флага
        jnz    muz1 ; Продолжение
        in     al,61h          ; Чтение системного порта B
        and    al,0fch         ; Запрещение звучания (биты 0 и 1)
        out    61h,al          ; Запись в системный порт B
        ret                    ; Выход, если флаг не взведен
muz1:                ; Программирование делителя частоты 2 канала
```

```
        mov   ax,frequency      ; Делитель частоты
        out   42h,al            ; Мл.байт частоты  канал 2 таймера
        xchg  al,ah             ; AH  AL
        out   42h,al            ; Ст.байт частоты  канал 2 таймера
; Разрешение звучания
        in    al,61h    ; Чтение системного порта B
        or    al,3      ; Разрешение звучания (биты 0 и 1)
        out   61h,al    ; Запись в системный порт B
        ret             ; Нормальный выход
muz     endp            ; Конец процедуры генерации звука
kbin    proc  near      ; Ввод с клавиатуры и проверка на выбор игры
; Процедура совпадает с одноименной в задаче 2.4.2.1.
kbin    endp
code    ends            ; Конец сегмента (кодового)
        END   Start     ; Указание на точку входа
```

Задача 3.4.2.4. Написать программу, исполняющую три различные мелодии при нажатии на клавиши '1', '2'и '3'. Для генерации звука следует использовать выход канала 2 таймера. Выход из программы должен осуществляться по нажатию клавиши 'q'. Использовать прерывание от таймера.

Для исполнения мелодии сначала формируется массив делителей частоты, соответствующих различным нотам звукоряда. Для нот используются номера от 1 до 48. Условное соответствие номеров нот и их значений показано на рисунке 3.3. Массив делителей частоты для нот называется NOTY. Значение 0 используется в качестве признака окончания мелодии. Значение 255 используется для обозначения паузы.

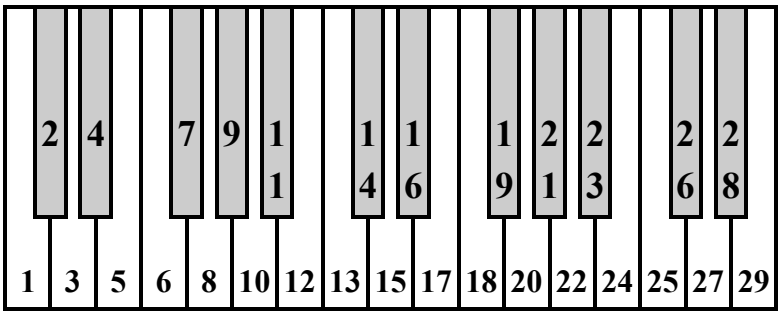


Рисунок 3.3 – Номера нот и их соответствие

В программе сформированы массивы для трех мелодий – "Чижик Пыжик" (mel1), "Подмосковные вечера" и «Кан-кан» (mel3). Длительность элементарного звука берется равной одному «тику» таймера (примерно 1/18 сек.). Для получения большей длительности в массиве мелодии записываются

подряд несколько одинаковых нот. Для получения четко выраженных соседних одинаковых нот используется пауза.

```
Assume  CS: Code, DS: Code
Code  SEGMENT
        org    100h
        .286
Start  proc   near   ; Основная процедура
        mov   ax,cs
        mov   ds,ax  ; DS = CS
        jmp   beg    ; «Перескок» через переменные
iniflag db    0      ; Флаг звучания
old_int1c_off       dw    0       ; Смещение старого вектора
old_int1c_seg       dw    0       ; Сегмент старого вектора
tek_mel             dw    ?       ; Адрес текущей ноты выбранной мелодии
beg:   mov   ax,351ch     ; Сохранение старого вектора 1c
        int    21h
        mov   cs:old_int1c_off,bx        ; Запись смещения
        mov   cs:old_int1c_seg,es        ; Запись сегмента
        lea    dx,new_int1c              ; Запись нового вектора 1c
        mov   ah,25h              ; Функция установки вектора прерыв.
        mov   al,1ch              ; Номер вектора прерывания
        int    21h                ; DS:DX – адрес новой программы обр.
beg1: call    kbin   ; Опрос клавиатуры
        cmp   al,'1'   ; = '1' ?
        jnz    beg2   ; Нет
        mov   byte ptr iniflag,1    ; Взведение флага звуч.
        lea    ax,mel1
        mov   tek_mel,ax
        jmp   beg1           ; Переход на начало цикла
beg2: cmp   al,'2'    ; = '2' ?
        jnz    beg3   ; Нет
        mov   byte ptr iniflag,1    ; Сброс флага звуч.
        lea    ax,mel2
        mov   tek_mel,ax
        jmp   beg1           ; Переход на начало цикла
beg3: cmp   al,'3'   ; = '3' ?
        jnz    beg4   ; Нет
```

```
        mov   byte ptr iniflag,1    ; Сброс флага звуч.
        lea   ax,mel3
        mov   tek_mel,ax
        jmp   beg1          ; Переход на начало цикла
beg4: cmp   al,'q'   ; = 'q' ?
        jnz   beg1   ; Нет
; Восстановление старого вектора 1c и выход
        mov   dx,old_int1c_off   ; Смещение старого вектора
        mov   ax,old_int1c_seg   ; Сегмент старого вектора
        mov   ds,ax              ; DS:DX – адрес устанавл. вектора
        mov   ax,251ch           ; Установка старого вектора 1ch
        int   21h
        int   20h
start   endp
; Новый обработчик прерывания 1ch
new_int1c   proc   far     ; Дальняя процедура
            pusha          ; Сохранение всех регистров
            call   muz     ; Вызов процедура извлечения звука
            popa           ; Восстановление всех регистров
            iret           ; Возврат из программы обработки прерывания
new_int1c   endp
; Процедура извлечения очередного звука
; tek_mel – адрес текущей ноты выбранной мелодии
muz   proc   near
        test   byte ptr cs:iniflag,0ffh    ; Проверка флага
        jnz    muz1                ; Продолжение
muze: in       al,61h       ; Чтение состояния системного порта B
        and    al,0fch      ; Запрещение звучания (биты 0 и 1)
        out    61h,al       ; Запись в системный порт B
        ret                 ; Выход, если флаг не взведен
muz1: mov   si,cs:tek_mel       ; Адрес текущей ноты
        mov   bl,cs:[si]    ; Текущая нота
        cmp   bl,255        ; Пауза ?
        jnz   muz2
; Выключение звука
        in    al,61h        ; Чтение состояния системного порта B
        and   al,0fch       ; Запрещение звучания (биты 0 и 1)
```

```
        out     61h,al; Запись в системный порт B
        inc     cs:tek_mel   ; Переход к адресу след. ноты
        ret
muz2: or      bl,bl          ; = 0 ?
        jnz     muz3 ; Продолжение
        jmp     muze ; Выход, если признак конца
muz3: shl     bl,1           ; Умножение bl на 2
        xor     bh,bh ; bh = 0
        mov   ax,cs:noty[bx]       ; B DI частота ноты
; Программирование делителя частоты 2 канала
        inc     cs:tek_mel   ; Переход к адресу след. ноты
        out     42h,al         ; Мл.байт частоты □ канал 2 таймера
        xchg   al,ah ; AH □ AL
        out     42h,al         ; Ст.байт частоты□□ канал 2 таймера
; Разрешение звучания
        in      al,61h         ; Чтение состояния системного порта B
        or      al,3           ; Разрешение звучания (биты 0 и 1)
        out     61h,al         ; Запись в системный порт B
        ret                    ; Нормальный выход
muz   endp
kbin   proc   near ; Ввод с клавиатуры и проверка на выбор игры
; Процедура совпадает с одноименной в задаче 2.4.2.1.
kbin   endp
; Мелодия «Чижик Пыжик»
mel1  db      17,17,255,13,13,255,17,17,255,13,13,255,18,18,255
        db      17,17,255,15,15,15,15,255,255
        db      8,8,255,8,8,255,8,8,255,10,255,12,255
        db      13,13,255,13,13,255,13,13,13,13
        db      0
; Мелодия «Подмосковные вечера»
mel2  db      1,1,1,4,4,4,8,8,8,4,4,4,6,6,6,6,6,6,4,4,4,3,3,3
        db      8,8,8,8,8,8,6,6,6,6,6,6,1,1,1,1,1,1,1,1,1,1,1
        db       0
; Мелодия «Кан-кан»
mel3  db      18,6,25,13,22,18,25,13,20,1,23,8,22,5,20,1
        db       25,6,13,13,25,10,13,13,25,10,27,13,22,10,23,13,20,1
        db      11,11,20,5,11,11,20,1,23,11,22,5,20,11,18,6,30,18
```

```
        db      29,17,27,15,25,13,23,11,22,10,20,8,18,6,13,13,18,10
        db      13,13,20,1,23,8,22,5,20,8,25,6,13,13,25,10,13,13
        db      25,6,27,13,22,10,23,13,20,1,8,8,20,5,8,8,20,1,23,8
        db      22,5,20,8,18,6,25,10,20,13,22,10,18,6,6,6,6,6,6,6
        db      34,8,24,12,24,15,34,12,32,1,25,5,25,8,29,5,30,6,34,13
        db      37,10,34,13,34,1,32,8,32,5,8,8,34,8,24,12,24,15,34,12
        db      32,1,25,5,25,8,29,5,29,3,27,7,29,10,27,13,34,12,32,8
        db      34,6,32,3,34,8,24,15,24,12,34,15,32,1,29,8,25,5,29,8
        db      30,6,34,13,37,10,34,13,34,1,32,5,32,8,5,5,34,8,24,15
        db      24,12,34,15,32,1,25,6,25,5,29,8,29,3,27,7,29,10,27,7
        db      32,8,30,6,29,5,27,3,25,1,8,8,25,5,8,8,27,12,30,15
        db      29,8,27,12,32,1,8,8,32,5,8,8,32,1,34,8,29,5,30,8,27,8
        db      15,15,27,12,15,15,27,8,30,12,29,15,27,12,25,1,37,1
        db      36,5,34,6,32,8,30,8,29,10,27,12,25,1,8,8,25,5,8,8
        db      27,8,30,15,29,12,27,15,32,1,8,8,32,5,8,8,32,1,34,8
        db      29,5,30,8,27,8,15,15,27,12,15,15,27,8,30,15,29,12
        db      27,15,25,1,32,8,27,5,29,8,25,1,32,32,37,37,0
; Коэффициенты деления для нот
noty    dw      0eeeh,0e18h,0d49h,0c8eh,0bdfh,0b2fh,0abeh
        dw 9f7h,968h,8e0h,861h,7e8h,777h,70ch,6a5h,647h
        dw 5edh,597h,547h,4fbh,4b4h,470h,430h,3f4h
        dw 3bbh,386h,352h,323h,2f6h,2cbh,2a3h,27dh,25ah,238h,218h,1fah
        dw 1ddh,1c3h,1a9h,192h,17bh,166h,152h,13fh,12dh,11ch,10ch,0fdh
        dw 0
code    ends
        END  Start
```

Задача 3.4.2.5. Модифицировать программу из предыдущей задачи так, чтобы длительность каждого звука задавалась в массиве мелодии. Для каждого звука должна записываться пара значений: номер ноты и длительность, выраженная в элементарных «тиках» таймера. Для простоты в программе будут реализованы только первые две короткие мелодии.

