

**К. Г. Финогенов**

**Самоучитель  
по системным  
функциям**

**MS-DOS**



ББК 32.973.1  
Ф60  
УДК 681.3

Финогенов К.Г.  
Ф60 Самоучитель по системным функциям MS-DOS. — Изд.  
2, перераб. и дополн. — М.: Радио и связь, Энтроп, 1995.  
— 382 с., ил.

ISBN 5-900797-02-3

Книга представляет собой учебное пособие по использованию программных средств операционной системы MS-DOS (включая версии 6.x) в прикладных программах. Последовательно рассматриваются функции DOS и BIOS, используемые для управления терминалом, обслуживания дисков, каталогов и файлов, организации резидентных программ и иерархических программных комплексов, обработки прерываний. Каждый раздел книги включает описание системных концепций и процедур, а также большое количество примеров и задач.

Во втором издании существенно расширены разделы, посвященные программированию аппаратных средств компьютера, файловой системы, обработкам прерываний и резидентным программам, а также связанным с ними недокументированным системным средствам.

Для программистов и пользователей, желающих освоить основы системного программирования в среде MS-DOS.

ББК 32.973.1

Кирилл Григорьевич Финогенов

## САМОУЧИТЕЛЬ ПО СИСТЕМНЫМ ФУНКЦИЯМ MS-DOS

Художник М.Н. Кузьмина

Лицензия № 063389 от 24 мая 1994 г.

Подписано к печати 21.11.94. Формат 60×90/16. Бумага офсетная. Печать офсетная.  
Усл. печ. л. 24. Тираж 20 000 экз. Зах. 4358  
ИЧП "Энтроп"

Производственно-издательский комбинат ВИНТИ  
140010, Люберцы, Октябрьский пр-т, 403

© Финогенов К.Г., 1995

## Введение

Книга рассчитана на тех пользователей персональных компьютеров типа IBM PC, которые хотели бы познакомиться с возможностями операционной системы MS-DOS и освоить основы системного программирования. Как известно, операционная система персональных компьютеров состоит из двух основных компонентов - базовой системы ввода-вывода (BIOS), обеспечивающей управление периферийным оборудованием компьютера, и собственно дисковой операционной системы, или DOS, в функции которой входит организация всех элементов вычислительного процесса: запуск и завершение задач, управление памятью, обслуживание файловой системы, служба времени, обработка ошибок и т.д.

Базовая система ввода-вывода, размещаемая в постоянном запоминающем устройстве BIOS, включает набор резидентных драйверов основных периферийных устройств компьютера, таких, как магнитные диски, консоль (клавиатура и экран терминала), принтер, последовательный порт, часы. Программы BIOS, обеспечивая управление аппаратурой на самом низком, "физическом" уровне, путем обращения к портам, регистрам и аппаратным буферам, являются аппаратно-зависимыми, поэтому микросхемы BIOS разных модификаций машин типа IBM PC могут отличаться друг от друга.

Программы DOS, размещаемые в файлах IO.SYS и MSDOS.SYS, образуют более высокий уровень управления компьютером. Так, если для записи данных на диск с помощью программ BIOS требуется задание номеров головки, цилиндра и сектора на конкретном дисковом, то при обращении к DOS достаточно указать спецификацию файла. Программы обслуживания файловой системы DOS анализируют содержимое диска, определяют местонахождение требуемого файла и поставят ряд запросов к BIOS на выполнение операций записи. Таким образом, DOS располагается, в логическом плане, между BIOS и программой пользователя, упрощая программные обращения к аппаратуре. С другой стороны, программы DOS обеспечивают ряд функций, не имеющих прямого отношения к аппаратуре, например, динамическое выделение памяти, запуск и завершение задач, обслуживание векторов прерываний и многое другое.

Знакомство с назначением и возможностями системных программ (их обычно называют функциями DOS и BIOS) является необходимым элементом подготовки квалифицированного программиста, независимо от того, в какой предметной области он работает. Состав функций DOS и BIOS, их возможности и ограничения в значительной степени определяют ориентацию DOS и ее пригодность для решения конкретных прикладных задач. Для программистов, работающих на языке ассемблера, функции DOS и BIOS являются основными инструментальными средствами, без которых практически не обходится ни одна программа. При разработке программ на языках высокого уровня (Паскаль, Си) многие средства DOS реализуются в неявной форме с помощью операторов языка, его встроенных функций или библиотечных процедур, и необходимость прямого использования системных функций возникает реже. Однако знакомство с внутренними возможностями DOS, ее алгоритмами и процедурами позволяет увидеть за формализмом языка высокого уровня те реальные процессы, которые будут протекать в системе при выполнении прикладной программы и, следовательно, более осознанно подойти к разработке структуры программы и ее конкретных алгоритмов.

Обращение к функциям DOS и BIOS осуществляется с помощью механизма программных прерываний. Настроив нужным образом регистры общего назначения процессора и выполнив команду программного прерывания INT с соответствующим номером, пользователь активизирует требуемую функцию DOS или BIOS. Обычно функция, отработав, возвращает в программу некоторую информацию, также передаваемую через регистры общего назначения. При написании более сложных программ требуется не только владеть стандартным интерфейсом DOS, но иметь отчетливое представление о расположении в памяти как отдельных элементов прикладной программы, так и некоторых системных программ, ячеек и таблиц. Поэтому при изучении системных средств гораздо удобнее пользоваться не языком высокого уровня, а более приближенным к аппаратным средствам компьютера языком ассемблера. Программы, написанные на этом языке, не только отличаются высокой эффективностью, но и обеспечивают максимальную наглядность, когда речь идет о таких вопросах, как управление аппаратурой или глубинные алгоритмы DOS.

В книгу включены разделы, посвященные основным вопросам программирования на языке ассемблера с использованием системных средств:

- архитектура процессора;
- структуры программ и модели памяти;
- типичные алгоритмы программ на языке ассемблера;

- файловая система IBM PC;
- системные средства ввода с клавиатуры и вывода на экран;
- управление шрифтами;
- вывод графической информации;
- обработчики прерываний;
- динамическое управление памятью;
- родительские и дочерние процессы;
- резидентные программы.

Каждый раздел включает краткое описание системных алгоритмов и процедур, а также большое количество задач на языке ассемблера, позволяющих в простой и наглядной форме уяснить возможности функций DOS и BIOS и технику их использования в прикладных программах.

Книга рассчитана на самостоятельную проработку ее читателем на персональном компьютере. Она требует минимальных начальных знаний и в то же время позволяет освоить практически весь арсенал системных средств и получить глубокое представление о возможностях и алгоритмах функционирования операционной системы MS-DOS. В конце книги приведен список рекомендуемой литературы, в которой можно найти дополнительные сведения по архитектуре, программированию и системным средствам машин типа IBM PC.

В втором издании книги существенно расширены главы, посвященные организации процессов, обработке прерываний, резидентным программам. В ряд глав включен материал по использованию недокументированных функций DOS.



## 1.1. Краткий обзор семейства микропроцессоров фирмы Intel

Микропроцессоры (МП) Intel 8086, 8088, 80286, 80386 и 80486, явившиеся в разные годы основой новых моделей персональных компьютеров фирмы IBM (IBM PC, PC/XT, PC/AT и PS/2) при всех своих различиях и особенностях сохраняют единство архитектурных принципов, системы команд и языка программирования, что обеспечивает программную совместимость многочисленных разновидностей компьютеров "типа IBM PC".

Важнейшей характеристикой любого микропроцессора является разрядность его внутренних регистров, а также внешних шин адресов и данных. МП 8086 имеет 16-разрядную внутреннюю архитектуру и такой же разрядности шину данных. Все регистры внутри процессора, в которых могут храниться данные, имеют длину 16 битов. Таким образом, максимальное целое число (данное или адрес), с которым может работать микропроцессор, составляет  $2^{16}-1=65535$  (64K-1). Однако адресная шина МП 8086 содержит 20 линий, что соответствует адресному пространству  $2^{20}=1$  Мбайт. Для того, чтобы с помощью 16-разрядных адресов можно было обращаться в любую точку 20-разрядного адресного пространства, в микропроцессоре предусмотрена сегментная адресация памяти, реализуемая с помощью четырех сегментных регистров.

Суть сегментной адресации заключается в следующем. Исполнительный 20-разрядный адрес любой ячейки памяти вычисляется процессором путем сложения начального адреса сегмента памяти, в котором располагается эта ячейка, со смещением к ней (в байтах) от начала сегмента, которое обычно называют относительным адресом, или смещением. Сегментный адрес без четырех младших битов, т.е. деленный на 16, хранится в одном из сегментных регистров. При вычислении исполнительного адреса процессор умножает содержимое сегментного регистра на 16 (путем сдвига влево на 4 двоичных разряда) и прибавляет к полученному 20-разрядному адресу относительный адрес.

Умножение базового адреса на 16 увеличивает диапазон адресуемых ячеек до величины  $64 \text{ Кбайт} \cdot 16 = 1 \text{ Мбайт}$ .

МП 8088 является, по существу, 8-разрядным вариантом МП 8086. В нем, как и в МП 8086, предусмотрена адресация физической памяти объемом до 1 Мбайт с помощью такого же набора сегментных регистров. Однако шина данных МП 8088 имеет ширину не 16, а 8 разрядов, т.е. доступ к памяти осуществляется байтами. Это обстоятельство никак не отражается на работе с процессором, так как, например, при считывании из памяти операнда-слова микропроцессор автоматически генерирует два цикла магистрали, реализующих чтение младшего и старшего байтов. С другой стороны, 8-разрядная шина данных облегчила согласование этого микропроцессора со схемами, разработанными ранее для 8-разрядных МП 8080 и 8085.

МП 80286, используемый как центральный процессор компьютеров IBM PC/AT, является усовершенствованным вариантом МП 8086, дополненным схемами управления памятью и ее защиты. МП 80286 работает с 16-разрядными операндами, но имеет 24-разрядную адресную шину, что соответствует адресному пространству  $2^{24}=16$  Мбайт. Однако описанный выше способ сегментной адресации памяти не позволяет выйти за пределы 1 Мбайт. Для преодоления этого ограничения в МП 80286 (так же, как и в МП 80386) используются два режима работы: реального адреса и виртуального защищенного адреса, или просто защищенный режим. В реальном режиме МП 80286 функционирует фактически так же, как МП 8086 с повышенным быстродействием и может обращаться лишь к 1 Мбайт адресного пространства. Оставшиеся 15 Мбайт памяти, даже если они установлены в компьютере, использоваться не могут.

В защищенном режиме по-прежнему используются сегменты и смещения в них, однако начальные адреса сегментов не вычисляются путем умножения на 16 содержимого сегментных регистров, а извлекаются из таблиц сегментных дескрипторов, индексируемых теми же сегментными регистрами. Каждый сегментный дескриптор занимает 6 байтов, из которых 3 байта (24 двоичных разряда) отводятся под сегментный адрес. Тем самым обеспечивается полное использование 24-разрядного адресного пространства.

В каждом сегментном регистре под индекс таблицы сегментных дескрипторов отводится 14 двоичных разрядов. Полный логический адрес адресуемой ячейки состоит из 14-разрядного индекса (номера) сегмента и 16-разрядного относительного адреса. Это позволяет каждой программе использовать до  $2^{30}=1$  Гбайт логического, или виртуального пространства, которое, таким образом, в 64 раза превышает максимально возможный объем физической памяти. Операционная система виртуальной памяти



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

МП 80386 и 80486 являются высокопроизводительными процессорами с 32-разрядными шинами данных и адресов и 32-разрядной внутренней архитектурой. Последнее означает, что внутренние регистры этих процессоров, в отличие от процессоров ранних моделей, имеют длину 32 бита. Поэтому максимальное целое число, с которым может работать микропроцессор, составляет  $2^{32}-1=4294967296$  (4Г-1). Во многих случаях использование 32-битовых операндов позволяет существенно упростить и ускорить вычисления. Помимо этого, в МП 80386 и 80486 расширен состав регистров, что также предоставляет программисту значительные удобства. Наконец, в новых моделях процессоров имеются встроенные средства поддержки многозадачного режима, а также мультипроцессорных систем. Естественно, что эти процессоры, как и МП 80286, могут работать в реальном и защищенном режимах. В последнем случае микропроцессор позволяет адресовать до  $2^{32}=4$  Гбайт физической памяти и  $2^{46}=64$  Тбайт виртуальной. При этом следует подчеркнуть, что разработчиками обеспечена полная совместимость новых моделей процессоров со старыми, в том смысле, что программы, написанные для процессоров 8086-80286, т.е. с использованием 16-битовых операндов, выполняются на новых процессорах без всяких исправлений. Фактически программист, создающий программу, предназначенную для работы под управлением MS-DOS, может не задумываться над тем, какой процессор установлен на его компьютере.

Используемые в настоящее время версии MS-DOS работают в реальном режиме и не обеспечивают управление виртуальной памятью. В то же время обширные классы программ, в частности, программы управления технологическими процессами или научно-исследовательскими установками, не требуют использования защищенного режима и успешно работают в среде операционной системы MS-DOS. Вообще во многих случаях относительно простая и надежная система MS-DOS, получившая, к тому же, широчайшее распространение, оказывается удобнее более совершенных, но и значительно более сложных систем, реализующих все возможности современным микропроцессоров.

## 1.2. Распределение адресного пространства

В зависимости от модификации персонального компьютера и состава его периферийного оборудования распределение адресного пространства может несколько различаться. Тем не менее размещение основных компонентов системы довольно строго

унифицировано. Типичная схема использования адресного пространства компьютера приведена на рис. 1.1. Значения адресов на этом рисунке, как и повсюду далее в книге, даны в 16-ричной системе счисления. Признаком 16-ричного числа служит буква h, стоящая после числа.

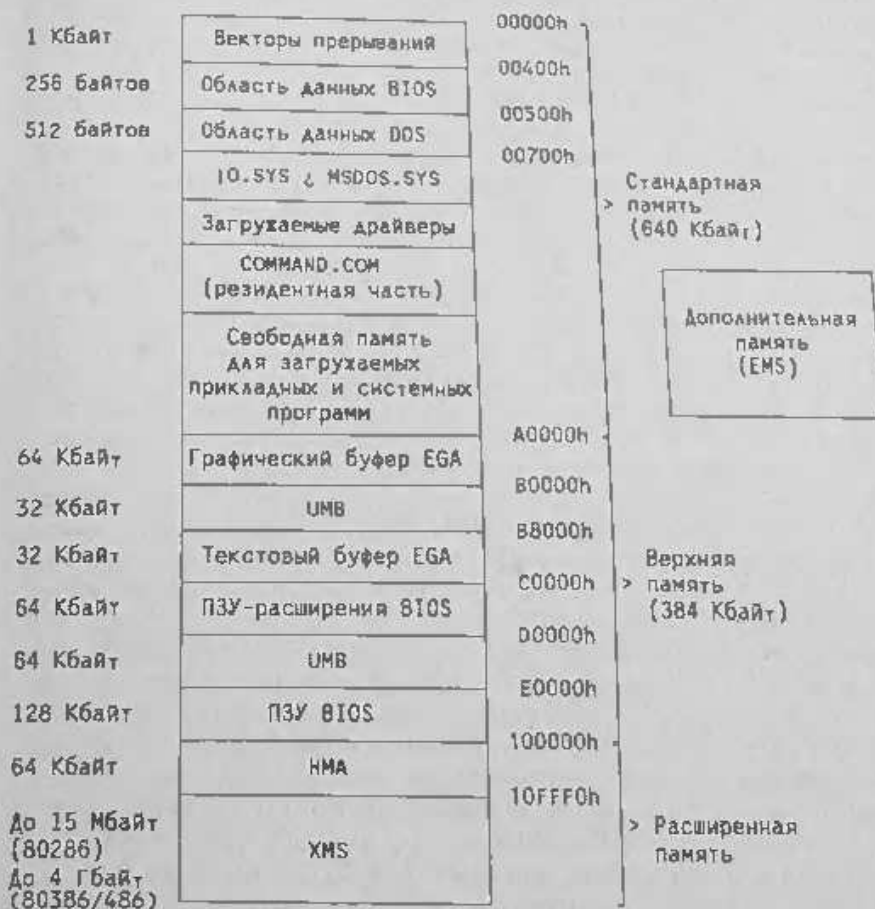


Рис. 1.1. Типичное распределение адресного пространства.

Первые 640 Кбайт адресного пространства с адресами от 00000h до 9FFFFh отводятся под основную оперативную память, которую еще называют стандартной (conventional). Начальный килобайт оперативной памяти занят векторами прерываний (256 векторов по 4 байта). Вслед за векторами прерываний располагается область данных BIOS, которая занимает

адреса от 00400h до 004FFh. В этой области хранятся разнообразные данные, используемые программами BIOS в процессе управления периферийным оборудованием. Так, здесь размещаются:

- входной буфер клавиатуры с системой указателей;
- адреса последовательных и параллельных портов;
- данные, характеризующие настройку видеосистемы (форма курсора и его текущее местоположение на экране, текущий видеорежим, ширина экрана и прочее);
- ячейки для отсчета текущего времени;
- область межзадачных связей и т.д.

Область данных BIOS заполняется информацией в процессе начальной загрузки компьютера и динамически модифицируется системой по мере необходимости; многие прикладные программы обращаются к этой области с целью чтения или модификации содержащейся в ней информации.

В области памяти начиная с адреса 500h содержатся некоторые системные данные DOS. Вслед за областью данных DOS располагается собственно операционная система, загружаемая из файлов IO.SYS и MSDOS.SYS (IBMBIO.COM и IBMDOS.COM для системы PC-DOS). Система обычно занимает несколько десятков Кбайт.

Если в файл CONFIG.SYS включены директивы DEVICE=... загрузки устанавливаемых драйверов (ADM.SYS, SMARTDRV.SYS, EMM386.EXE, ANSI.SYS и др.), то они загружаются вслед за системой. Наконец, ниже драйверов размещается резидентная часть командного процессора COMMAND.COM, занимающая около 3 Кбайт. В функции резидентной части COMMAND.COM входит обработка <Ctrl>/C, <Ctrl>/<Break> и критических ошибок, вывод сообщений об ошибках, завершение текущей задачи, загрузка транзитной части COMMAND.COM. Транзитная, загружаемая часть COMMAND.COM размещается в самом конце оперативной памяти, затирается при загрузке программ и после завершения выполняемой программы должна загружаться с диска заново.

Перечисленные выше компоненты операционной системы занимают обычно 60-90 Кбайт. Вся оставшаяся память до границы 640 Кбайт (называемая иногда транзитной областью) свободна для загрузки любых системных или прикладных программ. Как правило, в начале сеанса в память загружают резидентные программы (русификатор, электронный блокнот, резидентные расширения DOS, программы контроля состояния диска, входящие, например, в состав Нортонских утилит и др.). При наличии резидентных программ объем свободной памяти уменьшается.

Оставшиеся 384 Кбайт адресного пространства, называемого верхней (upper) памятью, первоначально были предназначены для размещения постоянных запоминающих устройств (ПЗУ). Практически под ПЗУ занята только часть адресов. В самом конце адресного пространства, в области F0000h...FFFFFFh (или E0000h... FFFFFFFh) располагается основное постоянное запоминающее устройство BIOS, а начиная с адреса C0000h - так называемое ПЗУ расширений BIOS для обслуживания графических адаптеров и дисков. Часть адресного пространства верхней памяти отводится для адресации к видеобуферам графического адаптера. Приведенное на рисунке расположение видеобуферов характерно для адаптера EGA; для других адаптеров оно может быть иным (например, видеобуфер простейшего монохромного адаптера MDA занимает всего 4 Кбайт и располагается, начиная с адреса B0000h).

В состав компьютеров PC/AT наряду со стандартной памятью (640 Кбайт) может входить расширенная (extended) память, максимальный объем которой зависит от ширины адресной шины процессора и при использовании процессора 80286 может достигать 15 Мбайт, а для процессоров 80386/486 - 4 Гбайт. Эта память располагается за пределами первого мегабайта адресного пространства и начинается с адреса 100000h. Реально на машине может быть установлен не полный объем расширенной памяти, а лишь 2 - 3 Мбайт или даже меньше, например, всего 384 Кбайт.

Поскольку функционирование расширенной памяти подчиняется "спецификации расширенной памяти" (Extended Memory Specification, сокращенно XMS), то и саму память часто называют XMS-памятью. Как уже отмечалось выше, доступ к расширенной памяти осуществляется в защищенном режиме, поэтому для MS-DOS, работающей только в реальном режиме, расширенная память недоступна.

Однако в современные версии MS-DOS включается драйвер HIMEM.SYS, поддерживающий расширенную память, т.е. позволяющий ее использовать, хотя и ограниченным образом. Конкретно в расширенной памяти можно разместить электронные диски (с помощью драйвера RAMDRIVE.SYS) или кеш-буферы диска (с помощью драйвера SMARTDRV.SYS).

Первые 64 Кбайт расширенной памяти, точнее, 64 Кбайт - 16 байт с адресами от 100000h до 10FFFFh, носят специальное название область старшей памяти (High Memory Area, HMA). Эта область замечательна тем, что хотя она находится за пределами первого мегабайта, к ней можно обратиться в реальном режиме работы микропроцессора, если определить сегмент, начинающийся в самом конце мегабайтного адресного пространства, с адреса FFFF0h, и разрешить использование



адресной линии A20. Первые 16 байтов этого сегмента заняты ПЗУ, область же со смещениями 0010h...FFFFh можно в принципе использовать под программы и данные. MS-DOS позволяет загружать в НМА (директивой файла CONFIG.SYS DOS=HIGH) значительную часть самой себя, в результате чего занятая системой область стандартной памяти существенно уменьшается. Старшую память обслуживает тот же драйвер HIMEM.SYS, поэтому загрузка DOS и НМА возможна только если установлен HIMEM.SYS.

Как видно из приведенного выше рисунка, часть адресного пространства верхней памяти, не занятая расширениями BIOS и видеобуферами, оказывается свободной. На компьютерах с МП 80386 и 80486 эти свободные участки можно использовать для адресации к расширенной памяти (конечно, не ко всей, а лишь к той ее части, объем которой совпадает с общим объемом свободных адресов верхней памяти). Переотображение расширенной памяти на свободные адреса верхней памяти выполняет драйвер EMM386.EXE, а сами участки верхней памяти, "заполненные" расширенной, называются блоками верхней памяти (Upper Memory Blocks, UMB). MS-DOS позволяет загружать в UMB устанавливаемые драйверы устройств, а также резидентные программы-расширения DOS (APPEND.EXE, DOSKEY.COM, KEYB.COM и др.). Загрузка системных программ в UMB освобождает от них стандартную память, увеличивая ее транзитную область. В UMB можно загрузить также и прикладные резидентные программы. Загрузка в UMB драйверов осуществляется директивой файла CONFIG.SYS DEVICEHIGH (вместо директивы DEVICE), а загрузка резидентных программ - командой DOS LOADHIGH.

По умолчанию драйвер EMM386.EXE преобразует в UMB 128 Кбайт расширенной памяти, располагая ее по адресам C000...CFFF. При необходимости (если, например, на эти адреса настроено какое-то нестандартное внешнее устройство) объем и расположение UMB в адресном пространстве верхней памяти можно изменить с помощью ключей в строке установки драйвера EMM386.EXE.

Независимо от наличия и объема расширенной (XMS) памяти, компьютер может быть укомплектован платой с дополнительной памятью, не отвечающей каким-либо определенным адресам 16-мегабайтного адресного пространства. Эта память функционирует в соответствии со спецификацией Lotus-Intel-Microsoft Expanded Memory Specification (LIM EMS) и может достигать объема (в версии EMS 4.0) 32 Мбайт. Обращение к EMS-памяти осуществляется через относительно узкие окна (физические страницы) размером по 16 Кбайт, в качестве которых используется часть адресного пространства верхней

памяти (от границы 640 Кбайт до 1 Мбайт). Любой блок дополнительной памяти, называемый логической страницей, может быть отображен на физическую страницу в верхней памяти, чем и обеспечивается прямая (хотя и не одновременная) адресация всего пространства дополнительной памяти. В дополнительной памяти, как и в расширенной, обычно размещают электронные диски или кэш-буферы, хотя спецификация EMS 4.0 допускает (в отличие от EMS 3.2) выполнение программ, находящихся в дополнительной памяти.

Компьютеры типа PC/AT или PS/2 обычно оснащаются расширенной памятью того или иного объема, но не всегда дополнительной. Между тем, некоторые программы в процессе своего выполнения обращаются к дополнительной памяти и при ее отсутствии просто не будут функционировать. Для того, чтобы позволить таким программам выполняться на компьютерах без дополнительной памяти, предусмотрена возможность преобразования части расширенной памяти в дополнительную. Это преобразование осуществляет тот же драйвер EMM386.EXE. По умолчанию для отображения дополнительной памяти используется диапазон адресов верхней памяти D000...DFFF, который в этом случае, естественно, выпадает из области блоков верхней памяти UMB. При необходимости область отображения дополнительной памяти можно изменить.

### 1.3. Регистры процессора

Как уже отмечалось выше, внутренняя архитектура микропроцессоров Intel практически совпадает, если не рассматривать имеющихся в старших моделях процессоров (начиная с МП 80286) схем организации защищенного режима. Поэтому ниже все эти микропроцессоры будут рассматриваться вместе под общим названием "процессор".

Процессор содержит двенадцать 16-разрядных программно-адресуемых регистров, которые принято объединять в три группы: регистры данных, регистры-указатели и сегментные регистры. Кроме того, в состав процессора входят счетчик команд и регистр флагов (рис. 1.2).

В группу регистров данных включаются регистры AX, BX, CX и DX. Программист может использовать их по своему усмотрению для временного хранения любых объектов (данных или адресов) и выполнения над ними требуемых операций. При этом регистры допускают независимое обращение к старшим (AH, BH, CH и DH) и младшим (AL, BL, CL и DL) половинам. Так, команда

mov BL, AH



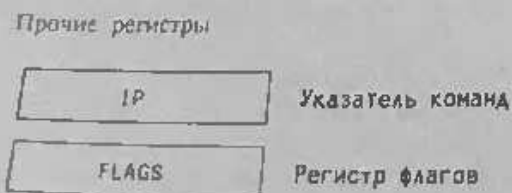
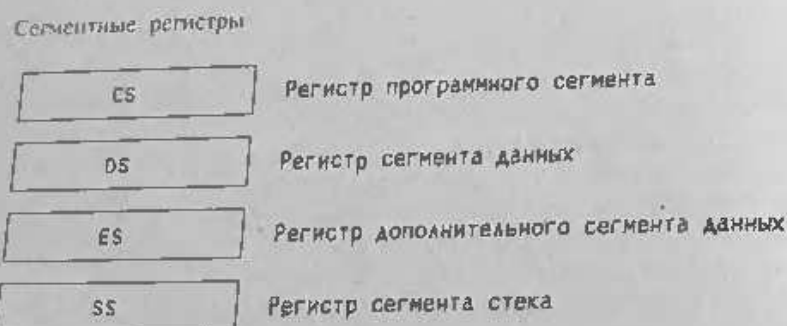
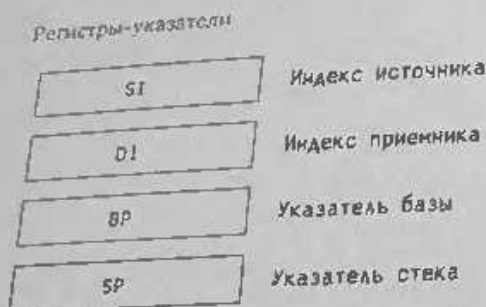
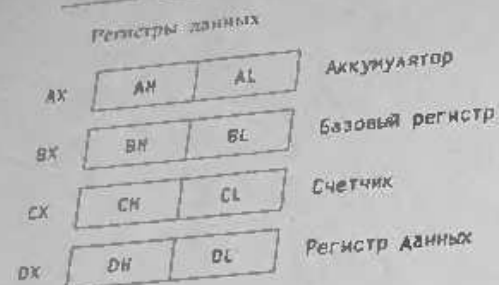


Рис. 1.2. Регистры процессора.

пересылает старший байт регистра AX в младший байт регистра BX, не затрагивая при этом вторых байтов этих регистров.

Заметьте, что сначала указывается операнд-приемник, а после запятой - операнд-источник. Во многих случаях регистры данных вполне эквивалентны, однако предпочтительнее пользоваться регистром AX, поскольку многие команды занимают в памяти меньше места и выполняются быстрее, если их операндом является регистр AX (или его половины AL или AH). С другой стороны, ряд команд использует определенные регистры неявным образом. Так, все команды циклов используют регистр CX в качестве счетчика числа повторений; в командах умножения и деления регистры AX и DX выступают в качестве неявных операндов; операции ввода-вывода можно осуществлять только через регистр AX (или AL) и т.д.

Индексные регистры SI и DI так же, как и регистры данных, могут использоваться произвольным образом. Однако их основное назначение - хранить индексы (смещения) относительно некоторой базы (т.е. начала массива) при выборке операндов из памяти. Адрес базы при этом может находиться в базовых регистрах BX или BP. Специально предусмотренные команды работы со строками используют регистры SI и DI в качестве неявных указателей и обрабатываемых строк.

Регистр BP служит указателем базы при работе с данными в стековых структурах, но может использоваться и произвольным образом в большинстве арифметических и логических операций.

Последний из группы регистров-указателей, указатель стека SP, стоит особняком от других в том отношении, что используется исключительно как указатель вершины стека, обеспечивая выполнение стековых команд (PUSH, POP и др.). Однако это не исключает его использование в качестве операнда в арифметических операциях или операциях пересылки, если требуется изменить положение вершины стека.

Регистры SI, DI, BP и SP, в отличие от регистров данных, не допускают побайтовую адресацию.

Четыре сегментных регистра CS, DS, ES и SS являются важнейшим элементом архитектуры процессора, обеспечивая адресацию 20-разрядного адресного пространства с помощью 16-разрядных операндов.

Обращение к памяти (как к стандартной памяти в пределах 640 Кбайт, так и к буферам или ПЗУ в области 640 Кбайт - 1 Мбайт) осуществляется исключительно посредством сегментов - логических образований, накладываемых на любые участки физического адресного пространства. Размер сегмента должен находиться в пределах 0 байт - 64 Кбайт (допустимы и иногда используются сегменты нулевой длины). Начальный адрес сегмента, деленный на 16, т.е. без младшей 16-ричной цифры, заносится в один из сегментных регистров. Как правило, это

действие выполняет программист с помощью соответствующих программных строк. При обращении к памяти процессор извлекает из сегментного регистра сегментный базовый адрес, умножает его на 16 сдвигом влево на 4 двоичных разряда и складывает с заданным каким-либо образом относительным адресом (смещением), получая 20-разрядный физический адрес адресуемой ячейки памяти (слова или байта). Этот процесс проиллюстрирован на рис. 1.3 на конкретном примере команды `inc mem1`.

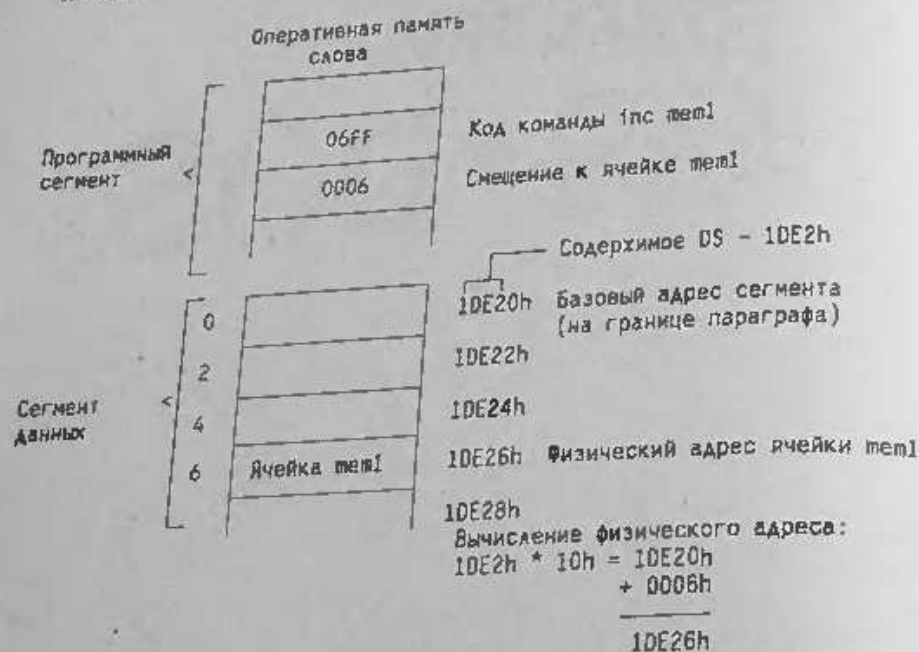


Рис. 1.3. Формирование физического адреса.

В примере предполагается, что сегмент данных, адресуемый через регистр DS, имеет базовый адрес 1DE20h, а ячейка `mem1` расположена в байтах 6 и 7 этого сегмента (смещение `mem1` относительно начала сегмента равно 6).

Поскольку младшая 16-ричная цифра базового адреса сегмента должна быть равна 0, сегмент всегда начинается с адреса, кратного 16, т.е. на границе 16-байтового блока памяти (параграфа). Число, хранящееся в сегментном регистре, называют сегментным адресом. Следует помнить, что сегментный адрес в 16 раз меньше физического; его можно рассматривать, как номер параграфа, с которого начинается данный сегмент.

Регистр CS обеспечивает адресацию к сегменту, в котором находятся программные коды, регистры DS и ES - к сегментам с данными (таким образом, в любой момент времени программа может иметь доступ к 128 Кбайт данных), а регистр SS - к сегменту стека, который на машинах типа IBM PC, в отличие от других вычислительных систем, может быть очень большим и достигать 64 Кбайт. Сегментные регистры, естественно, не могут выступать в качестве регистров общего назначения.

Указатель команд IP "следит" за ходом выполнения программы, указывая в каждый момент относительный адрес команды, следующей за исполняемой. Регистр IP программно недоступен (IP - это просто его сокращенное название, а не мнемоническое обозначение, используемое в языке программирования); наращивание адреса в нем выполняет микропроцессор, учитывая при этом длину текущей команды. Команды переходов, прерываний, вызова подпрограмм и возврата из них изменяют содержимое IP, осуществляя тем самым переходы в требуемые точки программы.

Регистр флагов, эквивалентный регистру состояния процессора других вычислительных систем, содержит информацию о текущем состоянии процессора (рис. 1.4). Он включает 6 флагов состояния и 3 бита управления состоянием процессора, которые, впрочем, тоже называются флагами.



Рис. 1.4. Регистр флагов.

Флаг переноса CF (Carry Flag) индицирует перенос или заем при выполнении арифметических операций, а также служит индикатором ошибки при обращении к системным функциям.

Флаг паритета PF (Parity Flag) устанавливается в 1, если результат операции содержит четное число двоичных единиц.

Флаг вспомогательного переноса AF (Auxiliary Flag) используется в операциях над упакованными двоично-десятичными числами. Он индицирует перенос или заем из старшей тетрады (бита 3).

Флаг нуля ZF (Zero Flag) устанавливается в 1, если результат операции равен 0.

Флаг знака SF (Sign Flag) показывает знак результата операции, устанавливаясь в 1 при отрицательном результате.

Управляющий флаг трассировки (ловушки) TF (Trace Flag) используется для осуществления пошагового выполнения программы. Если TF=1, то после выполнения каждой команды процессор реализует процедуру прерывания типа 1 (через вектор, расположенный по адресу 04).

Управляющий флаг разрешения прерываний IF (Interrupt Flag) разрешает (если равен 1) или запрещает (если равен 0) процессору реагировать на прерывания от внешних устройств.

Управляющий флаг направления DF (Direction Flag) используется командами обработки строк. Если DF=0, строка обрабатывается в прямом направлении, от меньших адресов к большим; если DF=1, обработка строки идет в обратном направлении.

Флаг переполнения OF (Overflow Flag) фиксирует переполнение, т.е. выход результата за пределы допустимого диапазона значений.

Для работы с регистром флагов предусмотрен ряд команд. Установка и сброс флагов CF, DF и IF осуществляется командами STC, STD и STI (установка) и CLC, CLD и CLI (сброс). Все содержимое регистра флагов можно сохранить в стеке командой PUSHF и извлечь из стека командой POPF; кроме того, младший байт регистра флагов (флаги CF, PF, AF, ZF и SF) можно переслать в регистр AH командой LAHF или загрузить в регистр флагов из AH командой SAHF.

## 2. Модели памяти и структуры программ

### 2.1. Структура и образ памяти программы .EXE

Программы, выполняемые под управлением MS-DOS, могут принадлежать к одному из двух типов, которым соответствуют расширения имен программных файлов .COM и .EXE. Основное различие этих программ заключается в том, что программы типа .COM состоят из единственного сегмента, в котором размещаются программные коды, данные и стек, а в программах типа .EXE для собственно программы, данных и стека предусматриваются отдельные сегменты. Таким образом, размер программы типа .COM не может превысить 64 Кбайт, а размер программы типа .EXE практически не ограничен, так как в нее может входить любое число сегментов программы и данных.

Структура типичной программы типа .EXE на языке ассемблера выглядит следующим образом.

```

title      Программа типа .EXE
text       segment 'code'
            assume  CS:text, DS:data

myproc     proc
            mov     AX,data
            mov     DS,AX
            ...
            myproc  endp
            ;Текст программы

text       ends
data       segment
            ...
            ;Определения данных

data       ends
stack     segment stack 'stack'
            dw      128 dup (0)
stack     ends
            myproc
            end

```

Следует заметить, что при вводе исходного текста программы с клавиатуры можно использовать как прописные, так и строчные буквы: транслятор воспринимает, например, строки MOV AX,DATA и mov ax,data одинаково. Однако с помощью соответствующих ключей можно заставить транслятор различать прописные и строчные буквы в именах. Тогда программные



строки MYPROC PROC и TURPROC PROC уже не будут эквивалентны. В настоящей книге принята следующая система обозначений:

- тексты программ набраны строчными буквами, за исключением обозначений регистров (AX, CS) и имен файлов (MYFILE.TXT), которые для наглядности выделены прописными буквами;
- в тексте книги (но не в программах) прописными буквами выделены зарезервированные слова, т.е. операторы языка ассемблера (SEGMENT, ENDS, MOV и т.д.), а также имена файлов.

Рассмотрим теперь структуру приведенной программы. Оператор TITLE позволяет предпослать программе текстовый заголовок, который будет выводиться на все страницы листинга трансляции. Программа состоит из трех сегментов: сегмента команд, или программного сегмента с произвольным именем text, сегмента данных с именем data и сегмента стека с именем stack (оба эти имени также могут выбираться произвольно). Каждый сегмент открывается оператором SEGMENT и закрывается оператором ENDS. Перед обоими операторами должно стоять имя сегмента. Порядок сегментов в большинстве случаев роли не играет.

Слово "CODE", стоящее в апострофах в строке описания сегмента команд, указывает класс сегмента - "программный". Классы сегментов анализируются компоновщиком и используются им при компоновке загрузочного модуля: сегменты, принадлежащие одному классу, загружаются в память друг за другом. Для простых программ, включающих один сегмент команд и один сегмент данных, эта процедура не имеет значения, однако для правильной работы компоновщика LINK и отладчика CodeView при описании сегмента команд необходимо указание его класса "CODE".

Текст сегмента команд начинается с оператора ASSUME, который позволяет транслятору сопоставить сегментные регистры и адресуемые ими сегменты. Определение CS:text указывает транслятору, что данный сегмент является программным и будет адресоваться с помощью сегментного регистра CS. Определение DS:data закрепляет за сегментом data сегментный регистр DS, как регистр, используемый по умолчанию, что позволяет ссылаться на переменные, описанные в сегменте data, без явного указания регистра DS. При этом ассемблер проверяет, действительно ли они описаны в сегменте data.