```
Assume  CS: Code, DS: Code
Code SEGMENT
        org     100h
        .286
Start   proc    near
        mov     ax,cs
```

```
        mov   ds,ax
        jmp   beg
pausa               equ   255
frequency           dw    1000h
iniflag             db    0             ; Флаг звучания
old_int1c_off       dw    0       ; Смещение старого вектора
old_int1c_seg       dw    0       ; Сегмент старого вектора
tek_mel             dw    ?
duration            db    0       ; Длительность текущей ноты
beg:  mov   ax,351ch    ; Сохранение старого вектора 1c
        int   21h
        mov   cs:old_int1c_off,bx       ; Запись смещения
        mov   cs:old_int1c_seg,es       ; Запись сегмента
        lea   dx,new_int1c              ; Запись нового вектора 1c
        mov   ah,25h              ; Функция установки вектора прерыв.
        mov   al,1ch              ; Номер вектора прерывания
        int   21h           ; DS:DX – адрес новой программы обр.
beg1: call  kbin            ; Опрос клавиатуры
        mov   byte ptr duration,1
        cmp   al,'1'        ; = '1' ?
        jnz   beg2          ; Нет
        mov   byte ptr iniflag,1    ; Взведение флага звуч.
        lea   ax,mel1
        mov   tek_mel,ax
        jmp   beg1                  ; Переход на начало цикла
beg2: cmp   al,'2'          ; = '2' ?
        jnz   beg3          ; Нет
        mov   byte ptr iniflag,1    ; Сброс флага звуч.
        lea   ax,mel2
        mov   tek_mel,ax
        jmp   beg1          ; Переход на начало цикла
beg3: cmp   al,'q'          ; = 'q' ?
        jnz   beg1          ; Нет
; Восстановление старого вектора 1c и выход
        mov   dx,old_int1c_off  ; Смещение старого вектора
        mov   ax,old_int1c_seg  ; Сегмент старого вектора
        mov   ds,ax             ; DS:DX – адрес устанавл. вектора
```

```
        mov   ax,251ch          ; Установка старого вектора 1ch
        int   21h
        int   20h
start   endp
; Новый обработчик прерывания 1ch
new_int1c  proc  far
           pusha
           dec   byte ptr cs:duration
           jnz    ex
           call   muz  ; Вызов процедуры извлечения звука
ex:        popa
           iret
new_int1c  endp
muz   proc  near
      test  byte ptr cs:iniflag,0ffh   ; Проверка флага
      jnz   muz1                ; Продолжение
muze: in    al,61h        ; Чтение состояния системного порта B
      and   al,0fch       ; Запрещение звучания (биты 0 и 1)
      out   61h,al        ; Запись в системный порт B
      ret                 ; Выход, если флаг не взведен
muz1: mov  si,cs:tek_mel        ; Адрес текущей ноты
      mov  bx,word ptr cs:[si]      ;  BL  -текущая  нота,  BH  –
длительность
      mov  cs:duration,bh     ; Длит. в системную переменную
      cmp  bl,255       ; Пауза ?
      jnz  muz2
; Выключение звука
      in    al,61h        ; Чтение состояния системного порта B
      and   al,0fch       ; Запрещение звучания (биты 0 и 1)
      out   61h,al        ; Запись в системный порт B
      inc   cs:tek_mel  ; Переход к адресу след. ноты
      inc   cs:tek_mel  ; след. длительности
      ret
muz2: or    bl,bl         ; = 0 ?
      jnz   muz3 ; Продолжение
      jmp   muze ; Выход, если признак конца
muz3: shl   bl,1          ; Умножение bl на 2
```

```
        xor     bh,bh ; bh = 0
        mov     ax,cs:noty[bx]      ; В DI частота ноты
; Программирование делителя частоты 2 канала
        inc     cs:tek_mel  ; Переход к адресу след. ноты
        inc     cs:tek_mel ; и след. длительности
        out     42h,al      ; Мл.байт частоты□□ канал 2 таймера
        xchg    al,ah       ; AH□□ AL
        out     42h,al      ; Ст.байт частоты □ канал 2 таймера
; Разрешение звучания
        in      al,61h      ; Чтение состояния системного порта B
        or      al,3        ; Разрешение звучания (биты 0 и 1)
        out     61h,al      ; Запись в системный порт B
        ret                 ; Нормальный выход
muz     endp
kbin    proc    near        ; Ввод с клавиатуры
; Процедура совпадает с одноименной в задаче 2.4.2.1.
kbin    endp
mel1    db      17,2,255,1,13,2,255,1,17,2,255,1,13,2,255,1,18,2,255,1, 17,2
        db      255,1,15,4,255,2, 8,2,255,1,8,2,255,1,8,2,255,1,10,1,255,1
        db      12,1,255,1,13,2,255,1,13,2,255,1,13,4,0
mel2    db      1,4,4,4,8,4,4,4,6,8,4,4,3,4,8,8,6,8,1,12,0
noty    dw      0eeeh,0e18h,0d49h,0c8eh,0bdfh,0b2fh,0abeh, 9f7h,968h
        dw      8e0h,861h,7e8h,777h,70ch,6a5h,647h, 5edh,597h,547h
        dw      4fbh,4b4h,470h,430h,3f4h, 3bbh,386h,352h,323h,2f6h,2cbh
        dw      2a3h,27dh,25ah,238h,218h,1fah, 1ddh,1c3h,1a9h,192h,17bh
        dw      166h,152h,13fh,12dh,11ch,10ch,0fdh,0
code    ends
        END  Start
```

3.5 Вывод динамических изображений

Написать программу, которая в графическом режиме CGA 4 * 320 * 200 рисует фигурку колобка размером 16 * 16 точек непосредственно в экранной области и перемещает его при нажатии курсорных клавиш. Коды управления курсором: Вверх – 48h, Вниз – 50h, Вправо – 4Dh, Влево – 4Bh.

Структура видеопамяти в режиме CGA: начинается с адреса B000h, четные линии имеют смещение 0000h – 1F3Fh (около 8 КВ), нечетные 2000h – 3F3Fh (около 8 КВ) (промежуток между ними не используется).

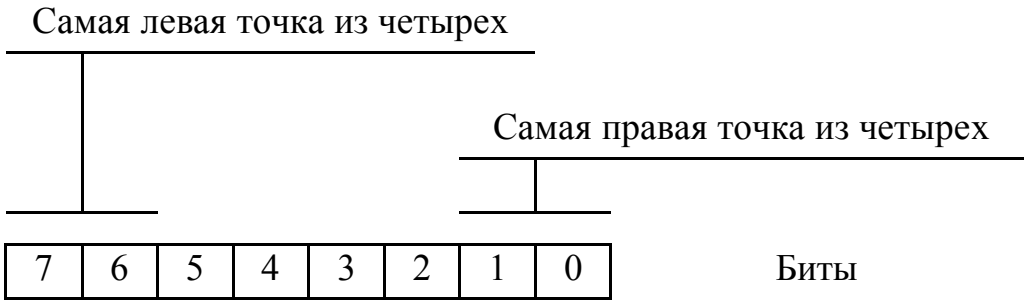


Рисунок 3.4 – Графические точки на экране

Цвет 00 – черный

01 – голубой (зеленый)

10 – сиреневый (красный)

11 – белый (желтый)

```
Assume CS: Code, DS: Code
    Code  SEGMENT
    org   100h
    Start: jmp   start1
    addr  dw    0       ; Адрес начальной точки спрайта
    addr0       dw    0       ; Старый адрес
    ; 16 * 16
    colob db    0,5,50h,0
          db    0,5fh,0f5h,0
    db    5,0ffh,0ffh,50h
    db    3fh,0ffh,0ffh,0fch
    db    0ffh,0afh,0fah,0ffh
    db    0ffh,0abh,0eah,0ffh
    db    0ffh,8bh,0e2h,0ffh
    db    0ffh,0ffh,0ffh,0ffh
    db    3fh,0beh,0beh,0fch
    db    0fh,0efh,0fbh,0f0h
    db    0,0fah,0afh,0
    db    0,3fh,0fch,0
    db    0,3ch,3ch,0
    db    15h,3ch,3ch,54h
    db    55h,54h,15h,55h
    db    55h,54h,15h,55h
    start1:                   ; Установка видеорежима
    mov   ah,0          ; Функция установки видеорежима
    mov   al,4          ; Граф. реж. CGA 320 * 200 точек
    int   10h
```

```
mov   di,addr
st2:  lea   si,colob
call  sprit0
call  sprite
; Сохранение исходных координат
mov   ax,addr
mov   addr0,ax
; Чтение клавиатуры
call  kbin
cmp   ah,1          ; Скан-код = Esc ?
jnz   st3           ; Нет
int   20h           ; Выход при нажатии Esc
st3:  cmp   ah,48h      ; Вверх ?
jnz   st4
sub   addr,80*2   ; Вверх на 4 строки
jmp   st2
st4:  cmp   ah,50h      ; Вниз ?
jnz   st5
add   addr,80*2   ; Вниз на 4 строки
jmp   st2
st5:  cmp   ah,4dh      ; Вправо ?
jnz   st6
inc   addr
jmp   st2
st6:  cmp   ah,4bh      ; Влево ?
jnz   st2
dec   addr
jmp   st2
; Ввод с клавиатуры
kbin  proc  near
mov   ah,0          ; Функция 0
int   16h           ; клавиатурного прерывания
ret
kbin  endp
; Построение спрайта 16 * 16
; addr – адрес левого верхнего угла
; SI – начало спрайта
```

```
sprite proc   near
; Построение четных строк
push  si
mov   di,addr
mov   ax,0b800h
mov   es,ax
mov   bp,8          ; Счетчик числа строк/2
sp1:  mov   cx,4          ; Счетчик числа слов
rep   movsb
add   di,76
add   si,4
dec   bp
jnz   sp1
; Построение нечетных строк
mov   di,addr
pop   si
add   si,4          ; Переход к нечетной строке
mov   ax,0ba00h
mov   es,ax
mov   bp,8          ; Счетчик числа строк/2
sp2:  mov   cx,4          ; Счетчик числа слов
rep   movsb
add   di,76
add   si,4
dec   bp
jnz   sp2
ret
sprite endp
; Стирание спрайта 16 * 16
; addr0 – координаты левого верхнего угла
sprit0        proc   near
; Стирание четных строк
mov   di,addr0
mov   ax,0b800h
mov   es,ax
mov   bp,8          ; Счетчик числа строк/2
xor   al,al
```

```
sp3:  mov  cx,4          ; Счетчик числа слов
rep   stosb
add   di,76
dec   bp
jnz   sp3
; Стирание нечетных строк
mov   di,addr0
mov   ax,0ba00h
mov   es,ax
mov   bp,8          ; Счетчик числа строк/2
sp4:  mov  cx,4          ; Счетчик числа слов
rep   stosb
add   di,76
dec   bp
jnz   sp4
ret
sprit0      endp
code  ends
END   Start          ; Указание точки входа в программу
```

Изменить палитру на красный-зеленый-желтый. При движении менять проекции фигурки. Палитра переключается в режиме 4:

int 10h, функция 0Bh

BH=1, BL=палитра (0, 1)

```
Assume CS: Code;, DS: Code
Code  SEGMENT
org   100h
Start: jmp   start1
addr  dw    0            ; Адрес начальной точки спрайта
addr0 dw    0            ; Старый адрес
; 16 * 16
colf  db    0,5,50h,0
      db    0,5fh,0f5h,0
db    5,0ffh,0ffh,50h
db    3fh,0ffh,0ffh,0fch
db    0ffh,0afh,0fah,0ffh
db    0ffh,0abh,0eah,0ffh
db    0ffh,8bh,0e2h,0ffh
```