Собственно программа обычно состоит из процедур. Деление программы на процедуры не обязательно, но повышает ее наглядность и облегчает передачу управления на подпрограммы и в другие программные модули. В рассматриваемом примере

сегмент команд содержит единственную процедуру TURPROC, открываемую оператором PROC и закрываемую оператором ENDP. Перед обоими операторами указывается имя процедуры. В первых строках программы инициализируется регистр DS - в него заносится сегментный адрес сегмента данных. Поскольку передача в сегментные регистры непосредственных значений не допускается, в качестве "перевалочного пункта" используется регистр AX. После того, как регистр DS инициализирован, программа может обращаться к данным, описанным в регистре данных.

Сегмент данных содержит описания всех переменных, используемых в программе. Способы описания данных будут рассмотрены ниже.

Строка описания сегмента стека должна содержать класс сегмента - "STACK", а также тип объединения - STACK. Тип объединения указывает компоновщику, каким образом должны объединяться одноименные сегменты разных модулей - накладываясь друг на друга (тип объединения COMMON) или присоединяясь друг к другу (тип объединения STACK для сегментов стека или PUBLIC для всех остальных). Хотя для одномодульных программ тип объединения значения не имеет, для сегмента стека обязательно указание типа STACK, поскольку в этом случае при загрузке программы выполняется автоматическая инициализация регистров SS (адресом начала сегмента стека) и SP (смещением конца сегмента стека). В приведенном примере для стека зарезервировано 128 слов памяти. Класс для сегмента стека указывать не обязательно, однако в тех случаях, когда программисту важен фактический порядок расположения сегментов программы в памяти, лучше всем сегментам присвоить какие-то различающиеся классы (один сегмент может не иметь класса). В этом случае компоновщик расположит сегменты в памяти в таком порядке, в котором они следуют в исходном тексте программы.

Текст программы заканчивается директивой END, завершающей трансляцию. В качестве операнда этой директивы указывается точка входа в главную процедуру.

При загрузке программы сегменты размещаются в памяти, как показано на рис. 2.1.

Образ программы в памяти начинается с префикса программного сегмента (Program Segment Prefix, PSP), образуемого и заполняемого системой. PSP всегда имеет размер 256 байтов и содержит таблицы и поля данных, используемые системой в процессе выполнения программы. Некоторые из этих полей будут описаны ниже. Вслед за PSP располагаются сегменты программы. Сегментные регистры автоматически инициализируются следующим образом: ES и DS указывают на начало PSP (что

дает возможность, сохранив их содержимое, обращаться затем в программе к PSP), CS - на начало сегмента команда, а SS - на начало сегмента стека. В указатель команд IP загружается относительный адрес точки входа в программу (из операнда директивы END), а в указатель стека SP - смещение конца сегмента стека. Таким образом, после загрузки программы в память адресуемыми оказываются все сегменты, кроме сегмента данных. Инициализация регистра DS в первых строках программы позволяет сделать адресуемым и этот сегмент.

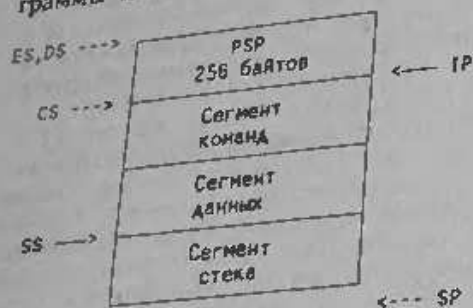


Рис. 2.1. Образ памяти программы .EXE.

## 2.2. Структура и образ памяти программы .COM

Как уже отмечалось, программа типа .COM отличается от программы типа .EXE тем, что содержит лишь один сегмент, включающий все компоненты программы: PSP, программный код (т.е. оттранслированные в машинные коды программные строки), данные и стек. Структура типичной программы типа .COM на языке ассемблера выглядит следующим образом:

```

title    Программа типа .COM
text     segment 'code'
        assume  CS:text, DS:text
        org    100h
myproc   proc                                ;Текст программы
        ...
myproc   endp                                ;Определения данных
        ...
text     ends
        end    myproc

```

Программа содержит единственный сегмент text, которому присвоен класс 'CODE'. В операторе ASSUME указано, что сегментные регистры CS и DS будут указывать на этот единственный сегмент. Оператор ORG 100h резервирует 256 байтов

для PSP. Заполнять PSP будет по-прежнему система, но место под него в начале сегмента должен отвести программист. В программе нет необходимости инициализировать сегментный регистр DS, поскольку его, как и остальные сегментные регистры, инициализирует система. Данные можно разместить после программной процедуры (как это показано на рисунке), или внутри нее, или даже перед ней. Следует только иметь в виду, что при загрузке программы типа .COM регистр IP всегда инициализируется числом 100h, поэтому сразу вслед за оператором ORG 100h должна стоять первая выполняемая строка программы. Если данные желательно расположить в начале программы, перед ними следует поместить оператор перехода на реальную точку входа, например JMP entry.

Образ памяти программы типа .COM показан на рис. 2.2. После загрузки программы все сегментные регистры указывают на начало единственного сегмента, т.е. фактически на начало PSP. Указатель стека автоматически инициализируется числом FFFEh. Таким образом, независимо от фактического размера программы ей выделяется 64 Кбайт адресного пространства, всю нижнюю часть которого занимает стек. Поскольку верхняя граница стека не определена и зависит от интенсивности и способа использования стека программой, следует опасаться затирания стеком нижней части программы. Впрочем, такая опасность существует и в программах типа .EXE.

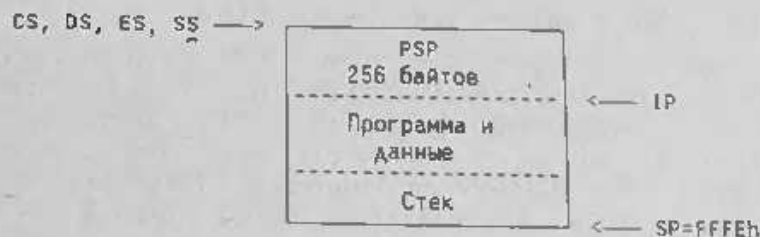


Рис. 2.2. Образ памяти программы .COM.

Процесс подготовки программы на языке ассемблера фирмы Microsoft. Процесс подготовки и отладки программы включает следующие этапы:

- подготовка исходного текста программы с помощью любого текстового редактора. Файл с исходным текстом должен иметь расширение .ASM;
- трансляция программы с помощью ассемблера MASM.EXE с целью получения объектного файла;
- компоновка объектного модуля с помощью компоновщика LINK.EXE с целью получения загрузочного (исполнимого) файла;

- отладка готовой программы с помощью интерактивного отладчика CodeView (файл CV.EXE).

При выборе редактора для подготовки исходного текста программы следует иметь в виду, что многие текстовые процессоры (например, Microsoft Word) добавляют в выходной файл служебную информацию. Поэтому следует воспользоваться редактором, выводящим в выходной файл "чистый текст", без каких-либо управляющих символов. К таким редакторам относятся, например, широко распространенные у нас Лексикон, Notepad Editor и др. Если файл с исходным текстом программы назван P.ASM, то строка вызова ассемблера может иметь следующий вид:

```
MASM /Z /ZI /N P,P,P;
```

Ключ /Z разрешает вывод на экран строк исходного текста программы, в которых ассемблер обнаружил ошибки (без этого ключа поиск ошибок пришлось бы проводить по листингу трансляции).

Ключ /ZI управляет включением в объектный файл номеров строк исходной программы и другой информации, не требуемой при выполнении программы, но используемой отладчиком CodeView.

Ключ /N подавляет вывод в листинг перечня символических обозначений в программе, от чего несколько уменьшается информативность листинга, но существенно сокращается его размер.

Стоящие далее параметры определяют имена модулей: исходного (P.ASM), объектного (P.OBJ) и листинга (P.LST). Точка с запятой подавляет формирование файла P.CRF с перекрестными ссылками.

Строка вызова компоновщика может иметь следующий вид:

```
LINK /CO P,P;
```

Ключ /CO передает в загрузочный файл символическую информацию, позволяющую отладчику CV выводить на экран полный текст исходной программы, включая метки, комментарии и проч. Стоящие далее параметры обозначают имена модулей: объектного (P.OBJ) и загрузочного (P.EXE). Точка с запятой подавляет формирование файла с листингом компоновки (P.MAP) и использование библиотечного файла с объектными модулями подпрограмм.

Компоновщик создает загрузочный модуль в формате .EXE. Если исходная программа написана в формате .COM, то после трансляции и компоновки обычным образом ее надо преобразовать в файл типа .COM. Для этого используется включенная в состав DOS внешняя команда EXE2BIN:

```
EXE2BIN P P.COM
```

Первый параметр обозначает исходный для команды EXE2BIN загрузочный файл P.EXE, второй - ожидаемый результат преобразования. Указание расширения .COM во втором параметре обязательно, так как по умолчанию команда EXE2BIN создаст файл с расширением .BIN.

Использование интерактивного отладчика CodeView Microsoft. Интерактивный отладчик CodeView позволяет выполнять программу по шагам или с точками останова, выводить на экран содержимое регистров и областей памяти, модифицировать (в известных пределах) загруженную в память программу, принудительно изменять содержимое регистров и выполнять другие действия, позволяющие в наглядной и удобной форме отлаживать программы, написанные на языке ассемблера. Отладчик запускается командой

```
CV P.EXE
```

где P.EXE - имя файла с отлаживаемой программой. В процессе работы отладчик использует также файл с исходным модулем P.ASM.

Основные команды отладчика CodeView Microsoft приведены в Приложении 5.

### 2.3. Задачи по моделям памяти и структурам программ

Задача 2.1. Подготовка простейшей программы типа .EXE и изучение ее структуры. Создайте файл с приведенным ниже текстом программы. Оттранслируйте и скомпонуйте его, получив файл с расширением .EXE. Убедитесь в том, что программа нормально запускается и выполняет предусмотренные в ней действия.

Запустите программу под управлением отладчика CodeView. Освойте основные правила работы с отладчиком (наблюдение текста программы и содержимого регистров, выполнение программы целиком и по частям с точками останова, рестарт программы с целью повторного выполнения). Изучите расположение сегментов программы в памяти, сопоставьте полученные результаты с размерами сегментов, полученными с помощью листинга трансляции.

```
;Подготовка программы:
;MASM /Z /ZI /N P,P,P;
;LINK /CO P,P;
```



```

;Программные строки:
text segment 'code'
    assume cs:text, ds:data

;Определения
stdout=1
cr=13
lf=10

myproc proc
    mov AX,data
    mov DS,AX
    ;Выведен на экран строку текста
    mov AH,40h
    mov BX,stdout
    mov CX,mes len
    mov DX,offset mes
    int 21h
;Завершим программу
    mov AX,4C00h
    int 21h
myproc endp
text ends

;Поля данных:
data segment
    mes db 'Программа .EXE стартовала',cr,lf
    mes len equ $-mes
data ends
stack segment para stack 'stack'
    db 128 dup (?)
stack ends
end myproc

```

Задача 2.2. Подготовка простейшей программы типа .COM и изучение ее структуры. Задание аналогично предыдущему примеру.

```

;Подготовка программы:
MASM /Z /ZI /N P,P,P;
LINK /CO P,P;
EXE2BIN P,EXE P.COM
;Программные строки:
text segment 'code'
    org 100h
    assume cs:text, ds:text

;Определения
stdout=1
cr=10
lf=13

myproc proc
    ;Выведен на экран строку текста
    mov AH,40h
    mov BX,stdout
    mov CX,mes len

```

```

    mov DX,offset mes
    int 21h
;Завершим программу
    mov AX,4C00h
    int 21h
myproc endp
;Поля данных:
mes db 'Программа .COM стартовала',cr,lf
mes len equ $-mes
text ends
end myproc

```

### 3. Основы языка ассемблера

3.1. Основные определения данных

Для заполнения последовательных байтов, слов или двойных слов памяти, а также для резервирования памяти служат директивы ассемблера (псевдооператоры) DB, DW и DD, а для резервирования массивов - директива DUP:

```
count    dw      1000      ;В слово с именем count
                           ;помещается число 1000
coeffs   dw      5,12000,0,168 ;Указанными числами заполняются 4
                           ;последовательные слова памяти.
                           ;Первому слову дается имя coeffs
mask     db      0FFh      ;В байт именем mask помещается
                           ;16-ричное число FFh
mes      db      '---Внимание!--' ;Строка байтов с именем mes
                           ;заполняется указанным текстом
addr     dw      mes        ;В слово с именем addr помещается
                           ;смещение первого байта строки mes
addrd    dd      myproc     ;В двойное слово с именем addrd
                           ;помещается двухсловный адрес
                           ;процедуры myproc (в первое слово
                           ;относительный адрес, во второе -
                           ;сегментный адрес)
area     dw      128 dup (?) ;Резервируется область памяти
                           ;объемом 128 слов
testmes  db      100 dup ('#') ;Строка с именем testmes
                           ;заполняется кодом символа #
array    dw      1024 dup (256) ;Массив из 1024 слов
                           ;заполняется числом 256
```

Для определения констант служат директива EQU и знак равенства:

```
kilo     equ      1024      ;Числу 1024 присваивается
                           ;символическое имя kilo
offs=80*2*12+40*2         ;Константа offs получает значение,
                           ;равное результату вычисления
                           ;указанного выражения
length=0FFFFh             ;Числу 0FFFFh=65535 присваивается
                           ;символическое имя length
```

Отличие директивы equ от знака равенства заключается в том, что директива equ присваивает символическим именам значения "раз и навсегда", а при использовании знака равенства значения констант можно изменять:

```
counter=0      ;Символическое имя counter принимает значение 0
counter=counter+1 ;Символическое имя counter принимает значение 1
```

Для ссылок на текущую ячейку используется обозначение счетчика текущего адреса \$:

```
mes      db      'Идите'
mes_len=$-mes
```

В этом примере константа mes-len получает значение длины строки mes (в данном случае 5 байтов), которая вычисляется как разность значения счетчика текущего адреса после определения строки и ее начального адреса mes.

Данные, описанные в программе, должны участвовать в операциях в соответствии со своими описаниями. Ассемблер фиксирует ошибку если, например, данное, описанное как байт, участвует в операции со словом. Во многих случаях, однако, требуются как раз такие операции. Для их реализации предусмотрен атрибутивный оператор PTR:

```
bits     dw      0F5E9h      ;Данное описано как слово
mov       AX,bits           ;Операция соответствует описанию
mov       BH,byte ptr bits  ;Из слова bits забирается
                           ;младший байт (число E9h)
mov       CL,byte ptr bits+1 ;Из слова bits забирается
                           ;старший байт (число F5h)
addr     dd      myproc      ;Данное описано как двойное слово
mov       BX,word ptr addr   ;Забираем младшее слово
                           ;(относительный адрес myproc)
mov       ES,word ptr addr+2 ;Забираем старшее слово
                           ;(сегментный адрес myproc)
```

### 3.2. Режимы адресации

Режимом, или способом адресации называют процедуру нахождения операнда. Различают следующие режимы адресации.

Регистровый. Операнд (байт или слово) находится в регистре. Этот способ адресации применим ко всем программно-адресуемым регистрам процессора.

```
inc       CX              ;Увеличение на 1 содержимого AX
push      DS              ;Сегментный адрес сохраняется
                           ;в стеке
xchg      BX,BP           ;Регистры BX и BP обмениваются
                           ;содержимым
mov       ES,AX           ;Содержимое AX пересылается в ES
```

Непосредственный. Операнд (байт или слово) указывается в команде; он может иметь любой смысл (число, адрес, код ASCII), а также быть представлен в виде символического обозначения.

mov	AL, 40h	; Число 40h загружается в AL
mov	AL, '*'	; Код ASCII символа '*' загружается в AL
int	21h	; Команда прерывания типа 21h
equ	52h	; Число 52h получает обозначение limit
mov	CX, limit	; Число, обозначенное limit, загружается в CX

Важным применением непосредственной адресации является пересылка относительных адресов (смещений). Чтобы указать, что речь идет об относительном адресе данной ячейки, а не об ее содержимом, используется описатель OFFSET:

mov	DI, offset mes	; Строка символов
mov	DX, offset mes	; Адрес строки засылается в DX

В приведенном примере относительный адрес строки mes, т.е. расстояние в байтах первого байта этой строки от начала сегмента, в котором она находится, заносится в регистр DX.

Прямой. Адресуется память; адрес ячейки памяти (слова или байта) указывается в команде:

mov	DI, 0	; Резервируется слово памяти (и в него засылается 0)
inc	mem	; Содержимое этого слова увеличивается на 1
mov	DX, mem	; Содержимое слова с именем mem загружается в регистр DX

Строго говоря, процессору следует передать информацию о том, с помощью какого сегментного регистра определять адрес:

inc	DS:mem
mov	DX, DS:mem

Однако, как уже отмечалось, по умолчанию все смещения вычисляются относительно DS, поэтому в данном случае это указание сегментного регистра избыточно. В тех же случаях, когда рассматриваемая ячейка находится в сегменте, адресуемом через какой-либо другой сегментный регистр (ES, CS или SS), указание сегментного регистра является обязательным:

inc	ES:mem1
-----	---------

Часто бывает нужно обращаться к памяти по известному абсолютному адресу. В этом случае указание сегментного регистра обязательно:

mov	AL, DS:17h	; Загрузка AL из ячейки с адресом 17h относительно DS
mov	BX, ES:2Ch	; Загрузка BX из ячейки с адресом 2Ch относительно ES

При обращении по абсолютным адресам константа, определяющая адрес, может быть заключена в квадратные скобки. Приведенные ниже команды эквивалентны предыдущим:

mov	AL, DS:[17h]
mov	BX, ES:[2Ch]

Следует подчеркнуть, что при любом обращении к памяти процессор обязательно использует один из сегментных регистров. Перед выполнением команды обращения к памяти в используемый сегментный регистр следует загрузить требуемый сегментный адрес. Пусть, например, мы хотим прочитать код символа, изображаемого в настоящий в первой позиции экрана. Известно, что текстовый видеобuffer начинается с адреса B8000h. Для того, чтобы настроить на этот адрес сегментный регистр, в него следует загрузить число, характеризующее номер параграфа, с которого начинается видеобuffer, т.е. B8000h/16=B800h:

mov	AX, 0B800h	; Сегментный адрес видеобufferа
mov	ES, AX	; отправим в ES
mov	AL, ES:[0]	; Прочитаем первый байт видеобufferа

Регистровый косвенный (базовый или индексный). Адресуется память (байт или слово). Относительный адрес операнда находится в регистрах BX или BP (базовая адресация) или в регистрах SI или DI (индексная адресация). При использовании регистров BX, SI и DI подразумевается сегмент, адресуемый через DS; при использовании BP подразумевается сегмент стека и, соответственно, регистр SS. Допускается замена сегмента. Обозначение этого способа адресации:

[BX]	{подразумевается DS:[BX]}
[BP]	{подразумевается SS:[BP]}
[SI]	{подразумевается DS:[SI]}
[DI]	{подразумевается DS:[DI]}

Регистровый косвенный способ адресации удобно использовать в тех случаях, когда к некоторой ячейке памяти приходится обращаться многократно:

mov	SI, offset cells	; Относительный адрес ячейки cells
		; загружается в SI



mov	AX, [SI]	: Содержимое ячейки cells
		: загружается в AX
inc	[SI]	: Инкремент содержимого ячейки
cells	mov	BX, [SI]
		: Новое содержимое ячейки cells
		: загружается в BX

Регистровый косвенный со смещением (базовый или индексный). Адресуется память (байт или слово). Относительный адрес операнда определяется как сумма содержимого регистра BX, BP, SI или DI и указанной в команде константы, называемой смещением. Смещение может быть числом или адресом. При использовании регистров BX, SI и DI подразумевается сегмент, адресуемый через DS; при использовании BP — сегмент стека и, соответственно, регистр SS. Допускается замена сегмента. Обозначение этого способа адресации:

смещение [BX] (подразумевается DS:смещение [BX])  
 смещение [BP] (подразумевается SS:смещение [BP])  
 смещение [SI] (подразумевается DS:смещение [SI])  
 смещение [DI] (подразумевается DS:смещение [DI])

Допустимы также обозначения (со всеми регистрами) вида:

[BX]+смещение  
 [BX+смещение]

Пусть в сегменте данных определен массив из 10 чисел:

array db 0, 10, 20, 30, 40, 50, 60, 70, 80, 90

Последовательность команд

mov BX, 5  
 mov AL, array[BX]

загрузит в регистр AL элемент массива с индексом 5, т.е. число 50. Тот же результат можно получить, загрузив в BX не индекс, а адрес массива:

mov BX, offset array  
 mov AL, 5[BX]

Другие варианты последней команды:

mov AL, [BX]+5  
 mov AL, [BX+5]

Базовый индексный. Адресуется память (байт или слово). Относительный адрес операнда определяется как сумма содержимого следующих пар регистров:

[BX][SI] (подразумевается DS:[BX][SI])  
 [BX][DI] (подразумевается DS:[BX][DI])

[BP][SI] (подразумевается SS:[BP][SI])  
 [BP][DI] (подразумевается SS:[BP][DI])

Допускается замена сегмента.

Пусть в сегменте данных определен массив из 10 слов:

words dw 0, 10, 20, 30, 40, 50, 60, 70, 80, 90

Последовательность команд

mov BX, offset words  
 mov SI, 10  
 mov AX, [BX][SI]

загрузит в регистр AX слово со смещением 10 байтов от начала массива, т.е. число 50.

Базовый индексный со смещением. Адресуется память (байт или слово). Относительный адрес операнда определяется как сумма содержимого двух регистров и смещения. Обозначение этого способа адресации:

смещение [BX][SI] (подразумевается DS:смещение [BX][SI])  
 смещение [BX][DI] (подразумевается DS:смещение [BX][DI])  
 смещение [BP][SI] (подразумевается SS:смещение [BP][SI])  
 смещение [BP][DI] (подразумевается SS:смещение [BP][DI])

Допустимы также обозначения (со всеми регистрами) вида:

смещение[BX+SI]  
 [смещение+BX+DI]  
 [BP][SI]+смещение

Пусть в сегменте данных определен массив из 24 байтов:

syms db 'ИЦУКЕНГЩЗХЬ'  
 db 'QWERTYUIOP{'

Последовательность команд

mov BX, 12  
 mov SI, 6  
 mov DL, syms[BX][SI]

загрузит в регистр DL элемент с индексом 6 из второго ряда, т.е. код ASCII буквы U. Тот же результат можно получить, загрузив в один из регистров не индекс, а адрес массива:

mov BX, offset sym  
 mov SI, 6  
 mov DL, 12[BX][SI]

### 3.3. Основы программирования на языке ассемблера

Циклы. Для организации многократного выполнения некоторого блока команд используется команда LOOP метка, которая

передает управление на указанную метку столько раз, каково содержимое регистра CX. В следующем примере таким образом очищается массив из 1024 чисел.

```

;Сегмент данных
;Сегмент кода
;Массив, требующий очистки
1024 dup (?)
;Счетчик повторений
;Адрес массива в BX
;Указатель в массиве
;Очистка элемента массива
;Смещение указателя
;к следующему элементу
;Команда цикла
push    CX
mov     BX, 1024
lea     SI, 0
mov     [BX][SI], 0
inc     SI
loop    null

```

При необходимости организовать вложенные циклы удобно воспользоваться стеком для сохранения внешнего цикла на время выполнения внутреннего. В следующем примере организуется программная задержка длительностью несколько секунд (величина задержки зависит от типа компьютера).

```

;Внешний счетчик повторений
;Сохраним его в стеке
;64K шагов во внутреннем цикле
;Тело внутреннего цикла -
;всего 1 строка
;Восстановим CX перед командой
;loop внешнего цикла
push    CX
push    CX
mov     CX, 0
inner:  loop inner
pop     CX
outer:  loop outer

```

**Обработка строк (цепочек) байтов или слов.** Для работы со строками символов или чисел (т.е. по существу с массивами) предусмотрен ряд специальных команд:

MOVSB Пересылка строки  
CMPS Сравнение строк  
SCAS Поиск в строке заданного элемента (сканирование)  
LODS Загрузка регистра AX или AL из строки  
STOS Запись элемента строки из регистра AX или AL

Команды имеют общие черты: они выполняются в предположении, что адрес строки-источника находится в регистрах DS:SI, а адрес строки-приемника в ES:DI; при однократном выполнении они обрабатывают только один элемент, а для обработки строки должны предваряться префиксом повторения; в процессе обработки элементов строки регистры SI и DI автоматически смещаются по строке вперед (если DF=0) или назад (если DF=1); каждая команда имеет модификации для работы с байтами или словами (например MOVSB и MOVSW).

В следующем примере символьная строка str1 пересылается в другое место памяти где, возможно, несколькими операциями пересылки формируется текст для вывода на экран.

```

str1    db      'Включите режим номер ' ;Пересылаемая строка
str1len equ     $-str1 ;Длина пересылаемой строки
text    db      80 dup(' ') ;Приемная строка
        mov     CX, str1len ;Столько байтов переслать
        push    DS ;Настроим ES
        pop     ES ;на наш сегмент данных
        lea     SI, str1 ;Теперь DS:SI -> строка-источник
        lea     DI, text ;Теперь ES:DI -> строка-приемник
        cld     ;Двигаться по строке вперед
rep     movsb ;Пересылаем CX байтов

```

Префикс повторения REP заставляет процессор выполнить команду movsb число раз, соответствующее содержимому регистра CX, т.е. в данном случае str1len раз.

В следующем примере выполняется сравнение двух строк. Предполагается, что в буфер TEXT поступают строки исходного текста программы, в которых мы ищем обозначение stringlen.

```

patrn    db      'stringlen' ;Строка для сравнения
patlen   equ     $-patrn ;Длина строки
text     db      80 dup(' ') ;Приемный буфер
        push    DS ;Настроим ES
        pop     ES ;на наш сегмент данных
        lea     SI, patrn ;Теперь DS:SI -> строка-источник
        lea     DI, text ;Теперь ES:DI -> строка-приемник
        mov     CX, patlen ;Длина строки
        cld     ;Выполнять сравнение вперед
repe     cmpsb ;Сравнение байтов пока они равны,
               ;не более CX раз
        jne     notequ ;Строки не совпадают, перейти
               ;на метку notequ
        ;Строки совпадают
        ...
notequ:   ;Строки не совпадают
        ....

```

Префикс повторения REPE заставляет процессор выполнять сравнение последовательных байтов строк, пока эти байты равны, т.е. до тех пор, пока не обнаружится пара различающихся байтов. Если все байты оказались попарно одинаковыми, сравнение выполняется CX раз, и после завершения цикла сравнения флаг нуля ZF устанавливается в 1. Если же в какой-то паре байты оказались разными, цикл сравнения заканчивается, а флаг ZF устанавливается в 0 (отсутствие равенства операндов). Команда JNE (или JNZ) позволяет проанализировать результат сравнения (в сущности, последнего байта) и перейти на соответствующую метку при обнаружении неравенства строк. В случае равенства всех CX байтов команда JNE не срабатывает и выполняются следующие за JNE строки программы.

Переход к следующим байтам строки осуществляется в результате автоматического инкремента содержимого регистров SI и DI. Поэтому после завершения сравнения и SI, и DI указывают (каждый в своей строке) на байты, следующие за теми, на которых закончилась операция сравнения, т.е. на байты, следующий за различающимися байтами (например, прочитать их или заменить), следует выполнить декременты регистров SI и DI.

В следующем примере выполняется поиск в строке заданного символа, например, кода пробела (т.е. промежутка между словами).

str	db	128 dup (?)		: Строка с текстом
	mov	AL, "		: Искомый символ
	push	DS		: Настроим ES
	pop	ES		: на наш сегмент данных
	lea	DI, str		: ES:DI -> строка
	mov	CX, 128		: Верхняя граница поиска
repne	scasb			: Сравнение с AL пока не равно
	jne	not_found		: Так и не было пробела
	dec	DI		: DI -> пробел в строке

Здесь поиск осуществляется, пока байты строки не равны искомому символу, находящемуся в регистре AL (в данном примере коду пробела). Как только в строке обнаружится код пробела, поиск заканчивается и устанавливается флаг ZF. Всего просматривается не более CX байтов строки. Если соответствие не обнаруживается, флаг ZF=0. После завершения операции просмотра регистр DI указывает на байты, следующий за тем, на котором закончилась операция, т.е. на байт, следующий за первым пробелом в строке.

Переходы. Команда безусловного перехода JMP может использоваться в 5 разновидностях. Переход может быть:

- прямым коротким (в пределах -128...+127 байтов);
- прямым ближним (в пределах текущего сегмента команд);
- прямым дальним (в другой сегмент команд);
- косвенным ближним (в пределах текущего сегмента команд через ячейку с адресом перехода);
- косвенным дальним (в другой сегмент команд через ячейку с адресом перехода).

Рассмотрим структуру программы с переходами разного вида.

#### 1. Прямой короткий (short) переход.

```
seg      segment 'code'
...
jmp      short cont      ;Код EB dd
...
```

```
cont:
...
seg      ends
```

Метка cont должна отстоять от команды, следующей за JMP, не более, чем на 128 байтов назад или на 127 байтов вперед. Если cont стоит в программе до команды JMP, описатель SHORT можно опустить. В приведенном коде команды dd обозначает байт с величиной относительного смещения к точке перехода от команды, следующей за командой JMP.

#### 2. Прямой ближний (near), или внутрисегментный переход.

```
seg      segment 'code'
...
jmp      near cont      ;Код E9 dddd
...
cont:
...
seg      ends
```

Метка cont может находиться в любом месте сегмента команд, как до, так и после команды JMP. В коде команды dddd обозначает слово с величиной относительного смещения к точке перехода от команды, следующей за командой JMP.

В некоторых случаях удобно пользоваться другой формой команды ближнего перехода:

```
jmp      near ptr cont
```

в которой в явной форме указывается, что переход должен быть ближним (в том же сегменте команд).

#### 3. Прямой дальний (far), или межсегментный переход.

```
seg1     segment 'code'
...
jmp      far ptr cont      ;Код EA dddd ssss
...
seg1     ends
seg2     segment 'code'
...
cont:
...
seg2     ends
```

Метка cont находится в другом сегменте команд этой двухсегментной программы. В коде команды ssss - сегментный адрес сегмента seg2, а dddd - относительный адрес точки перехода cont в сегменте команд seg2 (относительно начала этого сегмента).

#### 4. Косвенный ближний (внутрисегментный) переход.

```
seg      segment 'code'
...
...
```



38

```

    jmp     DS:contadr      ;Код FF 26 dddd
                           ;Точка перехода
cont:
    ...
seg1      ends
dat       segment
    ...
contadr   dd     cont      ;Адрес перехода (слово)
dat       ends

```

Точка перехода `cont` может находиться в любом месте сегмента команд, как до, так и после команды `JMP`. В коде команды `dddd` обозначает относительный адрес слова `contadr` в сегменте данных, содержащем эту ячейку. Такой способ перехода удобен тем, адрес перехода может быть вычислен программно и помещен в ячейку `contadr` по ходу выполнения программы.

В некоторых случаях удобно пользоваться другой формой команды косвенного перехода:

```

    jmp     word ptr contadr

```

Таким образом, и указание сегментного регистра, и описатель `WORD PTR` придают команде косвенность (переход осуществляется не на метку `contadr`, а по адресу, содержащемуся в ячейке `contadr`).

### 5. Косвенный дальний (межсегментный) переход.

```

seg1      segment 'code'
    ...
    jmp     DS:contadr      ; Код FF 2E dddd
    ...
seg1      ends
seg2      segment 'code'
    ...
cont:     ...
    ...
seg2      ends
dat       segment
    ...
contadr   dd     cont      ;Двухсловный адрес точки перехода
dat       ends

```

Точка перехода `cont` находится в другом сегменте команд этой двухсегментной программы. В коде команды `dddd` обозначает относительный адрес слова `contadr` в сегменте данных. Ячейка `contadr` объявляется директивой `DD` и содержит двухсловный адрес точки перехода - в первом слове смещение `cont` в сегменте команд `seg2`, во втором слове сегментный адрес

39

`seg2`. Обе компоненты адреса перехода могут быть вычислены и помещены в ячейку `contadr` по ходу выполнения программы.

В некоторых случаях удобно пользоваться другой формой команды косвенного перехода:

```

    jmp     dword ptr contadr

```

Здесь описатель `DWORD PTR` говорит о том, что адрес хранится в двухсловной ячейке и, следовательно, осуществляется дальний косвенный переход (в другой сегмент).

**Вызовы подпрограмм.** Команда вызова подпрограммы `CALL` может использоваться в 4 разновидностях. Вызов может быть:

- прямым ближним (в пределах текущего сегмента команд);
- прямым дальним (в другой сегмент команд);
- косвенным ближним (в пределах текущего сегмента команд через ячейку с адресом перехода);
- косвенным дальним (в другой сегмент команд через ячейку с адресом перехода);

Программа обычно оформляется в виде процедуры, завершающейся командой возврата из подпрограммы `RET`, однако это не обязательно. В качестве подпрограммы может служить любой участок программы, заканчивающийся командой `RET`. Начало этого участка может быть обозначено меткой, которая дает возможность организовать прямой вызов на подпрограмму. В случае косвенного вызова, когда вычисленный каким-то образом адрес перехода заносится в выделенную для этого ячейку, наличие метки в точке входа в подпрограмму не обязательно.

Рассмотрим структуру программных комплексов с подпрограммами.

### 1. Прямой ближний вызов.

```

seg       segment 'code'
mymain    proc                                ;Основная программа
    ...
    call   sub                                ;Код E8 dddd
    ...
mymain    endp
sub        proc    near                        ;Подпрограмма
    ...
    ret
sub        endp
seg        ends

```

Процедура-программа находится в том же сегменте команд, что и вызывающая программа. В коде команды `dddd` обозначает смещение в сегменте команд к точке входа в подпрограмму (не обязательно оформляемую в виде процедуры). При выполнении команды `CALL` процессор помещает адрес возврата

(содержимое регистра IP) в стек выполняемой программы, после чего к текущему содержимому IP прибавляет dddd. В результате в IP оказывается адрес подпрограммы. Команда RET выполняет обратную процедуру - заносит адрес возврата в IP.

## 2. Прямой дальний вызов.

```

seg1      segment 'code'           ;Основная программа
mymain    proc                     ;Код 9A dddd ssss
...
call      far ptr sub
...
mymain    endp
seg1      ends
seg2      segment 'code'           ;Подпрограмма
sub       proc                     ;Код CB
...
ret
sub2      endp
seg2      ends

```

Процедура-подпрограмма находится в другом сегменте команд той же программы. В коде команды dddd обозначает относительный адрес точки входа в подпрограмму в ее сегменте команд, а ssss - ее сегментный адрес. При выполнении команды CALL процессор помещает в стек сначала сегментный адрес вызываемой программы, а затем адрес возврата (содержимое IP). В сегментный регистр CS заносится ssss, а в IP - dddd. Поскольку процедура-подпрограмма атрибутом FAR объявлена, команда RET имеет код, отличный от кода аналогичной команды ближней процедуры и выполняется по-другому: из стека извлекаются два верхних слова и переносятся в IP и CS, чем и осуществляется возврат в вызывающую программу, находящуюся в другом сегменте команд.

Вместо того, чтобы объявлять процедуру дальней, можно вместо команды RET указать в явной форме команду дальнего возврата из подпрограммы RETF.

## 3. Косвенный ближний вызов.

```

seg      segment 'code'           ;Основная программа
mymain   proc
...
call     word ptr subadr          ;Код FF 16 dddd
...
mymain   endp
sub      proc near                ;Подпрограмма
...
ret      ;Код C3
sub      endp
seg      ends
dat      segment
...

```

```

subadr   dw      sub              ;Ячейка с адресом подпрограммы
...
dat      ends

```

Процедура-программа с атрибутом NEAR находится в том же сегменте, что и вызывающая программа, а ее относительный адрес в ячейке subadr в сегменте данных. В коде команды dddd обозначает относительный адрес слова subadr в сегменте данных. Второй байт кода команды (16 в данном примере) зависит от способа адресации. Косвенный вызов позволяет использовать разнообразные способы адресации подпрограммы:

```

call     word ptr [BX]           ;В BX адрес подпрограммы
call     word ptr [BX] [SI]      ;В BX адрес таблицы адресов
...                               ;подпрограмм, в SI индекс
...                               ;в этой таблице.
call     word ptr table [SI]     ;table - адрес таблицы адресов
...                               ;подпрограмм, в SI индекс в этой
...                               ;таблице.

```

## 4. Косвенный дальний вызов.

```

seg1     segment 'code'
mymain   proc                     ;Основная программа
...
call     dword ptr subadr        ;Код FF 1E dddd
...
mymain   endp
seg1     ends
seg2     segment 'code'           ;Подпрограмма
sub      proc far                ;Код CB
...
ret
sub      endp
seg2     ends
dat      segment
...
subadr   dd      sub              ;Двухсловная ячейка с адресом
...                               ;подпрограммы
dat      ends

```

Процедура-подпрограмма с атрибутом FAR находится в другом сегменте команд той же программы, а ее полный двухсловный адрес (сегментный адрес и смещение) - в ячейке subadr в сегменте данных. Второй байт кода команды (1E в данном примере) зависит от способа адресации. Косвенный дальний вызов, как и косвенный ближний, позволяет использовать разнообразные способы описания адреса подпрограммы (см. п. 3).

Многомодульные программы. При разработке сложных программных комплексов бывает удобно часть процедур выделить в отдельный исходный модуль, транслируемый самостоятельно.

После трансляции объектные модули этих процедур могут войти в состав объектной библиотеки, подсоединяемой к основной программе на этапе компоновки. Структура такого программного комплекса может выглядеть следующим образом:

```

;MAIN.ASM - файл с главной процедурой
text
segment public 'code'
extrn sub:proc
;sub - внешняя ссылка

mymain proc
...
call sub
;Прямой ближний вызов

mymain endp
text and mymain
;mymain - точка входа в главную
;процедуру

;MYSUB.ASM - файл с подпрограммой
text
segment public 'code'
sub ;sub - процедура "общего пользования"
;Ближняя процедура
proc near

...
ret
;без точки входа

sub endp
text and
end

```

Программные сегменты обоих исходных модулей имеют одно имя text, что обеспечивает их слияние компоновщиком в единый сегмент. Тип объединения PUBLIC определяет способ слияния - конкатенацию, т.е. подсоединение второго модуля к концу первого (так же действует описатель STACK, используемый в сегментах стеков, в то время как тип объединения COMMON требует от компоновщика наложения одноименных сегментов друг на друга).

Директива EXTRN в главном модуле определяет, что символическое обозначение sub является внешней ссылкой и представляет собой имя процедуры. Внешняя ссылка будет разрешена на этапе компоновки.

Процедуры, вызываемые из других программных модулей, должны быть определены как процедуры "общего пользования" с помощью директивы PUBLIC. В этом случае их имена записываются транслятором в объектный модуль, что позволяет компоновщику разрешить ссылки на них.

В приведенном примере предполагается, что программные модули после слияния образуют один сегмент команд, т.е. что их суммарный размер (в машинных кодах) не превышает 64 Кбайт. Поэтому процедура-подпрограмма объявлена ближней (near), а в главной процедуре использована команда прямого ближнего вызова.

Модуль с подпрограммой заканчивается директивой завершения трансляции END без параметра - в многомодульных программных комплексах только тот модуль, который содержит главную процедуру, оканчивается директивой END с указанием входной точки этой процедуры.

Файлы MAIN.ASM и MYSUB.ASM с исходными текстами компонентов комплекса следует порознь оттранслировать, а затем скомпоновать в единый загрузочный модуль:

```

MASM MAIN, MAIN, MAIN;
MASM MYSUB, MYSUB, MYSUB;
LINK MAIN + MYSUB, PROG;

```

В результате трансляции будут получены файлы MAIN.OBJ, MAIN.LST, MYSUB.OBJ и MYSUB.LST; компоновщик объединит модули MAIN.OBJ и MYSUB.OBJ и образует загрузочный (выполнимый) модуль PROG.EXE.