```
db      0ffh,0ffh,0ffh,0ffh
db      3fh,0beh,0beh,0fch
db      0fh,0efh,0fbh,0f0h
db      0,0fah,0afh,0
db      0,3fh,0fch,0
db      0,3ch,3ch,0
db      15h,3ch,3ch,54h
db      55h,54h,15h,55h
db      55h,54h,15h,55h
colb    db      0,5,50h,0
db      0,55h,55h,0
        db      5,55h,55h,50h
        db      15h,55h,55h,54h
db      0d5h,55h,55h,57h
        db      0d5h,55h,55h,57h
        db      0f5h,55h,55h,5fh
        db      0fdh,55h,55h,7fh
        db      3dh,55h,55h,7ch
        db      0fh,55h,55h,0f0h
        db      0,0ffh,0ffh,0
        db      0,3fh,0fch,0
        db      0,3ch,3ch,0
        db      15h,3ch,3ch,54h
        db      55h,54h,15h,55h
        db      55h,54h,15h,55h
coll    db      0,5,50h,0
        db      0,5fh,55h,50h
        db      5,7fh,55h,50h
        db      1,0ffh,0fdh,54h
        db      2,0ffh,0ffh,55h
        db      2,0bfh,0ffh,55h
        db      0fch,0bfh,0ffh,0d5h
        db      0ffh,0ffh,0ffh,0f5h
        db      0fh,0ffh,0bfh,0fch
        db      3,0feh,0ffh,0f0h
        db      0,0abh,0ffh,0
        db      0,3fh,0fch,0
```

```
        db      0,0fh,0fch,0
        db      15h,0fh,0fch,0
        db      55h,55h,55h,0
        db      55h,55h,55h,0
colr    db      0,5,50h,0
        db      0,55h,0f5h,0
        db      5,55h,0fdh,50h
        db      15h,7fh,0ffh,40h
        db      55h,0ffh,0ffh,80h
        db      55h,0ffh,0feh,80h
        db      57h,0ffh,0feh,3fh
        db      3fh,0ffh,0ffh,0ffh
        db      3fh,0feh,0ffh,0f0h
        db      0fh,0ffh,0bfh,0c0h
        db      0,0ffh,0eah,0
        db      0,3fh,0fch,0
        db      0,3fh,0f0h,0
        db      0,3fh,0f0h,54h
db      0,55h,55h,55h
        db      0,55h,55h,55h
start1:                         ; Установка видеорежима
mov     ah,0            ; Функция установки видеорежима
mov     al,5            ; Граф. реж. CGA 320 * 200 точек
        int     10h
        mov     ah,0bh          ; Установка палитры
        mov     bx,102h
        int     10h
        mov     di,addr
        lea     si,colf
st2:    call    sprit0
call    sprite
; Сохранение исходных координат
st0:    mov     ax,addr
mov     addr0,ax
; Чтение клавиатуры
call    kbin
cmp     ah,1    ; Скан-код = Esc ?
```

```
jnz     st3     ; Нет
int     20h     ; Выход при нажатии Esc
st3:    cmp   ah,48h        ; Вверх ?
jnz     st4
sub     addr,80*2    ; Вверх на 4 строки
lea     si,colb
jmp     st2
st4:    cmp   ah,50h        ; Вниз ?
jnz     st5
add     addr,80*2    ; Вниз на 4 строки
lea     si,colf
jmp     st2
st5:    cmp   ah,4dh        ; Вправо ?
jnz     st6
inc     addr
lea     si,colr
jmp     st2
st6:    cmp   ah,4bh        ; Влево ?
jnz     st0
dec     addr
lea     si,coll
jmp     st2
; Ввод с клавиатуры
kbin   proc   near
mov   ah,0          ; Функция 0
int     16h         ; клавиатурного прерывания
ret
kbin   endp
; Построение спрайта 16 * 16
; addr – адрес левого верхнего угла
; SI – начало спрайта
sprite proc   near
; Построение четных строк
push   si
mov   di,addr
mov   ax,0b800h
mov   es,ax
```

```
mov   bp,8          ; Счетчик числа строк/2
sp1:  mov   cx,4          ; Счетчик числа слов
rep   movsb
add   di,76
add   si,4
dec   bp
jnz   sp1
; Построение нечетных строк
mov   di,addr
pop   si
add   si,4          ; Переход к нечетной строке
mov   ax,0ba00h
mov   es,ax
mov   bp,8          ; Счетчик числа строк/2
sp2:  mov   cx,4          ; Счетчик числа слов
rep   movsb
add   di,76
add   si,4
dec   bp
jnz   sp2
ret
sprite endp
; Стирание спрайта 16 * 16
; addr0 – координаты левого верхнего угла
sprit0        proc   near
; Стирание четных строк
mov   di,addr0
mov   ax,0b800h
mov   es,ax
mov   bp,8          ; Счетчик числа строк/2
xor   al,al
sp3:  mov   cx,4          ; Счетчик числа слов
rep   stosb
add   di,76
dec   bp
jnz   sp3
; Стирание нечетных строк
```

```
mov   di,addr0
mov   ax,0ba00h
mov   es,ax
mov   bp,8          ; Счетчик числа строк/2
sp4:  mov   cx,4          ; Счетчик числа слов
rep   stosb
add   di,76
dec   bp
jnz   sp4
ret
sprit0      endp
code  ends
END   Start         ; Указание точки входа в программу
```

3.6 Работа с жестким диском

Написать программу, которая при первом запуске (инициализации) определяет начальный кластер своего расположения на диске, записывает в файл на диск этот номер и выполняет свою основную функцию – выводит сообщение на экран, что все в порядке. При повторном и последующих запусках программа должна проверять, соответствует ли записанный в ней при инициализации номер начального кластера действительному номеру. При совпадении программа должны выполнить свою основную функцию – вывести сообщение о том, что все в порядке. При несовпадении записанного в программе и действительного номеров начального кластера программа должна вывести на экран сообщение о нежелании работать.

```
Assume CS: Code;, DS: Code
Code  SEGMENT
bpb   struc  ; Структура блока параметров BIOS
sect_siz    dw    ?     ; Размер сектора в байтах
clus_siz    db    ?     ; Секторов в кластере
res_sect    dw    ?     ; Зарезервированных секторов
fat_num     db    ?     ; Кол-во FAT на диске
root_siz    dw    ?     ; Размер каталога (кол-во файлов)
num_sect    dw    ?     ; Общее количество секторов
med_desc    db    ?     ; Дескриптор носителя
fat_siz     dw    ?     ; Число секторов в FAT
sec_trac    dw    ?     ; Число секторов на дорожке
```

```
num_had     dw    ?       ; Число головок
hidd_sec    dw    ?       ; Число скрытых секторов
bpb    ends
bpd    struc  ; Структура блока параметров диска
sec_num     dd    ?       ; 32-битный номер сектора
number_s    dw    ?       ; Количество читаемых секторов
b_off       dw    ?       ; Смещение буфера
b_seg       dw    ?       ; Сегмент буфера
bpd    ends
cat struc      ; Структура записи каталога
f_name      db    11 dup (?)   ; Имя файла с расширением
f_attr      db    ?            ; Атрибуты
res_dos     db    10 dup (?)   ; Резерв
f_time      dw    ?            ; Время создания
f_date      dw    ?            ; Дата создания
b_clu       dw    ?            ; Начальный кластер
f_size            dd    ?              ; Размер файла
cat    ends
.286
org    100h
Start: jmp    st1
key         dw    ?       ; Начальный кластер
f_key             db    0       ; Флаг инициализации
err_read    db    'Ошибка чтения $'
ok          db    12 dup(0ah),0dh,25 dup(' ')
            db    'Все в порядке',12 dup(0ah),'$'
nok         db    12 dup(0ah),0dh,25 dup(' ')
            db    "I'm sorry",12 dup(0ah),'$'
fil_name    db    'LAB2.COM',0
f_num_s     dw    ?       ; Физ. номер сектора
sec_cyl     dw    ?       ; Число секторов на цилиндре
sec_cat     db    ?       ; Секторов в корневом каталоге
cyl         db    ?       ; Цилиндр
head        db    ?       ; Головка
beg_sec     dd    ?       ; Нач. сектор раздела (логич.)
num_sec     dw    ?       ; Кол. читаемых сект. каталога
point       dw    ?       ; Указатель в тек. каталоге
```

```
flag        db    ?       ; Флаг поиска
f_flag              db    ?       ; Флаг поиска файла
root_s      dw    ?       ; Лог. ном. сект. нач. корн. кат.
clu0        dw    ?       ; Лог. ном. первого доступн. кластера
handler     dw    ?       ; Дескриптор файла
buf_cat     db    100h dup(?),'$'     ; Буфер для тек. каталога
buf1        db    200h dup(?)               ; Буфер для MBR
buf2        db    200h dup(?)               ; Буфер для BR
            db    '$'
buf3        db    800h dup(?)               ; Буфер для корн.кат.
            db    '$'
b_param     db    10 dup(0)           ; Блок параметров для int 25h
st1:  mov  ax,cs
mov  ds,ax
; Определение начального кластера
lea    ax,buf_cat   ; Инициализация указателя на начало
mov  point,ax      ; буфера текущего каталога
; Получение текущего каталога в buf_cat
mov  ah,47h        ; Получение текущего каталога
xor    dl,dl       ; Текущий дисковод
lea    si,buf_cat   ; Буфер для ASCIIZ тек. каталога
int    21h         ; Тек' каталог в buf_cat
jnc    st2         ; Ошибки нет
; Ошибка чтения сектора
lea    dx,err_read
mov  al,1          ; Ошибка чтения 1
call   msg
int    20h
st2:                       ; Чтение BR в buf2
mov  al,2          ; Диск C
mov  cx,-1         ; Признак > 32 M
lea    bx,b_param ; Блок параметров
mov  word ptr [bx].sec_num,0              ; Отн. номер сектора
mov  word ptr [bx+2].sec_num,0            ; Отн. номер сектора
mov  [bx].number_s,1    ; Кол-во читаемых секторов
mov  [bx].b_off,offset buf2
mov  dx,cs
```

```
mov   [bx].b_seg,dx
int   25h          ; BR в buf2
pop   cx
jnc   st3          ; Ошибки нет
; Ошибка чтения сектора
lea   dx,err_read
mov   al,2         ; Второе чтение сектора
call  msg
int   20h
; Чтение корневого каталога в buf3
st3:  lea   si,[buf2+11]
; Определение количества читаемых секторов каталога
; (один кластер, но не более 4)
mov   al,[si].clus_siz     ; Размер кластера
cmp   al,4                 ; Разм. кластера < 4
jc    st31
mov   al,4
st31: xor   ah,ah
mov   num_sec,ax           ; Кол-во читаемых секторов кат.
; Определение числа секторов в корневом каталоге
mov   ax,[si].root_siz     ; Размер кат. (кол.файлов)
mov   cl,4
shr   ax,cl        ; /16 (записей в секторе)
mov   sec_cat,al   ; Секторов в каталоге
; Определение физ. номера сектора начала корневого каталога
mov   dl,[si].fat_num ; Кол-во FAT на диске
mov   ax,[si].fat_siz ; Число секторов в FAT
mul   dl           ; AX -номер сект. нач. корнев. кат.
inc   ax           ; Лог. номер сектора нач. корн. кат.
mov   root_s,ax    ; Лог. ном. сект. нач. корн. кат.
lea   bx,b_param ; Блок параметров
mov   word ptr [bx].sec_num,ax       ; Отн. номер сект.
mov   word ptr [bx+2].sec_num,0      ; Отн. номер сект.
mov   al,2         ; Диск C
mov   cx,-1        ; Признак > 32 M
mov   di,num_sec
mov   [bx].number_s,di  ; Кол-во читаемых секторов
```