Другой вариант процедуры разработки многомодульного программного комплекса - помещение объектного модуля MYSUB.OBJ в библиотеку объектных модулей с последующим извлечением его оттуда компоновщиком. Библиотекой целесообразно пользоваться при наличии большого числа объектных модулей, используемых в различных программных комплексах. В этом случае после трансляции файла MYSUB.ASM объектный файл MYSUB.OBJ записывается в создаваемую пользователем библиотеку объектных файлов. Включение объектных файлов с подпрограммами во вновь создаваемую объектную библиотеку выполняется следующим образом:

```

LIB MYOBJ.LIB +MYSUB.OBJ, MYOBJ.LST;

```

В этой команде LIB - имя программы библиотекаря, MYSUB.OBJ - имя помещаемого в библиотеку объектного файла, а MYOBJ.LST - имя создаваемого файла с каталогом библиотеки. Знак '+' перед именем модуля обозначает, что этот модуль надо добавить в библиотеку (знак '-' удаляет данный модуль). В файл MYOBJ.LST будет помещено оглавление библиотеки.

Все расширения, кроме расширения файла с каталогом, можно опустить, так как по умолчанию предполагаются именно указанные в примере расширения.

Если объектная библиотека уже имеется, и мы хотим добавить в нее вновь созданные объектные модули, в строке вызова библиотекаря надо в качестве последнего параметра еще раз указать имя библиотеки:

```

LIB MYOBJ +NEW1 +NEW2, MYOBJ.LST, MYOBJ

```



Здесь первый параметр определяет имя исходной библиотеки, а последний - имя создаваемой библиотеки, расширенной за счет добавления новых модулей. Это имя может совпадать с именем старой библиотеки, но может и отличаться от него.

При наличии объектной библиотеки в строке вызова компоновщика в качестве четвертого параметра команды следует указать имя библиотеки:

```
LINK/CO MAIN.OBJ, PROG.EXE, PROG.MAP, MYOBJ.LIB
```

В приведенном примере MAIN.OBJ - объектный файл с основной программой, PROG.EXE - результирующий выполнимый модуль, PROG.MAP - карта выполнимого модуля, а MYOBJ.LIB - объектная библиотека со требуемыми подпрограммами. Как и в предыдущих примерах, все расширения можно опустить, так они предполагаются или назначаются по умолчанию. Карта выполнимого модуля обычно не нужна (в ней мало полезной информации), а имя загрузочного модуля по умолчанию совпадает с именем объектного (при разных расширениях). Поэтому приведенную команду можно упростить:

```
LINK MAIN, PROG., MYOBJ.LIB
```

В результате выполнения этой команды компоновщик создаст выполнимый модуль PROG.EXE.

Если суммарный размер главной программы и подпрограмм (в машинных кодах) превышает 64 Кбайт, загрузочный модуль должен быть многосегментным. Структура такого программного комплекса может выглядеть следующим образом:

```
;MAIN.ASM - файл с главной процедурой
text1 segment 'code'
extrn sub:proc

text1 ends
text segment 'code'
mymain proc
...
call far ptr sub
...
mymain ends
text ends
end mymain

;MYSUB.ASM - файл с подпрограммой
text1 segment 'code'
public sub
sub proc far
...
sub endp
text1 ends
end
```

Поскольку сегменты text (с главной программой) и text1 (с подпрограммой) имеют разные имена, они не будут объединены компоновщиком в один сегмент. В исходном модуле, содержащем вызов подпрограммы, она должна быть описана директивой EXTRN. Однако это описание должно находиться в сегменте, одноименным с сегментом подпрограммы. Поэтому в модуль MAIN.ASM включен пустой пока что сегмент text1, содержащий лишь одну строку - объявление вызываемой процедуры EXTRN SUB:PROC. Далее следует основной сегмент команд text с главной программой. В ней использован прямой дальней вызов; процедура-подпрограмма, в свою очередь, атрибутом FAR объявлена дальней. Трансляция и компоновка выполняется так же, как и в случае многомодульной односегментной программы.

**Макрокоманды.** Программы, написанные на языке ассемблера, часто содержат повторяющиеся участки текста с одинаковой структурой. Такой участок текста можно оформить в виде макроопределения, характеризующегося произвольным именем и необязательным списком формальных аргументов. После того как такое определение сделано, появление в программе строки, содержащей имя макроопределения и список фактических аргументов, приводит к генерации всего требуемого текста, называемого макрорасширением. Варьируя фактические аргументы, можно, сохраняя неизменной структуру макрорасширения, изменить отдельные его элементы.

Макроопределение должно начинаться строкой с именем макроопределения и директивой MACRO, в поле аргументов которой указывается список формальных аргументов. Заканчивается макроопределение директивой ENDM.

Пусть в программе требуется неоднократно сохранять в стеке содержимое трех регистров, но в каждом конкретном случае номера регистров и их порядок отличаются. Оформим эти действия в виде макроопределения:

```
psh macro a,b,c
push a
push b
push c
endm
```

Появление в исходном тексте программы строки

```
psh AX, BX, CX
```

приведет к генерации следующего фрагмента текста:

```
push AX
push BX
push CX
```

Если же в исходном тексте имеется строка

```
psh    DX, ES, BP
```

то соответствующее макрорасширение будет иметь вид:

```
push    DX
push    ES
push    BP
```

В качестве фактических аргументов могут выступать любые обозначения ассемблера, допустимые для данной команды. В частности, макровывод

```
psh mes, [BX], ES:[17]h
```

приведет к следующему макрорасширению:

```
push    mes
push    [BX]
push    ES:[17h]
```

Если какие-то строки макроопределения должны быть помещены (например, с целью организации циклов), то обозначения меток должны быть объявлены локальными с помощью оператора LOCAL. В этом случае ассемблер, генерируя макрорасширения, будет создавать собственные обозначения меток, не повторяющиеся при повторных вызовах одной и той же макрокоманды:

```
delay    macro    point
          local    CX, 20000
point:    loop     point
          endm
```

Текст макроопределения можно включить непосредственно в текст программы, однако в тех случаях, когда макрокоманды описывают некоторые стандартные процедуры широкого назначения, например, программную задержку или вывод на экран строки текста, целесообразно тексты макроопределений поместить в макробиблиотеку.

Макробиблиотека представляет собой файл с текстами макроопределений. Макроопределения записываются в этот файл точно в таком же виде, как и в текст программы. Ниже приведен текст файла макробиблиотеки с произвольным именем MYMACRO.MAC, содержащей две макрокоманды.

;Макрокоманда outprog завершения программы

```
outprog macro
mov     AX, 4C00h
int     21h
endm
```

;Макрокоманда delay небольшой программной задержки

```
delay    macro
          local    point
          mov     CX, 20000
point:    loop     point
          endm
```

### 3.4. Задачи по программированию на языке ассемблера

Полный список команд МП 8086 с пояснениями и примерами приведен в Приложении 4.

**Задача 3.1. Циклы.** Создать тестовый символьный массив (64 символа, от символа пробела до символа —). Вывести созданную строку на экран.

;Основные фрагменты программы

;Заполним строку

```
mov     CX, 64
mov     AL, ' '
mov     SI, 0
fill:   mov     mes[SI], AL
inc     SI
inc     AL
loop    fill
```

;Выведем строку mes на экран

```
mov     AH, 4Ch
mov     BX, 1
mov     CX, 64
mov     DX, offset mes
int     21h
```

;Завершим программу

...

;Поля данных

```
mes     db      64 dup('—')
```

;Число символов

;Первый символ — пробел

;Смещение в строке

;Перемещаем байт в строку

;К следующему байту

;Следующий символ

;Цикл CX раз

;Файловая функция вывода

;Дескриптор стандартного вывода

;Столько байтов вывести

;Адрес строки

;Переход в MS-DOS

;Строка с контрольным содержанием

**Задача 3.2. Циклы.** Просмотреть изображения символов второй половины кодовой таблицы (коды 128 - 255, всего 128 символов). Для этого создать тестовый символьный массив, состоящий из кодов этих символов, и вывести его на экран.

**Задача 3.3. Вложенные циклы.** Создать программную задержку. Определить значения параметров программы, позволяющие получить задержки (на конкретной машине) 10, 3 и 1с.

;Основные фрагменты программы

;Организуем (для наглядности) демонстрационный

;цикл из 10 шагов

```
mov     CX, 10
loop4:  push    CX
```

;Выведем строку mes на экран

...

```

;Организуем программную задержку
mov     CX,time
outer:  push    CX,0
        mov     inner
inner:  pop     CX
        loop   outer
        pop     CX
        loop   loop4

;Число шагов внешнего цикла
;Сохраним его в стеке
;64K шагов во внутреннем цикле
;Тело цикла - всего 1 строка
;Восстановим CX перед командой
;loop внешнего цикла outer
;Восстановим CX демонстра-
;ционного цикла loop4

;Демонстрационная строка
;Настройка времени задержки
;Поля данных
mes     db     '<>'
time    dw     10

```

**Задача 3.4. Пересылка строк.** Переслать строку (массив) произвольной длины, включив ее в состав другой, более длинной строки. Вывести полученную строку на экран.

```

;Основные фрагменты программы
;Перешлем строку
mov     CX, strlen
        mov     AX, DS
        mov     ES, AX
        mov     SI, offset str
        mov     DI, mes+3
        movsb

;CX=длина строки
;Настроим сегментный ре-
;гистр ES на наши данные
;DS:SI=адрес исходной строки
;ES:DI=адрес пересылки
;Пересылка CX байтов

;Файловая функция вывода
;Дескриптор стандартного вывода
;Столько байтов вывести
;Адрес строки
;Переход в MS-DOS
;Поля данных
str     db     'Пересылаемая строка'
strlen  equ    $-str
mes     db     80 dup ('*')

```

**Задача 3.5. Сравнение строк.** Сравнить две одинаковые строки, вывести на экран результат сравнения. Отладить программу, в которой сравниваемые строки одинаковы (как в приведенном примере). Модифицировать текст программы, сделав строки не одинаковыми, и выполнить программу повторно.

```

;Основные фрагменты программы
;Сравним строки
mov     CX, 32
mov     AX, seg str2
mov     ES, AX
mov     SI, offset str1
mov     DI, offset str2
cmpsb
jne     notequ
;Длина строк
;Сегментный адрес str2
;Настроим на него ES
;DS:SI=адрес str1
;ES:DI=адрес str2
;Сравнение CX байтов
;Если не равны, на notequ
;Выведем на экран сообщение mes1 о равенстве строк
...
jmp     outprog
;На завершение программы

```

```

notequ:
;Выведем на экран сообщения mes2 о неравенстве строк
...
outprog:
;Завершим программу функцией 4Ch прерывания DOS 21h
...

```

```

;Поля данных
str1    db     32 dup ('&') ;Первая строка
str2    db     32 dup ('&') ;Вторая строка
mes1    db     'Строки одинаковы', 10, 13
mes2    db     'Строки различаются!', 10, 13

```

**Задача 3.6. Составить макроопределения каких-либо типовых действий (например, программной задержки и вывода строки на экран). Написать программу, содержащую макровыводы с разными значениями параметров и проверить правильность ее выполнения.**

```

;Основные фрагменты программы
;Макроопределение для программной задержки
;Параметр = задержка в секундах
delay   macro    time
        local    outer, inner
        mov     CX, time*3
;Имя delay, параметр time
;Локальные метки
;CX=время в секундах (коэффициент
;подбирается экспериментально!)
;Сохраним его в стеке
;64K шагов во внутреннем цикле
;Тело цикла - всего 1 строка
;Восстановим CX перед командой
;loop внешнего цикла outer
;Конец макроопределения

outer:  push    CX
        mov     CX, 0
inner:  loop   inner
        pop     CX
        loop   outer
endm

;Макроопределение вывода строки на экран
;Первый параметр = адрес строки, второй параметр = ее длина
outstring macro    mes, len
        mov     AH, 40h
        mov     BX, 1
        mov     CX, len
        mov     DX, offset mes
        int     21h
endm

```

```

;Начало главной процедуры
proc
mov     AX, data
mov     DS, AX
outstring m1, 3
delay   10
outstring m2, 4
delay   5
outstring m3, 1
;Конец макроопределения

;Инициализация сегментного
;регистра DS
;Вывести строку m1, 3 байта
;Задержка на 10 с
;Вывести строку m2, 4 байта
;Задержка на 5 с
;Вывести строку m3, 1 байт

```

```

;Поля данных
m1     db     '***'
m2     db     '***'
m3     db     '!'

```



**Задача 3.7.** Оформить макроопределения из предыдущей задачи в виде отдельного файла с именем, например, **MACRO.LIB**, создать тем самым макробиблиотеку пользователя. Написать программу, содержащую обращения к макроопределениям из макробиблиотеки.

;Основные фрагменты программы  
include macro.lib

```
...
outstring m1,3
delay 10
outstring m2,4
delay 5
outstring m3,1
```

```
;Поля данных
m1 db '<>'
m2 db '***'
m3 db '1'
```

**Задача 3.8.** Подпрограммы. Оформить фрагменты, организующие программную задержку и вывод строки на экран, в виде подпрограмм, включенных в текст основной программы (в виде отдельных процедур). Вызвать процедуры из главной программы, передавая параметры через выделенные для этого регистры.

;Основные фрагменты программы

delay proc near  
;Подпрограмма DELAY организации программной задержки

;На входе CX=значение задержки в секундах

```
push DX
push AX
xchg AX,CX
mov CX,3
```

;Сохраним регистры, неявно  
используемые подпрограммой  
;AX=значение задержки  
;Коэффициент перевода в секунды  
;(подобрать экспериментально)  
;Умножим CX на AX  
;Перешлем результат в CX  
;Восстановим значения AX  
;и DX из вызывающей программы  
;Далее как и раньше

```
mul CX
mov CX,AX
pop AX
pop DX
outer: push CX
mov CX,0
inner: loop inner
pop CX
loop outer
ret
```

;Возврат в вызывающую процедуру  
;Конец процедуры

delay endp

;Подпрограмма outstring вывода строки на экран

;На входе CX=число выводимых байтов, DX=адрес строки

```
outstring proc near
mov AH,40h
mov BX,1
int 21h
ret
outstring endp
```

;Начало главной процедуры

```
start proc
...
mov CX,meslen ;Настроим CX
mov DX,offset mes ;Настроим DX
call outstring ;Вызовем процедуру вывода
mov CX,10 ;Настроим CX
call delay ;Вызовем процедуру задержки
mov CX,meslen-6 ;Еще раз выведем строку
mov DX,offset mes+3 ;(немного другую)
call outstring
```

;Поля данных

```
mes db '*** Сообщение ***',10,13
meslen equ $-mes
```

**Задача 3.9.** Оформить подпрограммы из предыдущего примера в виде отдельных исходных файлов (дав им, естественно, различающиеся имена). Оттранслировать их и получить объектные модули. С помощью программы-библиотекаря **LIB.EXE** Microsoft создать объектную библиотеку пользователя и включить в нее объектные модули процедур-подпрограмм. Написать главную программу, содержащую вызовы процедур из объектной библиотеки. Скомпоновать главную программу с модулями из объектной библиотеки и проверить ее работоспособность.

Командная строка создания объектной библиотеки **MYOBJ.LIB** (вместе с файлом оглавления **MYOBJ.LST**) и включения в нее объектных модулей **DELAY.OBJ** и **OUTSTR.OBJ**:

>LIB MYOBJ.LIB +DELAY +OUTSTR, MYOBJ.LST;

Здесь **DELAY** и **OUTSTR** - произвольные имена объектных файлов с подпрограммами. Эти имена могут для наглядности совпадать с именами вызываемых процедур, но могут и отличаться от них. Обратите внимание на то, что в командных строках вызова библиотекаря или компоновщика указываются имена объектных файлов, а в программе - имена вызываемых процедур.

Командная строка компоновки главной программы **P.OBJ** с подпрограммами из объектной библиотеки:

>LINK /CO P.P.,MYOBJ.LIB

### 3.5. Обращение к системным средствам из прикладной программы

Обращение к функциям **DOS** и **BIOS** осуществляется с помощью программных прерываний (команда **INT**).

Система прерываний машин типа **IBM PC** в принципе не отличается от любой другой системы векторизованных прерываний. Самое начало оперативной памяти от адреса **0000h** до

03FFF отводится под векторы прерываний - четырехбайтовые области, в которых хранятся адреса программ обработки прерываний (ПОП). В два старшие байта каждого вектора записывается сегментный адрес ПОП, в два младшие - относительный адрес точки входа в ПОП в сегменте. Векторы, как и соответствующие им прерывания, имеют номера, называемые типами, причем вектор с номером 0 (вектор типа 0) располагается на адресе 0, вектор типа 1 - с адреса 4, вектор типа 2 - начиная с адреса 8 и т.д. Вектор с номером N занимает, таким образом, байты памяти от  $N \cdot 4$  до  $N \cdot 4 + 3$ . Всего в выделенной под векторы области памяти помещается 256 векторов.

Получив сигнал на выполнение процедуры прерывания с определенным номером, процессор сохраняет в стеке выполняемой программы слово флагов, а также сегментный и относительный адрес сегмента команд (содержимое CS и IP) и загружает CS и IP из соответствующего вектора прерываний, осуществляя тем самым переход на ПОП (рис 3.1).

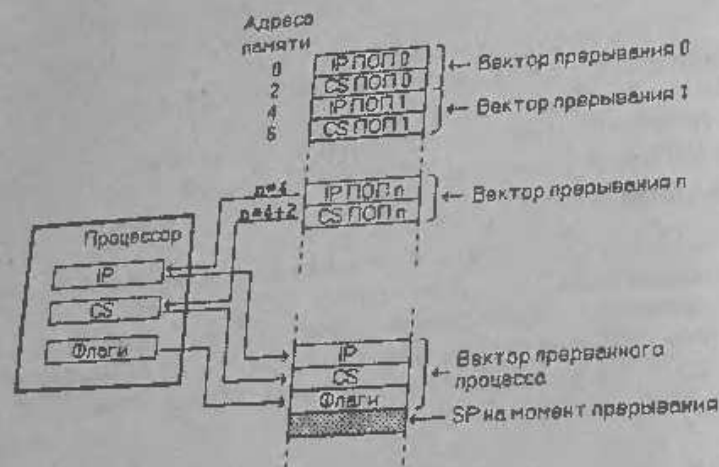


Рис. 3.1. Процедура прерывания.

Программа обработки прерывания обычно заканчивается командой возврата из прерывания IRET, выполняющей обратные действия - загрузку IP, CS и регистра флагов из стека, что приводит к возврату в основную программу в точку, где она была прервана.

Запросы на выполнение процедуры прерываний могут иметь различную природу. Прежде всего различают аппаратные прерывания от периферийных устройств или других компонентов

системы и программные прерывания, вызываемые командой INT, которая используется, в частности, для программного обращения к функциям DOS и BIOS. Сигналы, возбуждающие аппаратные прерывания, могут инициироваться сигналами самого процессора, например, при попытке выполнения операции деления на ноль (такие прерывания называются внутренними, или отказами), а могут приходить из периферийного оборудования (внешние прерывания). Независимо от источника, процедура прерывания, описанная выше, всегда выполняется одинаково, как для аппаратных, так и для программных прерываний.