```
mov   [bx].b_off,offset buf3
mov   dx,cs
mov   [bx].b_seg,dx
int 25h              ; Каталог в buf3
pop   cx
jnc   st4            ; Ошибки нет
; Ошибка чтения сектора
lea   dx,err_read
mov   al,3
call  msg
int   20h
st4:                 ; Определение логического номера сектора первого
                     ; доступного кластера на текущем логическом диске
mov   bp,root_s      ; Лог.ном.сект.нач.корн.кат.
mov   bl,sec_cat
xor   bh,bh
add   bp,bx          ; + сект. в корн. каталоге
mov   clu0,bp
mov   byte ptr f_flag,0    ; Ищется не файл
; Цикл поиска каталогов вплоть до текущего
st40: mov   cx,64          ; Кол. анализируемых записей
lea   si,buf3        ; Начало 1 записи каталога
st41: cmp   byte ptr [si],0 ; Пустая запись ?
jz    st42           ; Пропуск
cmp   byte ptr [si],0e5h   ; Стертая запись ?
jz    st42           ; Пропуск
call  comp           ; Сравнение с искомым
cmp   flag,1               ; Найден ?
jz    st43           ; Найден
st42: add   si,32          ; Переход к следующей записи
loop  st41
; Ошибка поиска
lea   dx,err_read
mov   al,4
call  msg
int   20h
st43: test   byte ptr [di],0ffh   ; Конец имени каталога ?
```

```
jz st5                          ; Нашли !
inc   di              ; Переход к следующему имени кат.
mov   point,di
call  read_cat        ; Чт.найденного кат. в buf3
jmp   st40
st5:  call  read_cat  ; Чт.найденного кат. в buf3
; Поиск защищаемого файла
mov   byte ptr f_flag,1     ; Ищется файл
lea   ax,fil_name           ; Защищаемый файл
mov   point,ax
mov   cx,64                 ; Кол. анализируемых записей
lea   si,buf3               ; Начало 1 записи каталога
st51: cmp   byte ptr [si],0       ; Пустая запись ?
jz    st52                  ; Пропуск
cmp   byte ptr [si],0e5h    ; Стертая запись ?
jz    st52                  ; Пропуск
call  comp                  ; Сравнение с искомым
cmp   flag,1                      ; Найден ?
jz    st53                  ; Найден
st52: add   si,32                 ; Переход к следующей записи
loop  st51
; Ошибка чтения
lea   dx,err_read
mov   al,6
call  msg
int   20h
st53: mov   ax,[si].b_clu
; В AX начальный кластер этого файла
; Проверка флага инициализации
test  f_key,0ffh    ; Проверка флага инициализации
jz    st54          ; Переход, если инициализации не было
jmp   nstart        ; Переход на основную программу
; Инициализация
st54: mov   key,ax        ; Запись начального кластера
mov   f_key,1       ; Установка флага инициализации
; Открытие файла
mov   ax,3d01h      ; Открытие файла для записи
```

```
lea     dx,fil_name ; Имя файла
int     21h
jnc     nst1
; Ошибка чтения
lea     dx,err_read
mov   al,7
call    msg
int     20h
nst1:  mov   handler,ax
; Установка указателя файла на 3
mov   ax,4200h     ; Абс. установка от начала
mov   bx,handler   ; Дескриптор файла
xor     cx,cx      ; Старшая часть смещения
mov   dx,3         ; Младшая часть смещения
int     21h
; Запись в файл
mov   ah,40h       ; Запись
mov   bx,handler   ; Дескриптор файла
mov   cx,3         ; Количество записываемых байтов
lea     dx,key     ; Смещение записываемых байтов
int     21h
jnc     nst2
; Ошибка чтения
lea     dx,err_read
mov   al,8
call    msg
int     20h
nst2:                      ; Сообщение основной программы
lea     dx,ok
mov   al,0
call    msg
call    kbin       ; Ожидание
int     20h
nstart: cmp  ax,key        ; Сравн. истинного и зап. номеров
jz      nst2       ; Все в порядке
; Сообщение основной программы
lea     dx,nok
```

```
mov   al,0
call    msg
call    kbin          ; Ожидание
int     20h
; Чтение подкаталога в buf3
; Вход: SI = начало записи этого подкаталога в вышестоящем каталоге
read_cat      proc   near
mov   bp,clu0
; BP = лог.номер сект.первого доступного кластера
lea     bx,buf2               ; BR
mov   cl,[bx+11].clus_siz         ; Секторов в кластере
xor     ch,ch
mov   ax,[si].b_clu               ; Нач. кластер найд. кат.
sub     ax,2          ; Кластеры начинаются с 2
mul     cx            ; DX:AX нач.сект.найд.кат.отн.конца корн.
add     ax,bp
adc     dx,0          ; DX:AX лог.нач.сект.найд.кат.
lea     bx,b_param ; Блок параметров
mov   word ptr [bx].sec_num,ax                ; Отн. номер сект.
mov   word ptr [bx+2].sec_num,dx     ; Отн. номер сект.
mov   al,2            ; Диск C
mov   cx,-1           ; Признак > 32 M
mov   di,num_sec
mov   [bx].number_s,di   ; Кол-во читаемых секторов
mov   [bx].b_off,offset buf3
mov   dx,cs
mov   [bx].b_seg,dx
int     25h           ; Каталог в buf3
pop     cx
jnc     rc1     ; Ошибки нет
; Ошибка чтения сектора
lea     dx,err_read
mov   al,5
call    msg
int     20h
rc1:    ret
read_cat      endp
```

```
; Сравнение тек. записи каталога с искомой (point)
; При успешном поиске: flag = 1
; SI – нач. найденной записи
comp proc   near
mov   flag,0              ; Сброс флага поиска
test    byte ptr f_flag,0ffh        ; Ищется файл ?
jnz     comp1             ; Да
test    byte ptr [si].f_attr,10h    ; Подкаталог ?
jnz     comp1                       ; Подкаталог
ret                 ; Выход, если не подкаталог
comp1: mov          bp,si           ; Нач. тек. имени в каталоге
mov   di,point      ; Указ. тек. кат.
push  cx
mov   cx,11
comp2: mov          al,byte ptr [di]      ; Симв. тек. кат.
cmp   al,'.'                ; Разделитель ?
jnz     comp3               ; Не разделитель
inc     di
jmp     comp2               ; Пропуск разделительной точки
comp3: cmpal,'\'                    ; Конец имени ?
jz      comp31              ; Конец
cmp   al,0                  ; Конец всего ?
jnz     comp4               ; Не конец
; Имя каталога совпало (имя короче 11 симв.)
comp31: mov   flag,1              ; Взведение флага поиска
   pop    cx
   ret                 ; Выход при успешном поиске
comp4: cmp  byte ptr cs:[bp],20h ; Пробел ?
jnz     comp5               ; Не пробел
inc     bp
dec     cx
jmp     comp4               ; Пропуск пробела
comp5: cmp          al,cs:[bp]
jz      comp6               ; Символы совпали
; Символы не совпали
pop     cx
ret                 ; Выход при несовпадении имен
```

```
comp6: inc  di              ; Адрес в тек. каталоге
inc    bp             ; Адрес в тек. записи каталога
loop   comp2
; Имя каталога совпало (имя 11 симв.)
mov    flag,1               ; Взведение флага поиска
pop    cx
ret                 ; Выход при успешном поиске
comp endp
; Печать каталога
pri_cat proc near
mov    cx,64        ; Кол. выводимых записей
lea    si,buf3      ; Начало 1 записи каталога
pri_cat1: cmp   byte ptr [si],0    ; Пустая запись ?
jz     pri_cat2             ; Пропуск
cmp    byte ptr [si],0e5h   ; Стертая запись ?
jz     pri_cat2             ; Пропустить печать
call   c_file               ; Печать строки
pri_cat2: add  si,32                ; Переход к следующей записи
loop   pri_cat1
       ret
pri_cat  endp
c_file         proc   near
call   wk
pusha
; Печать имени файла
push   si
mov    cx,11
c_fil1: mov  al,[si]
call   print
inc    si
loop   c_fil1
call   sp4
; Печать начального кластера
pop    si
mov    ax,[si].b_clu
call   prax
call   sp4
```

```
; Печать размера файла
mov   ax,word ptr [si+2].f_size
call    prax
mov   ax,word ptr [si].f_size
call    prax
popa
ret
c_file          endp
; Вывод строки
msg   proc   near
push  ax
mov   ah,9
int     21h
pop   ax
or      al,al
jnz     msg1
ret
msg1: call    prali
ret
msg   endp
; Ввод с клавиатуры
kbin   proc   near
mov   ah,0            ; Функция 0
int     16h           ; клавиатурного прерывания
ret
kbin   endp
; Печать 16 HEX байтов из [SI]
hex    proc   near
pusha
mov   cx,16
hex1: mov   dl,[si]
call    pral
call    space ; Печать пробела
inc     si
loop   hex1
call    wk            ; Перевод строки, возврат каретки
popa
```

```
ret
hex    endp
; Печать AX и ВК
praxi  proc   near
pusha
call   prax
call   wk
popa
ret
praxi  endp
; Печать AL и ВК
prali  proc   near
pusha
call   pral
call   wk
popa
ret
prali  endp
; Печать четырехзначного шестнадцатеричного числа из AX
prax   proc   near
push   bx
push   cx
push   ax
and    ax,0f000h
mov    cl,12
       shr    ax,cl
       call   prss
       pop    ax
       push   ax
       and    ax,0f00h
       mov    cl,8
       shr    ax,cl
       call   prss
       pop    ax
       push   ax
       and    ax,0f0h
       mov    cl,4
```

```
        shr     ax,cl
        call    prss
        pop     ax
        push    ax
        and     ax,0fh
        call    prss
        pop     ax
        pop     cx
        pop     bx
        ret
prax    endp
; Печать двухзначного шестнадцатеричного числа из AL
pral    proc    near
pusha
        push    ax
        mov     cl,4
        shr     al,cl
        call    prss
        pop     ax
        call    prss
        popa
        ret
pral    endp
; Перевод строки, возврат каретки
wk      proc    near
        mov     al,0dh          ; Код возврата каретки
        call    print           ; Печать 1 ASCII символа
        mov     al,0ah          ; Код перевода строки
        jmp     print           ; Печать 1 ASCII символа
wk      endp
; Печать пробела
space proc      near
        mov     al,32           ; Код пробела
        jmp     print           ; Печать 1 ASCII символа
space endp
; Печать 4 пробелов
sp4     proc    near
```