Большая часть векторов прерываний зарезервирована для выполнения определенных действий; часть из них автоматически заполняется адресами системных программ при загрузке системы. Векторы со следующими номерами могут представлять особый интерес для пользователя:

- 00h - деление на 0;
- 01h - пошаговое выполнение (при TF=1);
- 02h - немаскируемое прерывание (вывод NM1 процессора)
- 03h - команда INT без числового параметра;
- 04h - INTO - прерывание по переполнению (инициируется аппаратно, но лишь при наличии в программе команды INTO);
- 05h - прерывание возбуждается при нажатии клавиши PrintScreen
- 08h - таймер (аппаратное);
- 09h - клавиатура (аппаратное);
- 0Ah - зарезервировано для подключения нестандартного устройства (аппаратное);
- 0Bh - второй последовательный порт COM2 (аппаратное);
- 0Ch - первый последовательный порт COM1 (аппаратное);
- 0Dh - жесткий диск (PC, XT), второй параллельный порт LPT2 (AT) (аппаратное);
- 0Eh - гибкий диск (аппаратное);
- 0Fh - параллельный порт (принтер LPT1) (аппаратное);
- 10h - видеодрайвер BIOS;
- 13h - драйвер BIOS диска;
- 14h - драйвер последовательного порта;
- 16h - драйвер BIOS клавиатуры;
- 17h - драйвер BIOS принтера;
- 19h - начальный загрузчик BIOS;
- 1Ah - календарь-часы BIOS;
- 1Bh - обработчик прерываний по <Ctrl> / <Break>;
- 1Ch - программа-заглушка BIOS для обработки прерываний от системного таймера (18,2 прерываний в секунду);
- 1Dh - адрес таблицы видеопараметров, BIOS;

- 1Eh - адрес таблицы параметров дискеты, BIOS;  
 1Fh - адрес второй половины таблицы шрифтов графических режимов 4...6, BIOS;  
 21h - диспетчер функций DOS;  
 22h - адрес перехода при завершении процесса, DOS;  
 23h - обработчик прерываний по <Ctrl>/C;  
 24h - обработчик прерываний по критической ошибке;  
 25h - абсолютное чтение диска;  
 26h - абсолютная запись на диск;  
 28h - программа-заглушка DOS для активизации резидентных программ командами с клавиатуры;  
 2Fh - мультиплексное прерывание DOS;  
 33h - драйвер мыши фирмы Microsoft;  
 43h - адрес таблицы шрифтов графических режимов, BIOS;  
 60h...66h - прерывания пользователя;  
 67h - драйвер дополнительной памяти LIM EMS;  
 68h...6Fh - свободные векторы;  
 70h - КМОП-часы реального времени (AT, аппаратное);  
 71h - программа BIOS, возбуждающая прерывание INT 0Ah для совместимости машин XT и AT в части обслуживания нестандартных внешних устройств (AT, аппаратное);  
 72h...73h - зарезервировано (AT, аппаратное);  
 74h - мышь (PS/2, аппаратное);  
 76h - жесткий диск (AT, аппаратное);  
 77h - зарезервировано (AT, аппаратное);  
 78h...7Fh - свободные векторы;  
 F1h...FFh - не используются.

Как видно из таблицы, векторы прерываний можно условно разбить на следующие группы:

- векторы аппаратных прерываний (08h...0Fh и 70h...77h);
- драйверы BIOS (10h, 13h, 16h и т.д.);
- программы DOS (21h, 22h, 23h и т.д.);
- адреса системных таблиц BIOS (1Dh, 1Eh, 43h и т.д.).

Системные программы, адреса которых хранятся в векторах прерываний, в большинстве своем являются всего лишь диспетчерами, открывающими доступ к большим группам программ, реализующих системные функции. Так, видеодрайвер BIOS (вектор 10h) включает программы смены видеорежима, управления курсором, задания цветовой палитры, загрузки шрифтов и многие другие. Особенно характерен в этом отношении вектор 21h, через который осуществляется вызов практически всех функций DOS: ввода с клавиатуры и вывода на экран, обслуживания файлов, каталогов и дисков, управления памятью и процессами, службы времени и т.д.

Обращение из прикладной программы к системным функциям осуществляется единообразно. В регистр AH засылается номер функции (не путать с типом прерывания!), в другие регистры - исходные данные, необходимые для выполнения конкретной системной программы. После этого выполняется команда INT с числовым аргументом, указывающим тип (номер) прерывания, например, INT 21h.

Большинство функций DOS и многие функции BIOS возвращают в флаге переноса CF код завершения. Если функция выполнялась успешно, CF=0, в случае же любой ошибки CF=1. В последнем случае в одном из регистров (чаще всего в AX) возвращается еще и код ошибки. Таким образом, типичная процедура обращения к системным средствам выглядит следующим образом:

```
mov     AH,func           ;func - номер функции
;Заполнение тех или иных регистров (AL, BX, ES, BP и др.)
;параметрами, необходимыми для выполнения данной функции
...
int     21h               ;Переход в MS-DOS
jc      err0r             ;Строка выполняется сразу
;после возврата из DOS

;Продолжение программы
...
err0r:  cmp     AX,1       ;Анализ кода завершения
        je      err1
        cmp     AX,2
        je      err2
        ...
```

Аналогично вызываются и функции BIOS. Справочный материал по функциям DOS и BIOS, описанным в настоящей книге, приведен в Приложениях 1 - 3.



## 4. Работа с файлами, каталогами и дисками

### 4.1. Основные характеристики файловой системы MS-DOS

В машинах типа IBM PC предусмотрены два уровня обращения к магнитным дискам. При работе на нижнем уровне пользователь с помощью прерывания BIOS INT 13h обращается непосредственно к программам управления диска. Типичными операциями этого уровня являются запись или чтение сектора, позиционирование головок, форматирование дорожки. Файловая система DOS не используется; требуемая информация отыскивается не по имени файла, а по номерам поверхности, цилиндра и сектора.

Верхний уровень реализуется с помощью прерывания DOS INT 21h, поддерживающего, наряду с прочими, также и функции обслуживания файловой структуры. Программист работает не с программами управления физическим диском, а с файловой системой DOS, получая возможность оперировать такими понятиями файловой системы, как логический диск, каталог, файл.

Как известно, для удобства работы с большим количеством разнородных файлов в DOS используется древовидная структура каталогов. Каталог представляет собой файл обычно относительно небольшого размера, в котором содержится перечень всех подкаталогов следующего уровня и файлов, входящих в данный каталог. Каждому подкаталогу или файлу отводится одна запись размером 32 байт, в которую DOS заносит информацию о файле: имя, начальный адрес на диске (номер кластера), дата и время создания, длина в байтах, а также набор характеристик файла, называемых его атрибутами. Кроме записей, относящихся к нижележащим каталогам и файлам, каждый каталог содержит еще две записи: о себе самом и о родительском каталоге. Формат записи каталога (как корневого, так и любого подкаталога) приведен в табл. 4.1.

Атрибут файла (в том числе файла каталога) хранятся в специально отведенном для них байте атрибутов и могут иметь значения, приведенные в табл. 4.2.

Таблица 4.1. Формат записи каталога.

Смещение	Число байтов	Содержимое
00h	8	Имя файла в кодах ASCII
08h	3	Расширение имени файла в кодах ASCII
0Bh	1	Байт атрибутов файла
0Ch	10	Зарезервировано
16h	2	Время создания или последней модификации файла
18h	2	Дата создания или последней модификации файла
1Ah	2	Номер кластера, с которого начинается файл на диске
1Ch	4	Фактическая длина файла в байтах

Таблица 4.2. Атрибуты файла.

Атрибут	Назначение
01h	Файл только для чтения. Модификация или удаление файла запрещаются
02h	Скрытый файл (не "замечаемый" командой DIR и некоторыми функциями DOS)
04h	Системный файл (обычно системными и заодно скрытыми объявляются файлы IO.SYS и MSDOS.SYS)
08h	Запись о файле представляет собой метку тома. Такая запись (одна на весь том) может существовать только в корневом каталоге
10h	Файл представляет собой каталог
20h	Файл после создания или модификации не был архивирован и, следовательно, нуждается в архивации (атрибут архивации)

Файл может иметь несколько атрибутов одновременно. Так, для записи о метке тома характерно значение бита атрибутов 28: метка тома, не архивирована. Защищенный от стирания и модификации файл содержит в байте атрибутов число 21, а если к тому же он еще объявлен скрытым, то 23.

Время и дата создания файла помещаются в запись каталога после окончания работы программы с данным файлом и его закрытия. Формат этих полей приведен в табл. 4.3 и 4.4.

Таблица 4.3. Формат поля времени создания файла.

Биты	Назначение
04...00	Число двухсекундных интервалов в двоичной форме (от 0 до 29, т.е. от 0 с до 58 с)
10...05	Число минут в двоичной форме (от 0 до 59)
15...11	Число часов в двоичной форме (от 0 до 23)

Таблица 4.4. Формат поля даты создания файла.

Биты	Назначение
04...06	День месяца (от 1 до 31)
08...09	Месяц (от 1 до 12)
15...09	Год относительно 1980

Номер кластера, с которого начинается файл, позволяет найти на диске начало файла; информация о его последующих кластерах содержится в таблице размещения файлов (FAT от File Allocation Table).

В последних четырех байтах записи каталога хранится длина файла в байтах. Если в процессе модификации размер файла увеличился, DOS изменяет значение данного поля.

При создании нового файла DOS сама отыскивает на диске свободное место и назначает его новому файлу, создавая и заполняя соответствующую этому файлу запись в каталоге. Хотя минимальной порцией информации, передаваемой контроллером диска в процессе записи или чтения файла, является сектор (512 байтов) (и программы BIOS работают как раз с секторами), файловая система назначает место на диске целыми кластерами. Размер кластера на гибком диске составляет обычно два сектора (1 Кбайт); на жестком диске в кластер могут входить 4 - 8 секторов. Таким образом, минимальный физический размер файла, даже если данные в нем занимают лишь несколько байтов, составляет один кластер. Однако в записи каталога указывается не физическая, а логическая длина файла, т.е. объем содержащихся в нем данных в байтах.

Методика работы с файлами существенно определяется тем обстоятельством, что каждый файл может занимать на диске несколько несмежных областей, т.е. быть разрывным. Такая система выделения дискового пространства позволяет, во-первых, в процессе работы с файлом многократно дописывать в него новые данные, увеличивая при этом длину файла и, во-вторых, снимает проблемы с фрагментацией диска, поскольку даже самые маленькие и разрозненные свободные области на диске могут быть использованы для размещения нового файла. Следует, однако, иметь в виду, что сильно фрагментированный файл требует заметно больше времени для чтения или записи, что снижает скорость выполнения программ.

Существуют два способа выполнения операций с файлами: с использованием блоков управления файлами (FCB, File control block) и дескрипторов файлов (handle).

Блок управления файлами представляет собой 37-байтную таблицу, содержащую информацию о файле: имя и расширение

файла, его размер, длину записей в файле, номер текущей записи и т.д. Эта таблица размещается в памяти, отводимой программе. Для выполнения какой-либо операции с файлом, необходимо заполнить FCB соответствующей информацией (например, именем открываемого файла) и вызвать требуемую функцию.

Использование FCB было характерно для первых версий DOS (до 2.0). FCB-структуры и соответствующие функции DOS не поддерживают древовидных каталогов, и поэтому они могут работать только с файлами из текущего каталога текущего диска. С их помощью, естественно, нельзя создавать или удалять сами каталоги. В настоящее время функции типа FCB используются редко и в данном пособии рассматриваться не будут.

Другой способ операций с файлами предполагает использование дескрипторов (файловых индексов, файловых описателей), которые в первом приближении можно рассматривать, как номера открытых файлов.

Процедура обращения к файлу в общем случае распадается на следующие операции:

создание файла с заданным именем в указанном каталоге или открытие файла, если он был создан ранее;

запись в файл или чтение из файла всего содержимого либо любой его части;

закрытие файла.

В большинстве случаев работа с файлом начинается с выполнения операции его открытия, для чего предусмотрена особая функция DOS. Открывая файл, DOS назначает ему очередной свободный элемент (блок описания файла) специальной системной таблицы, называемой таблицей открытых файлов (System File Table, SFT) и располагаемой в оперативной памяти среди системных областей данных. Объем этой таблицы, определяющий максимальное число файлов, с которыми можно работать одновременно, задается на этапе конфигурирования DOS директивой FILES файла CONFIG.SYS.

Найдя в системе каталогов диска запись об открываемом файле, DOS записывает в выделенный ему элемент SFT (блок описания файла) основные характеристики файла, такие, как имя, длину, атрибуты, дату и время создания, стартовый кластер, физический адрес на диске записи каталога, содержащей информацию о файле и ряд других. Часть информации переписывается в элемент SFT из записи каталога, часть (например, указатель на блок параметров диска, где хранится информация о физических характеристиках диска) DOS предоставляет сама. Важным элементом блока описания файла является двухсловная ячейка, в которой хранится указатель файла - номер байта относительно начала файла, с которого начнется очередная



операция записи или чтения. Наличие указателя позволяет организовать прямой доступ к файлу, т.е. чтение или запись начинаются от любого места файла. Ссылку на номер выделенного файлу блока описания файла в SFT DOS возвращает в программу в виде дескриптора.

Обращение к открытому файлу (запись, чтение, изменение характеристик файла и т.д.) осуществляется по присвоенному ему дескриптору; несоткрытый файл дескриптора не имеет, система работать с ним не может. По мере выполнения операций с открытым файлом DOS модифицирует информацию в блоке SFT; содержимое SFT всегда отражает текущее состояние файла.

После окончания работы с файлом его надо закрыть специальной функцией DOS. В процессе закрытия осуществляется сброс на диск буферов DOS, модификация записи каталога и освобождение блока описания файла в SFT вместе с закрепленным за ним дескриптором.

Буферы DOS, количество которых определяется директивой BUFFERS файла CONFIG.SYS, служат для ускорения работы с файлом. DOS, получив из выполняемой программы заказ на чтение некоторой порции данных из файла, находит и считывает соответствующие секторы диска (в которых, между прочим, данных может быть больше, чем конкретно затребовала программа) и, переслав прочитанные данные в программу, помимо этого сохраняет содержимое прочитанных секторов в своих внутренних буферах. Если программа в дальнейшем перестанет DOS запрос на чтение с диска или запись на диск тех данных, которые уже находятся в буферах DOS, система выполнит затребованные операции не на диске, а лишь в буферах DOS, что на несколько порядков сократит время их выполнения. Однако в этом случае состояние файла на диске не всегда отвечает его логическому образу в программе. Сброс буферов DOS на диск в процессе закрытия файла выполняет физическое обновление файла на диске и приведение его в соответствие с логическим образом в программе.

Схожая ситуация складывается с характеристиками файла в записи каталога. Пока идет работа с файлом (например, добавление в него новых данных с увеличением его длины) информация о характеристиках файла обновляется только в блоке описания файла в SFT. Каталог на диске модифицируется лишь при закрытии файла, когда измененные характеристики файла переписываются из SFT в запись каталога.

Наконец, при закрытии файла освобождается выделенный ему блок описания файла вместе с дескриптором. И то, и другое можно теперь использовать для работы с другим файлом. Таким образом, система может последовательно работать с

неограниченным количеством файлов, но число одновременно открытых файлов определяется объемом системной таблицы файлов.

При завершении программы (для этого предусмотрена функция DOS 4Ch) выполняется автоматическое закрытие всех открытых в программе файлов. Поэтому в простых и не слишком ответственных программах файлы можно явным образом не закрывать - они все равно будут закрыты системой.

Файловые функции DOS чтения и записи через дескрипторы характерны тем, что их можно использовать и для ввода-вывода через стандартные устройства компьютера. При этом для работы со стандартными устройствами DOS предоставляет пять предопределенных дескрипторов:

- 0 - стандартный ввод (CON);
- 1 - стандартный вывод (CON);
- 2 - стандартная ошибка (CON);
- 3 - стандартный вспомогательный порт (AUX);
- 4 - стандартный принтер (PRN).

Таким образом, при работе с терминалом, принтером или последовательным портом нет необходимости открывать новые дескрипторы; ввод с клавиатуры осуществляется через дескриптор 0, вывод на экран - через дескрипторы 1 или 2, вывод на принтер - через дескриптор 4. Различие дескрипторов 1 и 2 заключается в том, что стандартный вывод (как и стандартный ввод) можно перенаправить средствами DOS на любое устройство или в файл, а стандартная ошибка всегда связана с экраном. Обычно дескриптор 2 используют для вывода на экран аварийных или диагностических сообщений.

Перенаправление ввода или вывода программы осуществляет командный процессор COMMAND.COM. Если в программе PROG предусмотрен ввод данных с помощью какой-либо файловой функции через дескриптор стандартного ввода (0), и вывод данных через дескриптор стандартного ввода (1), то при обычном запуске программы командой

PROG.EXE

программа будет требовать входные данные с клавиатуры и выводить результаты своей работы на экран. Если, однако, в команде запуска программы использовать символ перенаправления

PROG.EXE > FILE.DAT

система сама создаст файл FILE.DAT, и весь вывод программы будет записан в этот файл. Ввод по-прежнему будет осуществляться с клавиатуры. Запуск программы командой

PROG.EXE < FILE.INI



приведет к тому, что программа вместо обращения к клавиатуре попытается ввести всю требуемую ей информацию из файла FILE.INI. Естественно, этот файл должен быть заранее подготовлен пользователем. Вывод программы опять поступит на экран. Наконец, команда

PROG.EXE < FILE.INI > FILE.DAT

заставит программу выполняться в режиме ввода информации из файла FILE.INI и вывода в файл FILE.DAT. Ни экран, ни клавиатура использоваться не будут. Следует подчеркнуть, что сама программа ничего не знает об этих перенаправлениях — она во всех случаях обращается к стандартному устройству ввода для ввода информации и к стандартному устройству вывода для ее вывода. Однако DOS как бы подставляет ей на входе и выходе другие устройства.

Для облегчения ориентации в многочисленных функциях DOS, осуществляющих операции над файлами, каталогами и дисками, их удобно разбить на смысловые группы.

#### 1. Создание, открытие и закрытие файла:

- 3Ch - создать файл;
- 5Ah - создать временный файл;
- 5Bh - создать новый файл;
- 3Dh - открыть файл;
- 3Eh - закрыть файл;
- 68h - сбросить файл на диск;
- 41h - удалить файл.

#### 2. Запись и чтение данных:

- 42h - установить указатель;
- 3Fh - читать из файла или устройства;
- 40h - записать в файл или устройство.

#### 3. Изменение характеристик файла:

- 43h - получить или установить атрибуты файла;
- 56h - переименовать файл;
- 57h - получить или установить дату и время создания файла.

#### 4. Поиск файла:

- 1Ah - установить адрес области передачи данных (DTA);
- 2Fh - получить адрес области передачи данных (DTA);
- 4Eh - найти первый файл;
- 4Fh - найти следующий файл.

#### 5. Операции над каталогами:

- 39h - создать каталог;
- 3Ah - удалить каталог;
- 3Bh - сменить текущий каталог;
- 47h - получить текущий каталог.

#### 6. Операции над дисками:

- 19h - получить текущий диск;
- 0Eh - сменить текущий диск;
- 36h - получить информацию о диске.

Функции 3Ch и 5Bh позволяют создать файл с заданной спецификацией. Спецификация файла, т.е. путь к нему вместе с именем файла и расширением указывается в виде символической строки, завершающейся двоичным нулем ("строки ASCIIZ"). Адрес этой строки заносится в регистры DS:DX. В регистре CX задается код атрибутов создаваемого файла: 0 - отсутствие атрибутов, 1 - только для чтения, 2 - скрытый, 4 - системный, 8 - метка тома, 20h - атрибут архива. Таким образом, с помощью этих функций можно создать как "настоящий" файл, так и метку тома (в корневом каталоге диска). В регистре AX возвращается дескриптор созданного файла, которым можно в дальнейшем пользоваться для записи в файл или чтения из него. Различие функций 3Ch и 5Bh проявляется лишь в случае, когда файл с заданной спецификацией уже существует. Функция 3Ch при этом фактически уничтожает имеющийся файл и создает новый с тем же именем, а функция 5Bh завершается с CF=1.

Функция 5Ah используется для создания временного файла, имя которому (являющееся функцией текущего времени) дает система. В регистрах DS:DX указывается адрес пути к файлу (не имени файла!) в виде строки ASCIIZ, в конце которой должны быть предусмотрены 13 пустых байтов, куда DOS поместит обратный слэш и имя создаваемого файла, завершаемое двоичным нулем. При необходимости файлу можно придать любые атрибуты (см. описание функций 3Ch и 5Bh) кроме атрибута метки тома. Обычно временные файлы удаляются перед завершением программы, причем забота об этом лежит на программисте (автоматически файл не удаляется). Для записи в созданный временный файл следует использовать дескриптор, возвращаемый функцией 5Ah в регистре AX.