```
        call    space
        call    space
call    space
jmp     space
sp4     endp
; Печать одного hex символа из мл. тетр. al
prss    proc    near
        and     al,0fh
        add     al,30h
        cmp     al,39h
        jle     print
        add     al,7
; Печать 1 ASCII символа
print   proc    near
        pusha
mov     bx,0fh
        mov     ah,0eh
        int     10h
        popa
        ret
print   endp
prss    endp
Code    ENDS
        END  Start
```

3.7 вопросы для самопроверки

1 Как скомпилировать программу типа .com?
2 Для чего нужна псевдокоманда assume?
3 Как организован текстовый режим?
4 Как организован графический режим?
5 Как создать окно на экране?
6 Как организовать прокрутку в окне на экране?
7 Как организивать в окне прокоутку снизу вверх?
8 Как организовать вывод цветного текста на экран?
9 Как можно изменить символ в знакогенераторе?
10 Как создать движущуюся фигуру на экране?
11 Как опросить клавиатуру «на лету»?

12 Каков скан-код клавиши Esc?

13 Какие порты отвечают за системный звук?

14 Как самым простым образом вызвать системный писк?

15 Какова исходная частота делителя честоты звукового таймера?

16 Как запрограммировать звуковой таймер?

17 Каков режим по умолчанию работы звукового таймера?

18 Как сделать длительность звука независимой от частоты процессора?

19 Как заставить сигнал звучать ровно 2 секунды?

20 Как осуществляется работа с файлами под DOS?

21 Как отрыть файл в произвольном каталоге?

22 Для чего надо закрывать файл?

23 Почему в файловых фугкциях надо обрабатывать ошибки?

24 Что такое дескриптор файла?

25 Почему не используются FCB?

26 Что такое FCB?

27 Как напечатать символ из аккумулятора?

28 Почему в программе типа .com не создают стека?

29 Что находится в сегментных регистрах после запуска программы типа .com?

30 Что находится в сегментных регистрах после запуска программы типа .exe?

4 Резидентные программы в MS-DOS

4.1 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.

4.2 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.

Пример 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 примеру 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.

4.3 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.

4.4 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 примере 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.

4.5 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 в таблицу 4.1. 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).

Таблица 4.1 – Структура префикса программного сегмента

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

Ниже приведен фрагмент программы выполняющей анализ командной строки.

```
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 примере резидентной программы выполняется анализ командной строки для распознавания ключа выгрузки /u.

4.6 Примеры 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
```

4.7 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

5 ДРАЙВЕРЫ УСТРОЙСТВ В СРЕДЕ MS-DOS

5.1 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, показанной на рисунке 5.1.

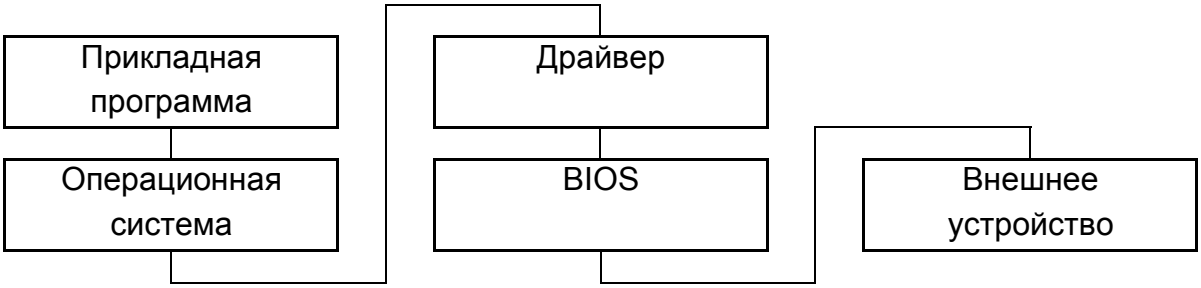


Рисунок 5.1 – Взаимодействие программы с внешним устройством

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.

5.2 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.

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 (таблица 5.1).

Рисунок 5.1 – Заголовок драйвера устройства

Таблица 5.1 – Поле DevAttr заголовка устройства

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 в таблицу 5.2.

Таблица 5.2 – Команды драйвера DOS

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

5.3 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.

5.4 Создание драйверов блочных устройств

Для написания драйвера блочного устройства (обычно это диски) необходимо хорошо представлять себе структуру данных этого блочного устройства. Ниже рассматривается пример 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 (таблица 5.3) и программы начальной загрузки (рисунок 5.2).

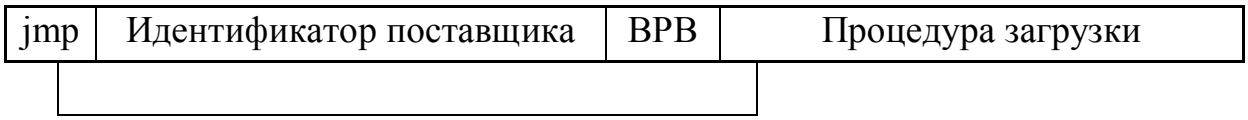


Рисунок 5.2 – Структура загрузочной записи

- 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.

Таблица 5.3 – 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 (рисунок 5.3). 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.

Таблица размещения файлов FAT содержит информацию об использовании дискового пространства файлами (таблица 5.4). В FAT имеется элемент для каждого доступного кластера.

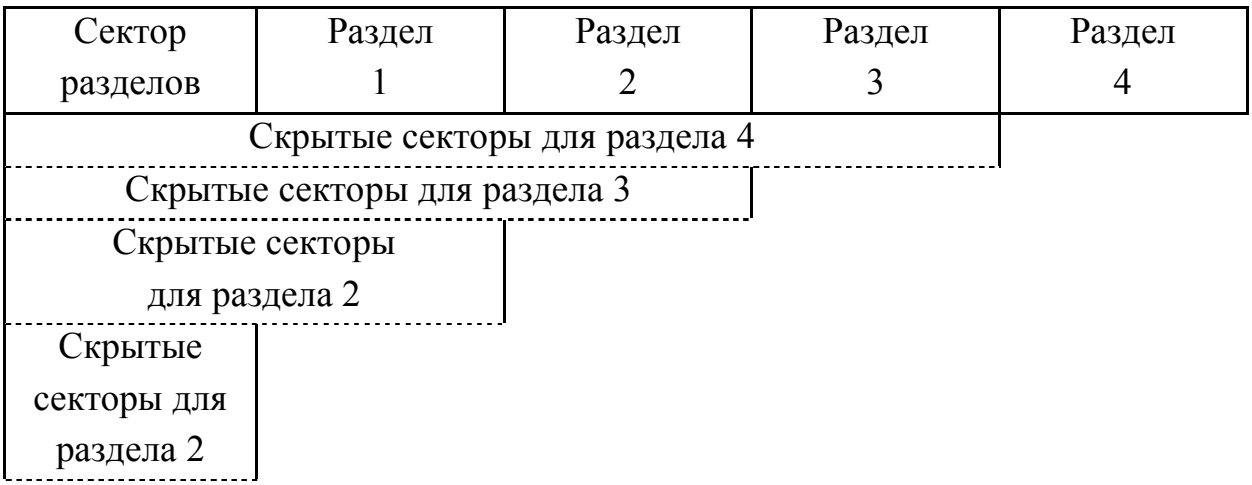


Рисунок 5.3 – Начальные секторы жесткого диска

Тпблица 5.4 – Значения элекментов FAT

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 (таблица 5.5).

Таблица 5.5 – Размер каталога для различных носителей

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

Структура элемента каталога (как корневотго, к и подкаталогов) одинакова для различных носителей (таблица 5.6)

Таблица 5.6 – Структура элемента каталога

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 приведены в таблице 5.7

Таблица 5.7 – Значения атрибутов 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 (таблица 5.8) и дата (таблица 5.9) создания или последней модификации подкаталога не изменяется при добавлении в подкаталог нового элемента.

Таблица 5.8 – Указание времени создания файла

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

Таблица 5.9 – Указание даты создания файла

Поле даты	Поле	Смещение	Биты
	Год (относительно 1980 года)	19h	7 – 1
	Месяц	19h	0
		18h	7 – 5
	Число	18h	4 – 0

5.5 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 (100K), размер FAT и размер каталога. BPB RAM-диска показан в таблице 5.10.

Таблица 5.10 – Структура блока параметров биос 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

Составные части RAM-диска показаны в таблице 5.11.

Таблица 5.11 – Секторы RAM-диска

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_segdw     ?                      ; Сегмент конца драйвера
rh0_bpb_tbo         dw     ?              ; Смещение указателя массива BPB
rh0_bpb_tbsdw     ?                      ; 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              ; ЗЗ 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    ; Сегмент
; 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 обрабаываются 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 «ошибка» 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
```

5.6 Драйвер консоли

В качестве примера драйвера символьного устройства рассмотрим 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 пример текста 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 устройййства 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      ?       ; Сегмент 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
```

5.7 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     ; Установоить сегмент вектора
```

5.8 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 запись

6 СОЗДАНИЕ ПРОГРАММЫ НА АССЕМБЛЕРЕ

В этой главе мы познакомимся со специальными программными средствами, предназначенными для преобразования исходных текстов на ассемблере к виду, приемлемому для выполнения на компьютере, и научимся использовать их [8].

Но прежде чем обсуждать сами инструментальные средства разработки программ, рассмотрим принципы разработки программного обеспечения. Для начинающего программиста, характерен большой интерес к практической работе и, возможно, разработку программы он производит на чисто интуитивном уровне. До определенного момента здесь нет ничего страшного – это даже естественно. Но совсем не задумываться над тем, как правильно организовать разработку программы (не обязательно на ассемблере), нельзя, так как хаотичность и ставка только на интуицию в конечном итоге станут стилем программирования. А это может привести к тому, что рано или поздно за программистом закрепится слава человека, у которого программы работают «почти всегда» со всеми вытекающими отсюда последствиями для карьеры. Поэтому необходимо помнить одно золотое правило: надежность программы достигается, в первую очередь, благодаря ее правильному проектированию, а не бесконечному тестированию.

Это правило означает, что если программа правильно разработана в отношении, как структур данных, так и структур управления, то это в определенной степени гарантирует правильность ее функционирования. При применении такого стиля программирования ошибки являются легко локализуемыми и устранимыми.

О том, как правильно организовать разработку программ (независимо от языка), написана не одна сотня книг. Большинство авторов предлагают следующий процесс разработки программы (мы адаптируем его, где это необходимо, к особенностям ассемблера):

1. Этап постановки и формулировки задачи:

- изучение предметной области и сбор материала в проблемно-ориентированном контексте;
- определение назначения программы, выработка требований к ней и представление требований, если возможно, в формализованном виде;
- формулирование требований к представлению исходных данных и выходных результатов;
- определение структур входных и выходных данных;
- формирование ограничений и допущений на исходные и выходные данные.

2. Этап проектирования:

- формирование «ассемблерной» модели задачи;
- выбор метода реализации задачи;
- разработка алгоритма реализации задачи;
- разработка структуры программы в соответствии с выбранной моделью памяти.

3. Этап кодирования:

- уточнение структуры входных и выходных данных и определение ассемблерного формата их представления;
- программирование задачи;
- комментирование текста программы и составление предварительного описания программы.

4. Этап отладки и тестирования:

- составление тестов для проверки правильности работы программы;
- обнаружение, локализация и устранение ошибок в программе, выявленных в тестах;
- корректировка кода программы и ее описания.

5. Этап эксплуатации и сопровождения:

- настройка программы на конкретные условия использования;
- обучение пользователей работе с программой;
- организация сбора сведений о сбоях в работе программы, ошибках в выходных данных, пожеланиях по улучшению интерфейса и удобства работы с программой;
- модификация программы с целью устранения выявленных ошибок и, при необходимости, изменения ее функциональных возможностей.

К порядку применения и полноте выполнения перечисленных этапов нужно подходить разумно. Многое определяется особенностями конкретной задачи, ее назначением, объемом кода и обрабатываемых данных, а также другими характеристиками задачи. Некоторые из этих этапов могут либо выполняться одновременно с другими этапами, либо вовсе отсутствовать. Главное, чтобы, приступая к созданию нового программного продукта, программист помнил о необходимости его концептуальной целостности и недопустимости анархии в процессе разработки.

Приведенные ранее примеры программ на ассемблере выполнялись нами в полном согласии с этим процессом. После написания программы на ассемблере нужно было ввести программу в компьютер, перевести в машинное представление

и выполнить. Как это сделать? Дальнейшее обсуждение будет посвящено именно этому вопросу.

Традиционно у существующих реализаций ассемблера нет интегрированной среды, подобной интегрированным средам Turbo Pascal, Turbo C или Visual C++.

Поэтому для выполнения всех функций по вводу кода программы, ее трансляции, редактированию и отладке необходимо использовать отдельные служебные программы. Большая часть их входит в состав специализированных пакетов ассемблера.

Общая схема процесса разработки программы на ассемблере расмотрим на примере программы из раздела 3. Схема состоит из четырех основных этапов – Ввод исходного текста – Создание объектного модуля – Создание загрузочного модуля – Отладка программыю На первом этапе, когда вводится код программы, можно использовать любой текстовый редактор. Основным требованием к нему является то, чтобы он не вставлял посторонних символов (спецсимволов редактирования). Файл должен иметь расширение . asm.

Программы, реализующие остальные шаги схемы, входят в состав программного пакета ассемблера. Традиционно на рынке ассемблеров для микропроцессоров фирмы Intel имеется два пакета:

- «Макроассемблер» MASM фирмы Microsoft.
- Turbo Assembler TASM фирмы Borland.

У этих пакетов много общего. Пакет макроассемблера фирмы Microsoft (MASM) получил свое название потому, что он позволял программисту задавать макроопределения (или макросы), представляющие собой именованные группы команд. Они обладали тем свойством, что их можно было вставлять в программу в любом месте, указав только имя группы в месте вставки. Пакет Turbo Assembler (TASM) интересен тем, что имеет два режима работы. Один из этих режимов, называемый MASM, поддерживает все основные возможности макроассемблера MASM. Другой режим, называемый IDEAL, предоставляет более удобный синтаксис написания программ, более эффективное использование памяти при трансляции программы и другие новшества, приближающие компилятор ассемблера к компиляторам языков высокого уровня.

В эти пакеты входят трансляторы, компоновщики, отладчики и другие утилиты для повышения эффективности процесса разработки программ на ассемблере. Мы воспользуемся тем, что транслятор TASM, работая в режиме MASM, поддерживает почти все возможности транслятора MASM. Для работы с данной книгой вполне достаточно иметь пакет ассемблера фирмы Borland — TASM 3.0 или выше. Обратившись к этому пакету, мы «убьем сразу двух зайцев»

— изучим основы и TASM, и MASM. В будущем это позволит вам при необходимости использовать любой из этих пакетов.

6.1 Создание объектного модуля (трансляция программы)

Итак, исходный текст программы на ассемблере подготовлен и записан на диск. Следующий шаг — трансляция программы. На этом шаге формируется объектный модуль, который включает в себя представление исходной программы в машинных кодах и некоторую другую информацию, необходимую для отладки и компоновки его с другими модулями. Для получения объектного модуля исходный файл необходимо подвергнуть трансляции при помощи программы *tasm.exe* из пакета *TASM.* Формат командной строки для запуска tasm.exe следующий:

TASM [опции] имя_исходного_файла [,имя_объектного_файла]
[,имя_файла_листинга] [,имя_файла_перекрестных_ссылок]

На первый взгляд, все очень сложно. Не пугайтесь — если вы вдруг забыли формат командной строки и возможные значения параметров, то получить быструю справку на экране монитора можно, просто запустив tasm.exe без задания каких-либо аргументов. Обратите внимание, что большинство параметров заключено в квадратные скобки. Это общепринятое соглашение по обозначению параметров, которые могут отсутствовать. Таким образом, обязательным аргументом командной строки является лишь имя_исходного_файла. Этот файл должен находиться на диске и обязательно иметь расширение . asm. За именем исходного файла через запятую могут следовать необязательные аргументы, обозначающие имена объектного файла, файла листинга и файла перекрестных ссылок. Если не задать их, то соответствующие файлы попросту не будут созданы. Если же их нужно создать, то необходимо учитывать следующее:

- Если имена объектного файла, файла листинга и файла перекрестных ссылок должны совпадать с именем исходного файла (наиболее типичный случай), то нужно просто поставить запятые вместо имен этих файлов:
  tasm.exe prg_3_1 , , , ,
  В результате будут созданы файлы, как показано на рисунке 6.1
  для шага 2.
- Если имена объектного файла, файла листинга и файла перекрестных ссылок не должны совпадать с именем исходного файла, то нужно в соответствующем порядке в командной строке указать имена соответствующих файлов, к примеру:

tasm.exe prg_3_1 , ,prg_list , ,

В результате на диске будут созданы файлы

prg_.3_1.obj

prg_list.lst

prg.3J.crf

– Если требуется выборочное создание файлов, то вместо ненужных файлов необходимо подставить параметр nul. Например:

tasm.exe prg_3_1 , ,nul, ,

В результате на диске будут созданы файлы

prg_.3_1.obj prg_.3_l.crf

Необязательный аргумент опции позволяет задавать режим работы транслятора TASM. Этих опций достаточно много. Некоторые из опций понадобятся нам в ближайшее время, а большинство из них, скорее всего, никогда не будут вами востребованы.

6.2 Создание загрузочного модуля (компоновка программы)

После устранения ошибок и получения объектного модуля, можно приступать к следующему шагу — созданию исполняемого (загрузочного) модуля, или, как еще называют этот процесс, к компоновке программы. Главная цель этого шага — преобразовать код и данные в объектных файлах в их перемещаемое выполняемое отображение. Чтобы понять, в чем здесь суть, нужно разобраться, зачем вообще разделяют процесс создания исполняемого модуля на два шага — трансляцию и компоновку. Это сделано намеренно для того, чтобы можно было объединять вместе несколько модулей (написанных на одном или нескольких языках). Формат объектного файла позволяет, при определенных условиях, объединить несколько отдельно оттранслированных исходных модулей в один модуль. При этом в функции компоновщика входит разрешение внешних ссылок (ссылок на процедуры и переменные) в этих модулях. Результатом работы компоновщика является создание загрузочного файла с расширением *.exe* или *.com*. После этого операционная система может загрузить такой файл в память и выполнить его.

Полный формат командной строки для запуска компоновщика достаточно сложен, но нам достаточно упрощенного формата:

TLINK [опции] список_объектных_файлов [, имя_загрузочного_модуля]
[,имя_фай-ла_карты] [,имя_файла_библиотеки]

Здесь:

- опции – необязательные параметры, управляющие работой компоновщика,
- *список__объектных_файлов* — обязательный параметр, содержащий список компонуемых файлов с расширением . obj. Файлы должны быть разделены пробелами или знаком «+», к примеру

tlink /v prog + mdf + fdr

- *имя_загрузочного_модуля* — необязательный параметр, обозначающий имя целевого исполняемого модуля. Если оно не указано, то имя загрузочного модуля будет совпадать с первым именем объектного файла из списка объектных файлов,
- *имя_файла_карты* — необязательный параметр, наличие которого обязывает компоновщик создать специальный файл с картой загрузки. В ней перечисляются имена, адреса загрузки и размеры всех сегментов, входящих в программу,
- *имя_файла_библиотеки* — необязательный параметр, который представляет собой путь к файлу библиотеки. Этот файл с расширением .lib создается и обслуживается специальной утилитой *tlib.exe* из пакета *TASM*. Данная утилита позволяет объединить часто используемые подпрограммы в виде объектных модулей в один файл. В дальнейшем мы можем указывать в командной строке *tlink.exe* имена нужных для компоновки объектных модулей и *имя_файла_библиотеки*, в которой следует искать подпрограммы с этими именами.

Так же как и для синтаксиса *tasm.exe*, совсем не обязательно запоминать подробно синтаксис команды *tlink.exe*. Для того чтобы получить список опций программы *tlink.exe*, достаточно просто запустить ее без указания параметров.

Для выполнения нашего примера запустим программу *tlink.exe* командной строкой вида

tlink.exe /v prg_3_1.obj

В результате вы получите исполняемый модуль с расширением .exe – *prg_3_1.exe*.

Если запустить программу *tlink.exe* с ключом */t* – командной строкой вида

tlink.exe /t prg_3_1.obj

В результате получитсz исполняемый модуль с расширением *.com* – *prg_3_1.com*.

Получив исполняемый модуль, не спешите радоваться. К сожалению, устранение синтаксических ошибок еще не гарантирует того, что программа будет

хотя бы запускаться, не говоря уже о правильности работы. Поэтому обязательным этапом процесса разработки является отладка.

На этапе отладки, используя описание алгоритма, выполняется контроль правильности функционирования, как отдельных участков кода, так и всей программы в целом. Но даже успешное окончание отладки еще не является гарантией того, что программа будет работать правильно со всеми возможными исходными данными. Поэтому нужно обязательно провести тестирование программы, то есть проверить ее работу на «пограничных» и заведомо некорректных исходных данных. Для этого составляются тесты. Вполне возможно, что результаты тестирования вас не удовлетворят. В этом случае придется вносить поправки в код программы, то есть возвращаться к первому шагу процесса разработки (см. рисунок 6.1).

Специфика программ на ассемблере состоит в том, что они интенсивно работают с аппаратными ресурсами компьютера. Это обстоятельство заставляет программиста постоянно отслеживать содержимое определенных регистров и областей памяти. Естественно, что человеку трудно следить за этой информацией с большой степенью детализации. Поэтому для локализации логических ошибок в программах используют специальный тип программного обеспечения — программные отладчики.

Отладчики бывают двух типов:

- интегрированные — отладчик реализован в виде интегрированной среды типа среды для языков Turbo Pascal, Quick C и т. д.;
- автономные — отладчик представляет собой отдельную программу.

Из-за того, что ассемблер не имеет своей интегрированной среды, для отладки написанных на нем программ используют автономные отладчики. К настоящему времени разработано большое количество таких отладчиков. В общем случае с помощью автономного отладчика можно исследовать работу любой программы, для которой создан исполняемый модуль, независимо от того, на каком языке был написан его исходный текст.

6.3 Вопросы для самопроверки

1 Как скомпилировать ассемблерную программу?
2 Как скрмпоновать программу?
3 Чем отличается компоновка программ ,com и ,exe?
4 Как использлвать несколько сегментов в программе?
5 Как организовать стек в программе?
6 Сколько памяти отводит MS-DOS программе?

7 ОСОБЕННОСТИ РАБОТЫ С 32-РАЗРЯДНЫМИ ПРОЦЕССОРАМИ

7.1 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*.