Функция 3Dh позволяет открыть уже имеющийся файл. В регистрах DS:DX задается спецификация файла (путь и имя файла с расширением) в виде строки ASCIIZ; в регистре AL - режим доступа (0 - чтение, 1 - запись, 2 - чтение и запись). В дальнейшем запись в файл и чтение из него осуществляется с помощью дескриптора, возвращаемого функцией в регистре AX.

Для каждого открытого файла DOS создает и поддерживает указатель, который представляет собой относительный номер байта в файле, начиная от которого будут выполняться запись или чтение данных. Указатель только что открытого или созданного файла позиционируется системой на начало файла, а

функции чтения или записи смешают его на число прочитанных или записанных байтов. Таким образом, повторное использование функций чтения или записи реализует последовательный доступ к файлу. Для организации прямого доступа к произвольному месту файла предусмотрена функция 42h, позволяющая задать положение указателя относительно начала файла (для этого надо задать AL=0), конца файла (AL=2) или текущего положения указателя (AL=1). Само значение смещения указателя (со знаком) заносится в регистры CX (старшая половина) и DX (младшая).

Функции 3Fh и 40h используются для чтения из файла или устройства (функция 3Fh) и записи в файл или устройство (функция 40h). Перед вызовом функции в регистр BX помещается дескриптор, в регистр CX - число читаемых или записываемых байтов, а в регистры DS:DX - адрес буфера в программе пользователя.

Иногда возникает необходимость найти в некотором каталоге все файлы, удовлетворяющие условиям шаблона групповой операции (например, все файлы с расширением .TXT или все файлы с именем EXAMPLE и любыми расширениями). Поиск файлов по заданному шаблону групповых операций осуществляется с помощью функций 4Eh (найти первый файл) и 4Fh (найти следующий файл). Для их использования необходимо с помощью функции 1Ah организовать в программе область передачи данных (Disk transfer area, DTA) размером не менее 43 байтов, либо с помощью функции 2Fh получить адрес области передачи данных, созданной DOS. Известно, впрочем, что в качестве DTA DOS использует область PSP от байта 80h до конца. DOS помещает в DTA информацию о найденном файле (атрибуты, время и дата создания, размер и т.д.). В частности, в байтах 1Eh...2Ah DTA содержится имя и расширение файла в виде строки ASCII.

При поиске файлов по заданному шаблону сначала активируется функция 4Eh. В регистры DS:DX помещается адрес строки ASCII с путем к рассматриваемому каталогу, а в регистр CX - код комбинации атрибутов искомого файла. 0 обозначает "нормальный" файл, т.е. файл без атрибутов, 1 - только для чтения, 2 - скрытый, 4 - системный, 8 - метка тома, 10h - каталог, 20h - атрибут архивации. Если установлены атрибуты поиска, то ищутся как нормальные файлы, так и файлы с заданными атрибутами. В случае успешного нахождения заданного файла функция возвращает CF=0, а имя и расширение файла в виде строки ASCII помещаются в DTA, в байты 1Eh...2Ah. Получив имя файла, можно открыть его с помощью функции 3Dh и выполнить далее требуемые операции (чтение, запись и т.д.).

Поиск следующих файлов, удовлетворяющих условиям заданного шаблона, осуществляется с помощью функции 4Fh, которая используется так же, как и функция 4Eh поиска первого файла. При необходимости функцию 4Fh можно активизировать многократно, пока CF=1 не покажет, что все файлы, удовлетворяющие условиям заданного шаблона, исчерпаны.

Использование остальных перечисленных выше функций для работы с файлами не представляет особых трудностей. Действие многих из них проиллюстрировано в приведенных ниже примерах.

#### 4.2. Задачи по программированию операций над файлами, каталогами и дисками.

**Задача 4.1.** Создание файла. В текущем каталоге диска создать файл с именем MYFILE.001 и записать в него символьную строку. После выполнения программы проверить наличие файла на диске и распечатать его содержимое. Это можно выполнить с помощью оболочки DOS Norton Commander или с помощью команд DOS DIR и TYPE.

```
;Определения
cr=13
lf=10
text segment 'code'
assume CS:text,DS:data
myproc proc
    mov     AX,data
    mov     DS,AX

;Создадим файл
    mov     AH,3Ch
    mov     CX,0
    mov     DX,offset filename
    int     21h
    mov     handle,AX

;Запишем строку в файл
    mov     AH,40h
    mov     BX,handle
    mov     CX,stringln
    mov     DX,offset string
    int     21h

;Закроем файл (нет необходимости, если файл не надо читать повторно)
    mov     AH,3Eh
    mov     BX,handle
    int     21h

;Завершим программу
outproc: mov     AX,4C00h
        int     21h
myproc endp
text ends
```

;Возврат каретки  
;Перевод строки  
;Функция создания файла  
;Без атрибутов  
;Адрес имени файла  
;Сохраним дескриптор файла  
;Функция записи  
;Дескриптор  
;Длина строки  
;Адрес строки  
;Функция закрытия  
;Дескриптор  
;Функция завершения, код  
;завершения = 0

```

data segment
string db 'Текстовая строка', 0, 10 ;Строка для записи в файл
string equ $-string ;Ее длина
handle dw ? ;Ячейка для дескриптора
filename db 'MYFILE.001', 0 ;Имя файла в формате ASCIIIZ
data ends
stack segment para stack 'STACK'
db 128 dup (?)
stack ends
end mprog

```

**Задача 4.2.** Чтение файла. Прочитать содержимое файла MYFILE.001 в память и вывести его на экран. Предполагается, что размер файла не более 80 байтов.

```

;Определения
stdout=1
;Основные фрагменты программы
;Откроем файл
mov AH, 3Dh ;Функция открытия файла
mov AL, 2 ;Доступ для чтения/записи
mov DX, offset filename ;Адрес имени файла
int 21h ;Получили дескриптор
mov handle, AX
;Попытаемся прочитать 80 байтов
mov AH, 3Fh ;Функция чтения
mov BX, handle ;Дескриптор
mov CX, 80 ;Столько читать
mov DX, offset bufin ;Сюда
int 21h ;Столько реально прочитали
mov CX, AX
;Выведем прочитанное на экран
mov AH, 40h ;Функция записи
mov BX, stdout ;Дескриптор стандартного вывода
mov DX, offset bufin ;Отсюда выводить (CX байтов)
int 21h
;Завершим программу
...
;Поля данных
bufin db 80 dup (' ') ;Буфер ввода
handle dw ? ;Ячейка для дескриптора
filename db 'MYFILE.001', 0 ;Имя файла в формате ASCIIIZ

```

**Задача 4.3.** Прямой доступ к файлу. Прочитать 8 байтов из созданного ранее файла MYFILE.001, начиная с байта 5, вывести их на экран.

```

;Определения
stdout=1 ;Дескриптор стандартного вывода
;Основные фрагменты программы
;Откроем файл
...
;Установим указатель
mov AH, 42h ;Функция установки указателя

```

```

mov AL, 0
mov BX, handle ;От начала файла
mov CX, 0 ;Дескриптор
mov DX, 5 ;Старшая половина указателя
int 21h ;Младшая половина указателя

```

;Прочитаем 8 байтов данных с помощью функции 3Fh

;Выведем прочитанное на экран с помощью функции 40h

;Завершим программу

```

;Поля данных
bufin db 80 dup (' ') ;Буфер ввода
handle dw ? ;Ячейка для дескриптора
filename db 'MYFILE.001', 0 ;Имя файла

```

**Задача 4.4.** Добавление данных к файлу. Добавить символьную строку к концу символического файла. Проконтролировать содержимое и длину файла до и после добавления (средствами Norton Commander). Повторить выполнение программы и еще раз проконтролировать результат.

;Основные фрагменты программы  
;Откроем файл с указанным именем функцией 3Dh и сохраним полученный дескриптор в ячейке handle

;Установим указатель на конец файла

```

mov AH, 42h ;Функция установки указателя
mov AL, 02 ;От конца файла
mov BX, handle ;Дескриптор
mov CX, 0 ;Старшая половина указателя
mov DX, 0 ;Младшая половина указателя
int 21h

```

;Допишем новую строку

```

mov AH, 40h ;Функция записи
mov BX, handle ;Дескриптор
mov CX, stringln ;Длина строки
mov DX, offset string ;Ее адрес
int 21h

```

;Завершим программу

```

;Поля данных
string db 'Новая строка' ;Строка для вывода в файл
stringln equ $-string ;Длина строки
handle dw ? ;Ячейка для дескриптора
fname db 'MYFILE.001', 0 ;Имя файла

```

**Задача 4.5.** Изменение атрибутов файла. Установить у файла MYFILE.001 атрибут "только для чтения". После завершения программы проконтролировать результат с помощью средств DOS

;Основные фрагменты программы  
;Установим атрибут "только для чтения"



```

mov     AH, 43h      ;Функция работы с атрибутами
mov     AL, 1         ;Установка атрибутов
mov     CX, 1         ;"Только для чтения"
mov     DX, offset fname ;Адрес имени файла
mov     int 21h

```

Задача 4.6. Изменение характеристик файла. Изменить дату и время создания файла MYFILE.001. Эта задача не будет выполняться, если ее запустить после задачи 4.5. Предварительно следует снять с файла атрибут "только для чтения" с помощью команды DOS ATTRIB или программы Notion Commander (или PCTools).

```

;Основные фрагменты программы
;Откроем файл и сохраним его дескриптор

```

```

;Изменим дату и время создания файла
mov     AH, 57h      ;Функция даты/времени
mov     AL, 1         ;Установить дату/время
mov     BX, handle    ;Дескриптор файла
mov     CX, 0         ;Очистим CX
mov     CX, sec        ;Добавим секунды
or      CX, min        ;Добавим минуты
or      CX, hour       ;Добавим часы
or      DX, DX         ;Очистим DX
xor     DX, DX         ;Добавим день
or      DX, day        ;Добавим месяц
or      DX, mon        ;Добавим год
or      DX, year
int     21h

```

```

;Завершим программу

```

```

;Поля данных
fname   db 'MYFILE.001'.0 ;Имя файла
handle  dw 7              ;Ячейка для дескриптора
sec     dw 6/2            ;6 секунд
min     dw 15*32          ;15 минут
hour    dw 16*2048        ;16 часов
day     dw 25             ;25 число
mon     dw 3*32           ;Март
year    dw 13*512         ;13 лет от 1980, т.е. 1993 г.

```

Задача 4.7. Переименование файла. Переименовать файл MYFILE.001, находящийся в текущем каталоге, дав ему имя NEWNAME.DAT.

```

;Основные фрагменты программы
;Настроим сегментный регистр ES на наш сегмент данных

```

```

push    DS
pop      ES

;Переименуем файл
mov     AH, 56h      ;Функция переименования
mov     DX, offset oldname ;Адрес старого имени
mov     DI, offset newname ;Адрес нового имени
int     21h

```

```

;Завершим программу

```

```

;Поля данных
oldname db 'MYFILE.001'.0 ;Старая спецификация
newname db 'NEWNAME.DAT'.0 ;Новая спецификация

```

Задача 4.8. Пересылка файла в другой каталог на том же диске. Переслать файл NEWNAME.DAT из текущего каталога в нижележащий каталог NEWDIR, изменив при этом имя файла на NEWNAME.LEX. Отладив программу, проверить ее возможности, помещая файл в разные каталоги и пересылая его в другие.

```

;Основные фрагменты программы
;Настроим сегментный регистр ES на наш сегмент данных

```

```

;Переименуем файл
mov     AH, 56h      ;Функция переименования
mov     DX, offset oldname ;Адрес старого имени
mov     DI, offset newname ;Адрес нового имени
int     21h
;Завершим программу

```

```

;Поля данных
oldname db 'NEWNAME.DAT'.0 ;Старая спецификация
newname db 'NEWDIR\NEWNAME.LEX'.0 ;Новая спецификация

```

Задача 4.9. Создание и удаление каталогов. Создать в текущем каталоге подкаталог NEWDIR. Удалить из текущего каталога подкаталог OLDDIR (каталог, естественно, должен быть пустым).

```

;Основные фрагменты программы
;Создадим новый каталог

```

```

mov     AH, 39h      ;Функция создания каталога
mov     DX, offset newname ;Адрес нового имени
int     21h

```

```

;Удалим ненужный каталог

```

```

mov     AH, 3Ah      ;Функция удаления каталога
mov     DX, offset oldname ;Адрес имени каталога
int     21h

```

```

;Завершим программу

```

```

;Поля данных
oldname db 'OLDDIR'.0 ;Старая спецификация
newname db 'NEWDIR'.0 ;Новая спецификация

```

Задача 4.10. Смена диска. Сделать текущим диск A:

```

;Основные фрагменты программы
;Сделаем текущим диск A:

```

```

mov     AH, 0Eh      ;Функция смены диска
mov     DL, 0         ;Код диска A (A=0, B=1 и т.д.)
int     21h

```

```

;Завершим программу

```

**Задача 4.11.** Создание метки тома. Создать в корневом каталоге дискеты запись с меткой тома PROGRAMDISK. После выполнения программы проверить наличие метки командами DOS DIR или VOL.

```

;Основные фрагменты программы
;Создадим "файл" - метку тома
mov     ax,3Ch
mov     cx,8
lea     dx,labela
int     21h
;Функция создания файла
;Атрибут метки тома
;Адрес имени метки

```

```

;Завершим программу
;Поля данных
labela  db

```

**Задача 4.12.** Проверка метки тома. Найти в корневом каталоге дискеты запись с меткой тома и убедиться, что на дисководе установлена требуемая дискета.

```

;Основные фрагменты программы
;Установим нашу область передачи диска (DTA),
;чтобы не искать ее в префиксе программного сегмента
mov     ax,1Ah
mov     dx,offset dta
int     21h
;Функция установки DTA
;Адрес нашей DTA

```

```

;Найдем метку
mov     ax,4Eh
mov     cx,8
mov     dx,offset dname
int     21h
;Функция поиска первого файла
;Атрибут "метка тома"
;Адрес спецификации файлов

```

```

;Сравним метки
mov     si,offset lbl
mov     ax,seg dta
mov     es,ax
mov     di,offset dta+1Eh
mov     cx,lbllen
;Адрес имени требуемой метки
;Настроим сегментный
;регистр ES на наш сегмент
;Место для имени файла в DTA
;Длина имени метки
;Будем сравнивать вперед

```

```

;Сравнение
;Метки не совпадают
;Выведем сообщение mes1 о том, что на дисководе стоит
;требуемая дискета

```

```

... outprog
jmp

```

```

;Выведем сообщение mes2 об отсутствии метки на дисководе
;требуемая дискета

```

```

error:
;Выведем сообщение mes3 о том, что на дисководе не та дискета

```

```

;Завершим программу

```

```

;Поля данных
dname  db
A:\
dta     db
lbl     db
lbllen=$-lbl
mes1    db
mes1len=$-mes1
mes2    db
mes2len=$-mes2
mes3    db
mes3len=$-mes3
'A:\*.*',0 ;Просмотрим все имена в корневом каталоге
43 dup(0) ;Область передачи данных
'PROGRAMDISK'
'На дисководе стоит правильная дискета',cg,lf
'На дискете НЕТ метки!',cg,lf
'На дисководе НЕ ТА дискета!',cg,lf

```

### 4.3. Системные средства обслуживания дисков и файлов

При работе с диском на уровне DOS мы обращаемся к файлам по их именам. От пользователя остается скрытым местоположение файлов на диске, и он не может обратиться к системным областям диска - загрузочным записям, таблицам размещения файлов и каталогам. Для работы с этими областями, а также с конкретными секторами или кластерами файлов используются либо функции драйвера BIOS (INT 13h), либо два специальных прерывания DOS - INT 25 и INT 26, осуществляющие доступ не к файлам, а к секторам диска. Для использования этих системных средств необходимо отчетливое представление о физической организации диска и нумерации его секторов.

Диск в общем случае состоит из нескольких двусторонних (или частично односторонних) пластин, на которые информация записывается концентрическими дорожками, как это показано на рис. 4.1.

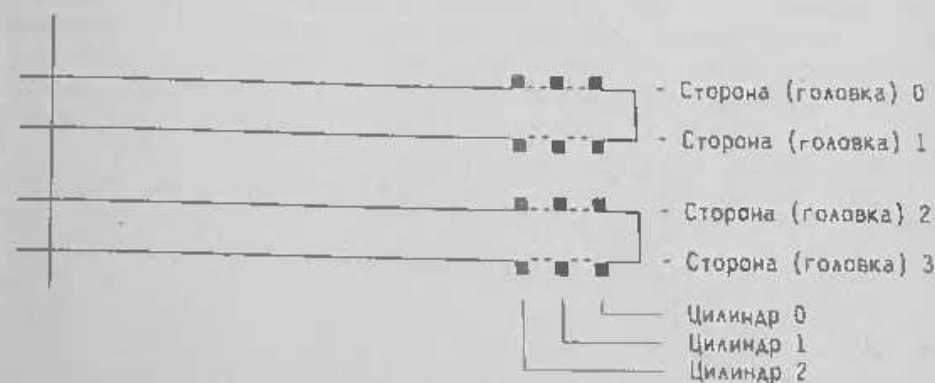


Рис. 4.1. Физическая структура диска.

Дорожки всех пластин (сторон), расположенные на одном удалении от центра, объединяются понятием цилиндра.

Цилиндры (как и дорожки) нумеруются от края диска внутрь, нумерация начинается от 0. Стороны также нумеруются от 0, и поскольку каждую рабочую сторону диска обслуживает отдельная магнитная головка, предназначенная для чтения и записи данных, часто говорят не о номерах сторон, а о номерах головок.

Число сторон и цилиндров жестких дисков изменяется в широких пределах, определяя суммарную емкость и тип диска. Тип диска представляет собой число, хранящееся (для машин типа АТ) в КМОП - ПЗУ. Например, диск типа 1 имеет 306 цилиндров, 4 головки и емкость 10 Мбайт; диск типа 2 - 615 цилиндров, 4 головки и емкость 20 Мбайт и т.д. Используются и нестандартные диски неопределенного типа.

Каждая дорожка делится на секторы. Логический размер сектора для всех дисков составляет 512 байтов. Число секторов на одной дорожке зависит от типа диска: жесткие диски обычно имеют на дорожке 17 или 26 секторов. Существуют две системы нумерации секторов на диске - абсолютная (физическая) и относительная (логическая).

Абсолютная нумерация относится к физическому диску в целом, независимо от того, разбит ли он (в случае жесткого диска) на логические диски или нет. Абсолютное местоположение сектора требует указания номеров стороны, цилиндра и сектора (на дорожке). Относительная нумерация секторов жестких дисков ведется в пределах логических дисков (C:, D:, E: и т.д.). Для дискет понятия логического и физического диска совпадают.

Абсолютная нумерация секторов начинается с 1 (не с 0!) на каждой дорожке. При записи информации на диск DOS сначала целиком заполняет все дорожки цилиндра 0, затем цилиндра 1 и т.д. Поэтому если выстроить все секторы в их логической последовательности, они будут идти в таком порядке (для жесткого диска, имеющего 4 стороны и 17 секторов на дорожке):

секторы 1...17 сторона 0 цилиндр 0  
секторы 1...17 сторона 1 цилиндр 0  
секторы 1...17 сторона 2 цилиндр 0  
секторы 1...17 сторона 3 цилиндр 0  
секторы 1...17 сторона 0 цилиндр 1  
секторы 1...17 сторона 1 цилиндр 1 и т.д.

Самый первый сектор жесткого диска (сектор 1 стороны 0 цилиндра 0) содержит главную загрузочную запись (Master boot), занимающую ровно один сектор и включающую в себя часть программы начальной загрузки и таблицу разделов диска, как это показано на рис. 4.2.



Рис. 4.2. Структуры служебных полей жесткого диска.

В таблице разделов указываются адреса и размеры разделов, на которые разбит жесткий диск. Всего в таблице разделов зарезервировано место для четырех записей о разделах, которых, таким образом, не может быть больше четырех. С другой стороны, DOS предоставляет возможность создания не более двух разделов. Один из них, называемый первичным, служит для размещения системных файлов DOS и выступает, таким образом, в качестве загрузаемого диска C:. Второй раздел называется расширенным; в нем можно создать один или несколько логических дисков (D:, E: и т. д. до Z:). Администраторы диска (например, ADM.SYS) несколько изменяют систему разметки диска и используют все четыре элемента таблицы разделов.