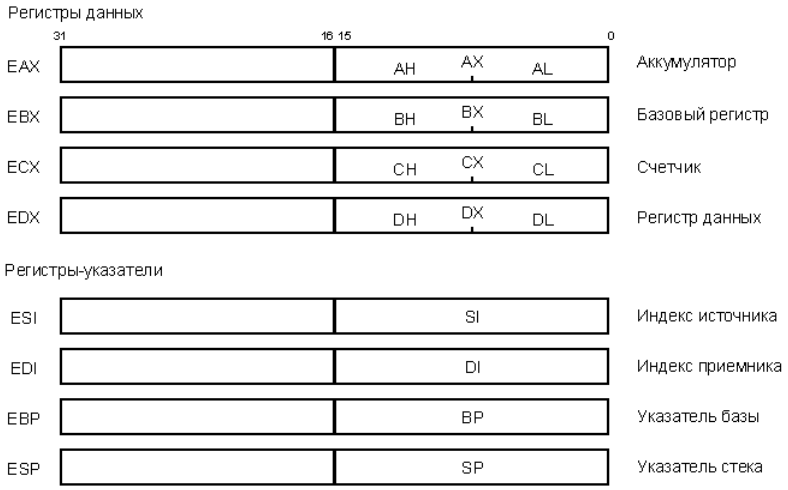


Рисунок 7.1 – Регистры общего назначения

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)
```

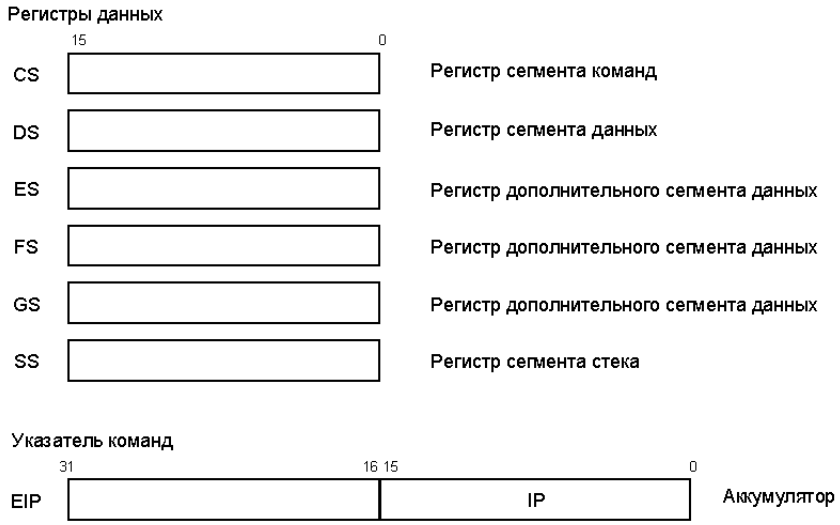


Рисунок 7.2 – Сегментные регистры и указатель команд

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 7.3).

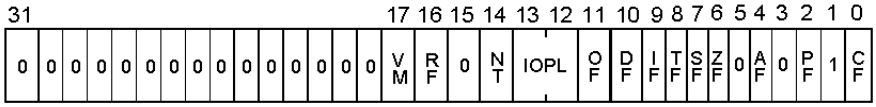


Рисунок 7.3 – Регистр флагов *EFLAGS*

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 7.4), 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*:

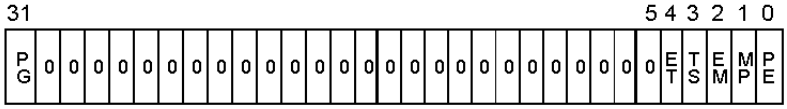


Рисунок 7.4 – Управляющий регистр *CR0*

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 7.5) используются в защищенном режиме работы процессора. Они задают расположение системных таблиц, служащих для организации сегментной адресации в защищенном режиме.

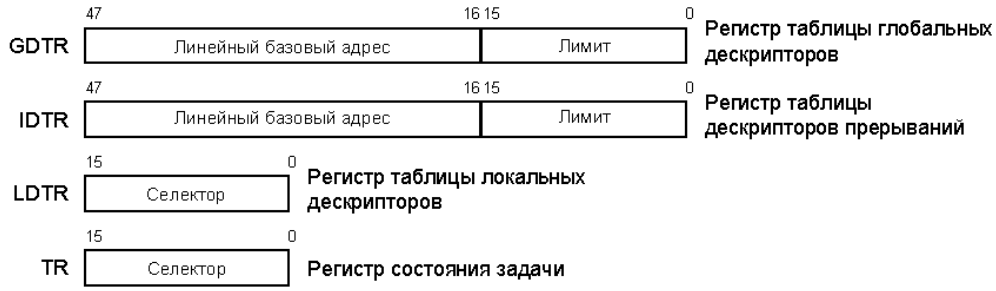


Рисунок 7.5 – Регистры системных адресов

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.

Отладочные регистры и регистры тестирования будут рассмотрены в главе 4.

Рассмотрим пример простой программы для 32-разрядного процессора.

Пример 1.1. 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 данном примере 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 7.6.

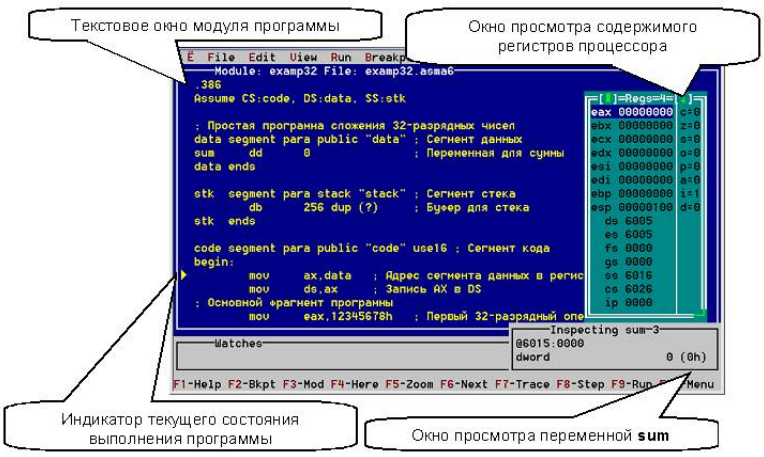


Рисунок 7.6 – Окно отладчика с исходным состоянием программы и переменных

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 1.6 видно, что содержимое аккумулятора в окне просмотра регистров процессора и содержимое переменной ***sum*** в окне просмотре переменных нулевые. На рисунке 7.7 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).

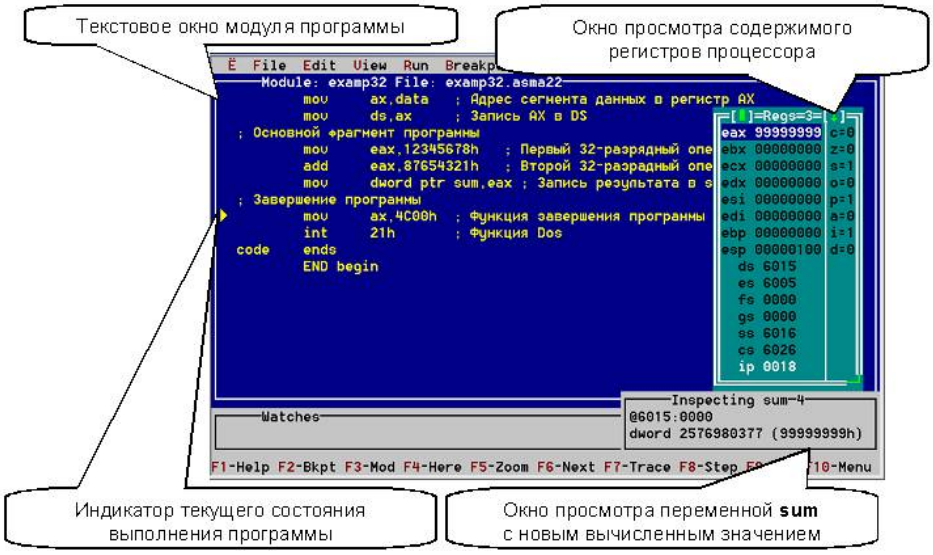


Рисунок 7.7 – Окно отладчика с результатом выполнения 32-разрядного сложения

В рассмотренном примере 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.

7.2 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.

Листинг 7.1 – 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>
```

```
24 ; Селектор 16 - сегмент кода
25 gdt_code  descr <code_size-1,0,0,98h,0,0>
26 ; Селектор 24 – сегмент стека
27 gdt_stack descr <255,0,0,92h,0,0>
28 ;Селектор 32 - видеобуфер
29 gdt_screen descr <4095,8000h,0bh,92h,0,0>
30 gdt_size=$-gdt_null              ; Размер GDT
31 ;=============================================================
32 ; Поля данных программы
33 pdescr      dq      0       ; Псевдодескриптор для команды lgdt
34 symdb      1       ; Символ для вывода на экран
35 attr  db      1ah   ; Атрибут символа
36 mesdb      27,'[31;42mReal mode now',27,'[0m',10,13,'$'
37 mes1       db      26 dup(32),'A message in protected mode',27 dup(32),0
38 data_size=$-gdt_null                  ; Размер сегмента данных
39 dataends
40 ;=============================================================
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
106 screen:
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 stksegment       stack 'stack'
```

```
172         db          256 dup(0)
173 stkends
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 7.8) [8].

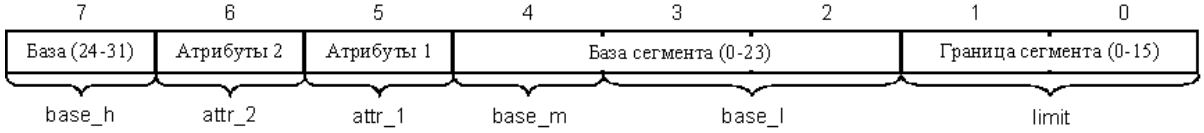


Рисунок 7.8 – Дескриптор сегмента

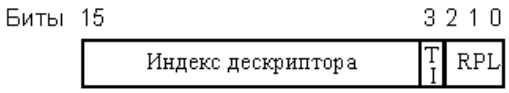


Рисунок 7.9 – Селектор дескриптора

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 нашего примера 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 примере 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 рассмотренном примере 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 вся программа примера, 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 7.10.

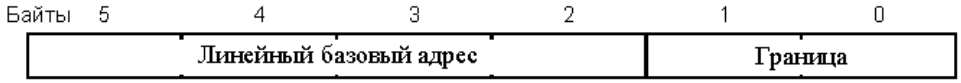


Рисунок 7.10 – Формат псевдодескриптора

В нашем примере 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 примере 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 примере 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.

Рисунок 7.11 – Сегментные и теневые регистры

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.

В процессоре для каждого из сегментных регистров имеется так называемый теневой регистр дескриптора, который имеет формат дескриптора (рисунок 7.11). 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 примера 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 нашем примере 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 примере 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.

7.3 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 (примеры)?
19. Каково главное ограничение реального режима работы процессора?
20. Какие дополнительные возможности появляются в защищенном режиме работы микропроцессора?

8 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].

8.1 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 примеры 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 8.1.

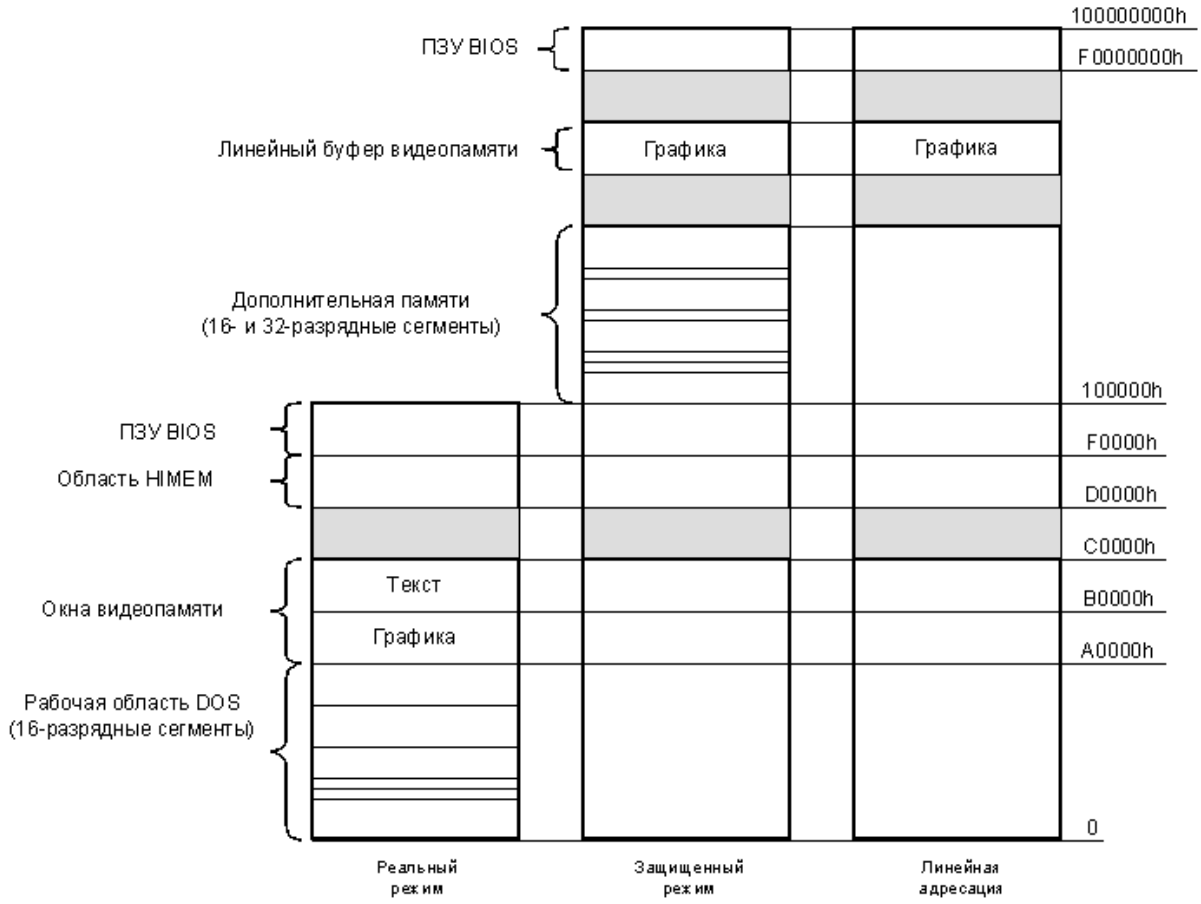


Рисунок 8.1 – Организация адресного пространства в реальном, защищенном и линейном режимах работы процессора x86

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).

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 примере 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 8.2. 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.

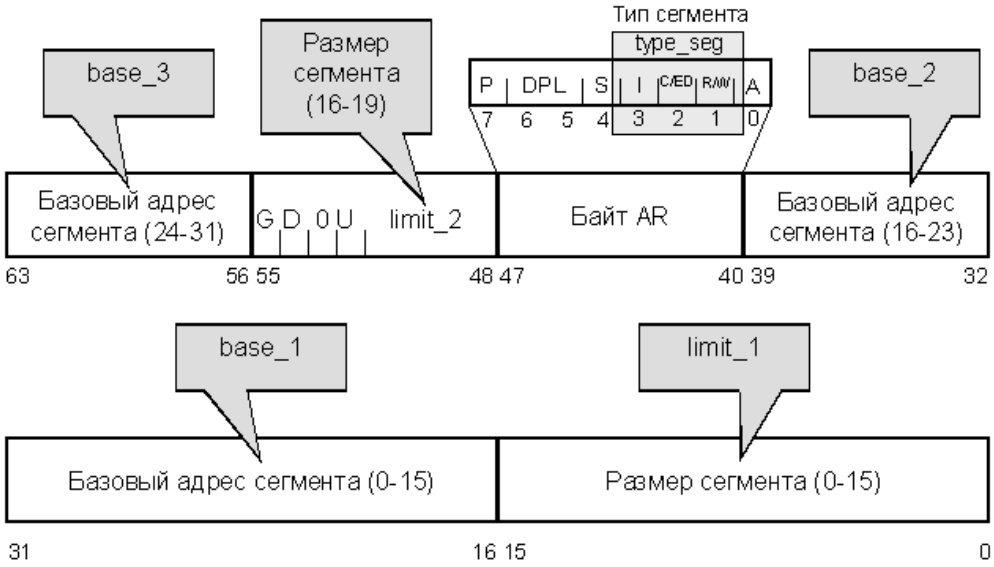


Рисунок 8.2 – Формат дескриптора сегмента

- 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.

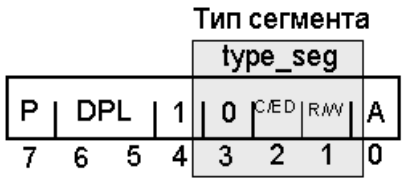


Рисунок 8.3 – Формат прав доступа для сегмента данных

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.

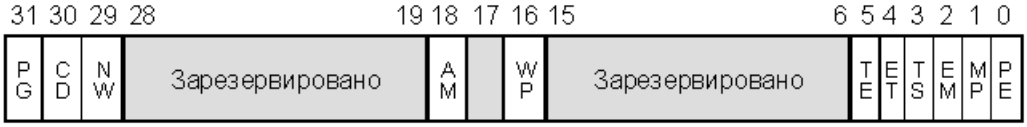


Рисунок 8.4 – Формат управляющего регистра CR0

Осуществить загрузку теневых регистров можно только в защищенном режиме. Для переключения режимов работы процессора используется регистр управления CR0, формат которого показан на рисунке 8.4 [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.

Листинг 8.1 – 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 примера 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.

Листинг 8.2 – 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 в листинге 8.3 [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.

Листинг 8.3 – 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
```

Листинг 8.4 [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).

Листинг 8.4 – 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.

Ниже приведены файлы, включаемые в программу, приведенную в листинге 8.4 [9].

Листинг 8.5 – 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
```

Листинг 8.6 – 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
```

8.2 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. Что такое бит гранулярности дескриптора?
13. Каков наибольший размер сегмента в защищенном режиме?
14. Что такое глобальная таблица дескрипторов?
15. Кто может пользоваться глобальной таблицей дескрипторов?
16. Что такое локальная таблица дескрипторов?
17. Что содержит локальная таблица дескрипторов?
18. Чем отличается локальная таблица дескрипторов от глобальной?
19. Как выполняется перевод процессора в защищенный режим из реального?
20. Что такое псевдодескриптор?
21. Какая команда используется для загрузки глобальной таблицы дескрипторов?

22. Какая команда используется для загрузки локальной таблицы дескрипторов?
23. Можно ли обратиться к глобальной таблице дескрипторов из пользовательского приложения?
24. Где находится глобальная таблица дескрипторов?
25. Можно ли из пользовательского режима осуществить запись в сегмент кода?
26. Можно ли из режима ядра осуществить запись в сегмент кода?
27. Можно ли выполнить код, записанный в сегмент данных?
28. Чем отличается обычный сегмент данных от сегмента данных стека?
29. Какие уровни защиты сегментов существуют?
30. Что такое теневые регистры дескрипторов?
31. Для чего необходимы теневые регистры дескрипторов?
32. Почему перед переводом процессора в защищенный режим надо запретить все прерывания?
33. Почему нельзя корректно завершить программу, находясь в защищенном режиме?
34. Когда загружаются теневые регистры дескрипторов процессора?
35. Каким образом обнуляется стек предвыбранных команд при переходе в защищенный режим?
36. Зачем обнуляется стек предвыбранных команд при переходе в защищенный режим?
37. Как производится возврат из защищенного режима работы процессора в реальный?
38. Что такое линейный режим адресации памяти в реальном режиме?
39. С каким объемом памяти позволяет работать режим линейной адресации в реальном режиме?
40. Чем различается организация памяти в реальном, защищенном и линейном режимах адресации?
41. Почему при работе в линейном режиме адресации необходимо включить адресный сигнал *A*20?
42. Можно ли в режиме линейной адресации работать с памятью видеоконтроллера?
43. Можно ли в режиме линейной адресации выполнять функции DOS?
44. Работают ли механизмы защиты памяти при использовании метода линейной адресации в реальном режиме?
45. Можно ли использовать строковые команды с сегментом, настроенным на линейную адресацию памяти?

ЛИТЕРАТУРА

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. Кулаков В. Программирование на аппаратном уровне. Специальный справочник. – СПб: Питер, 2010. – 496 с.: ил.
10. Thomas Roden. Four Gigabytes in Real Mode. – Programmer’s Journal 7.6, 2009.
11. Intel Architecture Software Developer’s Manual, Volume1: Basic Architecture. – Intel Corp., 2009.
12. Intel Architecture Software Developer’s Manual, Volume1: Instruction Set Reference Architecture. – Intel Corp., 2009.
13. Intel Architecture Software Developer’s Manual, Volume1: System Programming. – Intel Corp., 2009.
14. Гук М. Процессоры Pentium II, Pentium Pro и просто Pentium. – СПб: Питер Ком, 2009. – 288 с.: ил.
15. Рудаков П.И., Финогенов К.Г. Программируем на языке ассемблера IBM PC. Изд. 3-е. – Обнинск: Изд-во «Принтер», 2009, – 495 с.: ил.
16. Гук М. Аппаратные средства IBV PC. Энциклопедия. – СПб.: Питер, 2001. – 922 с.
17. Таненбаум Э. Современные операционные системы. – СПб.: Питер, 2002. – 1040 с.
18. Столлингс В. Операционные системы. – М.: Вильямс, 2002. – 848 с.
19. Brent R. Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation. – ACM Transactionsjn ProgrammingLanguagesand Systems, July 1989.
20. Shore J. On the External Storage Fragmentation Produced buy First-Fit and Best-Fit Allocation Strategies. – Communication of the ACM, August 1975.

21. Beck L. System Software. – Reading MA: Addison-Wesley, 1990.
22. Clarke D., Merusi D. System Software Programming: The Way Thing Work. – Upper SaddlyRiver, HJ: Prentice Hall, 1998.
23. Кнут Д.Э . Искусство программирования. Том 1. Основные алгоритмы. – М.: Вильямс, 2000.
24. Milenkovic M.Operating System: Concepts and Design. – New York: McGraw-Hill, 1992.

Учебное издание

Рощин Алексей Васильевич

ФУНКЦИОНАЛЬНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВСТРАИВАЕМЫХ СИСТЕМ

Учебное пособие

Печатается в авторской редакции

Подписано в печать _____ . Формат 60x84 1/16
Физ. печ. л. 12. Тираж 100 экз. Изд № ____ Заказ № _____

Московский технологический университет (МИРЭА)
119454, Москва, пр. Вернадского, д. 78