В простейшем случае на диске создается (с помощью программы DOS FDISK) лишь один первичный раздел. DOS не ставит ограничений на размер первичного раздела; он может занимать весь диск, независимо от его емкости. Такая разметка диска не очень удобна, так как обилие разнородной информации, хранящейся на единственном диске, затрудняет пользование ею. Поэтому обычно первичный раздел делают небольшим, иногда не более 300 - 500 Кбайт, и размещают в нем только основные системные файлы. Оставшееся пространство диска отдают расширенному разделу, организуя в нем 2, 3 или более логических дисков, количество которых выбирают исходя из полного объема физического диска и условий использования компьютера.

Так или иначе жесткий диск оказывается разбит на логические диски, каждый из которых занимает целое число



цилиндров; диск C: обычно начинается с начала первой свободной дорожки после главной загрузочной записи, т.е. с сектора 1 стороны 1 цилиндра 0. Каждому логическому диску, входящему в расширенный раздел, предшествует сектор, содержащий таблицу логических дисков (см. рисунок выше). В этой таблице указываются адреса и размеры данного и следующего логических дисков. Таким образом DOS, просматривая последовательно таблицы логических дисков, может постепенно "обратиться" до адреса любого логического диска расширенного раздела.

Таблица логических дисков располагается в самом первом секторе области, выделенной под логический диск, то-есть в секторе 1 стороны 0 цилиндра n. Оставшаяся часть этой дорожки не используется, таким образом собственно логический диск начинается в секторе 1 стороны 1 того же цилиндра. Каждый диск начинается со своей загрузочной записи (Boot), включающей таблицу характеристик диска и программу начальной загрузки (используемую только на загружаемом диске C:). Вслед за загрузочной записью располагаются две таблицы размещения файлов (FAT) и корневой каталог (Root). Все остальное пространство логического диска отводится под область каталогов и файлов (см. рис. 4.3).

Каждый логический диск имеет свою относительную нумерацию секторов, начинающуюся от 0. Таким образом, относительный номер сектора надо использовать с указанием логического диска, при этом относительный сектор 0 соответствует загрузочному сектору, относительный сектор 1 - первому сектору первой копии FAT и т.д. Сектор с таблицей логических дисков (как и сектор главного загрузчика) не входит в систему относительной нумерации секторов и в этом смысле как бы не принадлежит логическому диску. Между прочим, именно по этой причине инструментальные пакеты, работающие не со всем физическим диском, а с логическими дисками (например PC Tools) не позволяют прочитать или модифицировать главный загрузчик и таблицы логических дисков.

Если считать, что размер FAT составляет 7 секторов, и корневой каталог рассчитан на 512 записей, т.е. имеет длину 32 сектора, то системная область займет 47 секторов и область данных начнется с сектора 47. Этот сектор будет соответствовать началу первого кластера данных, имеющего всегда номер 2. На следующем логическом диске вся описанная картина повторяется. На рисунке 4.4 показано возможное соответствие абсолютных и относительных номеров секторов и логических дисков (в предположении, что диск имеет 4 рабочих стороны по 17 секторов на дорожке, а логический диск C: занимает 60 цилиндров).

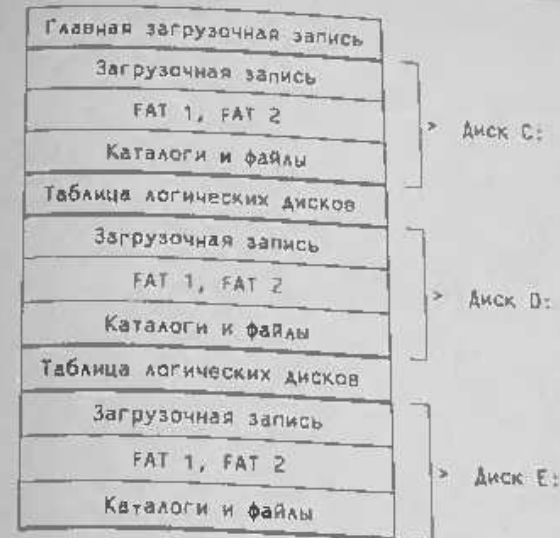


Рис. 4.3. Структура физического диска, разбитого на тома.

Цилиндр	Сторона	Абсолют. секторы	Относ. секторы	Содержимое	Логический диск
0	0	1	-	Master boot	> C:
0	1	1...17	0...16	Boot, FAT1, FAT2, Root	
0	2	1...17	17...33	Root (продолжение)	
0	3	1...17	34...50	Root (окончание), данные	
1	0	1...17	51...67	Данные	
1	1	1...17	68...84	Данные	> D:
60	0	1	-	Таблица логических дисков	
60	1	1...17	0...16	Boot, FAT1, FAT2, Root	
60	2	1...17	17...33	Root (продолжение)	
60	3	1...17	34...50	Root (окончание), данные	
61	0	1...17	51...67	Данные	> E:
61	1	1...17	68...84	Данные	

Рис. 4.4. Нумерация секторов физического диска.

Для вычисления номера сектора, относящегося к какому-то файлу или области диска, необходимо иметь такие характеристики диска, как размеры сектора и кластера, число секторов, составляющих FAT и корневой каталог, количество копий FAT и другие. Все эти характеристики содержатся в загрузочном

Глава 4  
секторе диска (Boot). Наиболее важные из них приведены в табл. 4.5.

Таблица 4.5. Содержимое некоторых полей сектора загрузки

Смещение	Число байтов	Описание
00h	3	Команда перехода на программу начальной загрузки
03h	8	Идентификатор изготовителя
08h	2	Размер сектора в байтах
09h	1	Размер кластера в секторах
0Ah	2	Число зарезервированных секторов
0Eh	1	Число FAT
10h	2	Число записей в корневом каталоге
11h	2	Число секторов на томе
13h	1	Байт описания носителя
15h	2	Размер FAT в секторах
16h	2	Число секторов на дорожке
18h	2	Число сторон (головок)
1Ah	4	Число скрытых секторов
1Ch	4	Серийный номер тома
27h	11	Метка тома
28h		

Операционная система MS-DOS использует в своей работе целый ряд системных таблиц, лишь частично нашедших отражение в официальной документации. Пользоваться этими средствами следует очень осторожно, так как их форматы могут различаться в разных версиях DOS, однако обращение к ним оказывается во многих случаях необходимым. Ниже будут рассмотрены наиболее важные системные таблицы, имеющие отношение к обслуживанию дисков и файлов.

Основополагающей системной таблицей, содержащей адреса целого ряда системных структур, является так называемый список списков (List of lists). Адрес списка списков можно получить, используя недокументированную функцию 52h прерывания 21h. Функция возвращает двухсловный адрес списка списков в регистрах ES:BX. В таблице 4.6 приведены выборочные данные из списка списков для DOS, начиная с версии 4.

В байтах 4...7 списка списков хранится адрес упоминавшейся выше системной таблицы файлов (SFT), куда DOS записывает характеристики открываемых по заказам выполняемой программы файлов или стандартных устройств, к которым предполагается доступ средствами файловой системы. По умолчанию при загрузке DOS формируется одна SFT, в которую входит 5 блоков описания файлов.

Таблица 4.6. Содержимое некоторых полей списка списков

Смещение	Число байтов	Описание
-8	4	Указатель на текущий дисковый буфер
-2	2	Сегментный адрес первого блока управления памятью
00h	4	Указатель на первый блок параметров диска
04h	4	Указатель на первую системную таблицу файлов
08h	4	Указатель на заголовок активного устройства CLOCK\$
0Ch	4	Указатель на заголовок активного устройства CON
10h	2	Максимальный размер сектора в байтах в любом блочном устройстве
12h	4	Указатель на информационную запись дискового буфера
16h	4	Указатель на массив структур текущих каталогов
20h	1	Число установленных блочных устройств
21h	1	Число элементов в массиве структур текущих каталогов
22h	18	Заголовок драйвера устройства NUL
3Fh	2	Значение x в директиве BUFFERS=x,y
41h	2	Значение y в директиве BUFFERS=x,y
43h	1	Дисковод загрузки (I=A:)
45h	2	Размер расширенной памяти в килобайтах

В такой конфигурации компьютер практически не может эксплуатироваться, так как три блока система занимает под характеристики стандартных устройств CON, AUX и PRN, и для открываемых программами файлов остается всего два блока. При включении в файл CONFIG.SIS директивы BUFFERS DOS создаст вторую таблицу, связывая ее с первой (рис. 4.5). Суммарное число блоков в двух таблицах равно параметру директивы BUFFERS.

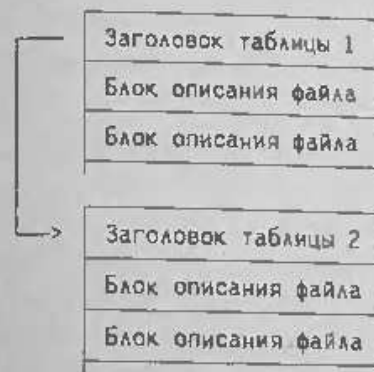


Рис. 4.5. Системные таблицы файлов.

Каждая SFT начинается с 6-байтового заголовка (см. табл. 4.7).

Таблица 4.7. Заголовок SFT

Смещение	Число байтов	Описание
0	4	Адрес следующей таблицы или FFFFh в первом слове, если эта таблица последняя
4	2	Количество блоков описания файлов в данной таблице

Далее идут блоки описания файлов в количестве, указанном в заголовке таблицы. Формат блоков и даже их размер зависят от версии DOS. В таблице 4.8 приведены наиболее интересные для прикладного программиста поля блока описания файла для DOS начиная с версии 4.

Таблица 4.8. Блок описания файла

Смещение	Число байтов	Описание
00h	2	Количество дескрипторов, закрепленных за данным файлом или 0, если данный блок свободен
02h	2	Режим доступа к файлу, заданный при его открытии
04h	1	Атрибуты файла
06h	2	Информационное слово устройства
08h	2	Указатель на заголовок драйвера символического устройства или (для файлов) указатель на блок параметров дискового
0Ah	2	Номер первого кластера файла
0Ch	2	Время последней модификации файла
0Eh	2	Дата последней модификации файла
10h	4	Размер файла в байтах
14h	4	Текущее положение указателя файла
18h	2	Относительный номер последнего прочитанного или записанного кластера файла
1Ah	4	Номер сектора с записью каталога о данном файле
1Ch	1	Номер записи каталога внутри сектора
1Eh	11	Имя и расширение файла (без пути)
20h	2	Сегментный адрес PSP программы, открывшей файл
22h	2	Абсолютный номер последнего записанного или прочитанного кластера файла
35h	2	

Блок параметров диска (Drive Parameter Block, DPB), адрес которого (для того диска, на котором расположен открытый файл) помещается в блок описания файла, содержит информацию, частично повторяющую данные в секторе загрузки. Формат DPB приведен в табл. 4.9.

Таблица 4.9. Блок параметров диска

Смещение	Число байтов	Описание
00h	1	Номер диска (00h=A:, 01h=B: и т.д.)
01h	1	Номер устройства внутри драйвера устройства
02h	2	Размер сектора в байтах
04h	1	Наибольший номер сектора в кластере (размер кластера-1)
05h	1	Число сдвигов, чтобы преобразовать кластеры в секторы
06h	2	Число зарезервированных секторов в начале диска
08h	1	Число таблиц размещения файлов
09h	2	Число элементов в корневом каталоге
0Ah	2	Номер сектора, с которого начинаются данные
0Ch	2	Наибольший номер кластера (число кластеров данных+1)
0Eh	2	Число секторов в одной таблице размещения файлов
10h	2	Номер первого сектора каталога
12h	4	Адрес заголовка драйвера устройства
14h	1	Байт описания носителя
16h	1	00, если к диску было обращение; FFh в противном случае
19h	4	Указатель на следующий DPB
1Dh	2	Номер кластера, с которого следует начинать поиск свободного пространства на диске
1Fh	2	Число свободных кластеров на диске, FFFFh, если неизвестно

Между прочим, для получения адреса блока параметров диска можно воспользоваться функцией DOS 32h (документированной начиная с версии DOS 5.0).

При обращении к SFT DOS пользуется относительными номерами блоков описания файлов. Нумерация блоков начинается с 0 и идет "насквозь" через обе таблицы.

Поскольку порядок описания открытых файлов в SFT заранее неизвестен, для нахождения интересующего нас блока описания файла следует открыть файл функцией 3Dh, получить от системы его дескриптор, а затем по дескриптору найти соответствующий ему блок описания файла в SFT. Для выполнения этой операции надо знать, как значение дескриптора связано с местоположением блока описания файла в SFT.

В префиксе программного сегмента (PSP) имеется таблица, называемая таблицей файлов задания (Job File Table, JFT). По умолчанию эта таблица начинается с байта 18h PSP и имеет размер 20 байтов. В байты JFT DOS по мере открытия файлов записывает номера соответствующих блоков описания файлов в SFT. Свободные байты JFT содержат коды FFh. Относительные номера байтов JFT (от ее начала) и являются дескрипторами открытых файлов.





загрузочную запись, потому что она не входит в состав логического диска. Прерывание INT 26h позволяет записать один или группу секторов с заданным начальным относительным номером. Прерывание INT 13h служит для работы с физическими дисками.

Функция 00h сбрасывает дисковую систему, позиционируя головку на цилиндр 0. Она может понадобиться для привода ния контроллера диска и дисководов. Функция 02h выполняет чтение одного или группы секторов фиксации ошибки чтения или записи.

Функция 03h выполняет чтение одного или группы секторов физического диска в память. Для начального сектора группы указывается его абсолютный адрес, т.е. номера цилиндра, дорожки и собственно сектора в пределах дорожки. Функция позволяет прочитать все сектора диска, в том числе главную загрузочную запись, однако для чтения секторов системной области логического диска требуется определить их абсолютные номера. То же относится и к чтению секторов файлов. При чтении нестандартной дискеты перед вызовом функции 02h необходимо подготовить таблицу параметров дискеты, записав в нее код размера сектора.

Функция 03h выполняет запись секторов на физический диск.

Функция 05h форматирует указанную дорожку диска. Перед ее вызовом для нестандартного форматирования дискеты необходимо подготовить таблицу параметров дискеты, записав в нее код размера сектора.

На машинах PC/AT, оснащенных универсальным дисководом, перед форматированием дискеты необходимо вызвать функцию 17h прерывания 13h, чтобы указать тип используемого дисковода и тип установленной на нем дискеты. Например, на машине может быть установлен дисковод на 1.2 Мбайт, но пользователь в настоящий момент работает с дискетой 360 Кбайт. На машинах PC/XT, оснащенных одним дисководом на 360 Кбайт, надобность в этой функции отпадает.

#### 4.4. Защита программных продуктов от копирования и несанкционированного использования.

Методы защиты данных на персональных компьютерах чрезвычайно разнообразны как по конечной цели, так и по техническому воплощению; их можно разделить на механические, аппаратные и программные.

К механическим способам защиты относятся разнообразные крышки и чехлы с замками (запирающие, например, дисковод гибких дисков или сетевой выключатель), клейкие пластины

для приклеивания терминала к компьютеру, а компьютера к столу, запираемые помещения с сигнализацией и многие другие.

Аппаратные средства реализуются в виде специальных электронных модулей, подключаемых к системному каналу компьютера или портам ввода-вывода, и осуществляющих обмен кодовыми последовательностями с защищаемыми программами.

Наиболее разнообразны программные средства. Сюда относятся программы шифрации данных по заданному пользователем ключу, администраторы дисков, позволяющие ограничить доступ пользователей к отдельным логическим дискам, методы установки программного продукта с дистрибутивных дискет, позволяющие выполнить установку не более заданного числа раз, запуск защищаемых программ посредством не копируемых ключевых дискет, специальные защитные программные оболочки, куда погружаются защищаемые программы и многие другие.

В настоящем разделе, не претендующем на полноту освещения вопроса, будут рассмотрены некоторые программные средства защиты, требующие более глубокого понимания вопросов организации дисков и файлов, а также знакомства с системными областями DOS.

Привязка к местоположению на диске. Если требуется исключить копирование программы с жесткого диска на другой жесткий диск, ее можно привязать к номеру кластера или сектора, с которого начинается файл программы на диске. Привязка осуществляется следующим образом. Специально подготовленная установочная программа открывает файл с рабочей программой и по таблице открытых файлов находит начальный номер кластера. Это число, являющееся своеобразным ключом, записывается установочной программой в определенное место файла рабочей программы (естественно, в поле данных). Рабочая же программа после запуска прежде всего выполняет ту же операцию - определяет свой начальный адрес, а затем сравнивает его с ключом. Если числа совпадают, программа приступает к выполнению своей содержательной части; если не совпадают - аварийно завершается. При копировании программы на другой диск (или даже на тот же самый) она окажется расположенной в другом месте и номер кластера, записанный установочной программой уже не будет соответствовать реальному адресу файла. В то же время с помощью установочной дискеты программу нетрудно установить на любом диске.

Для программы типа .COM легко определить местонахождение в файле программы на диске место, зарезервированное для ключа. Проще всего было бы расположить ключ в самом начале программы, однако так поступить нельзя, так как



программа типа COM обязана начинаться с выполнимой строки, а не с данных. Если, однако, начать программу таким образом:

```
text    segment 'code'
        assume cs:text, ds:text
        org     100h
```

```
myproc  proc
        jmp short entry
```

```
cluster dw 0 ; Установочный номер первого кластера файла .COM
entry:
```

то в образе программы на диске двухбайтовая команда `jmp short entry` займет байты 0 и 1, а поле кластер попадет в байты 2 и 3 от начала файла (место, зарезервированное под директивой `ORG 100h`, в файле будет отсутствовать; это поле будет "развернуто" только в процессе загрузки программы в память). Таким образом, открыв файл и определив из соответствующего ему блока SFT номер начального кластера, надо записать этот номер в более общем случае, когда требуется загрузить программу типа .EXE. Дело в том, что файл типа .EXE начинается не с текста программы, а с заголовка, в котором хранятся важные для системного загрузчика характеристики программы. Размер заголовка (который указывается в самом заголовке) зависит от сложности программы. Таким образом, для того, чтобы найти в файле с программой ее поля данных, надо сначала прочитать первый блок заголовка и определить его общий размер. Второе затруднение может возникнуть, если сегмент данных не является в программе первым сегментом. В этом случае, чтобы попасть в сегмент данных, надо определить размеры сегментов, расположенных перед ним. Проще, конечно, расположить сегмент данных в начале программы.

В табл. 4.10. приведен формат заголовка файла загрузочного модуля типа .EXE в системе MS-DOS.

Для определения размера заголовка и, соответственно, истинного начала программы в файле .EXE (на диске, не в памяти!) следует прочитать первые 512 байтов файла и содержимое слова со смещением 08h умножить на 10h. Получится размер заголовка в байтах. Далее надо прочитать из файла весь заголовок и начало программы, после чего можно записать в обусловленную ячейку в начале программы найденных заранее ключ и перенести начало программы назад на диск. Однако при неизвестной заранее длине заголовка возникнут проблемы с чтением из файла - какой длины буфер отводить в программе? Это затруднение можно разрешить следующим образом. Прочитав первые 512 байтов заголовка в буфер размером 512

байтов и найдя размер заголовка в параграфах, делим это число на  $512/16=32$  и получаем размер заголовка в виде числа блоков по 512 байтов (размер заголовка всегда кратен 512). Если теперь мы последовательно прочитаем в тот же буфер полученное число блоков по 512 байтов, в буфере окажутся первые 512 байтов программы.

Таблица 4.10. Заголовок файла загрузочного модуля типа .EXE.

Смещение	Число байтов	Описание
00h	2	Признак файла типа .EXE (Коды символов MZ)
02h	2	Размер программы включая заголовок по модулю 512
04h	2	Размер программы включая заголовок в блоках по 512 байтов
06h	2	Число элементов таблицы настройки
08h	2	Размер заголовка в параграфах
0Ah	2	Число параграфов, требуемых программе дополнительно к ее образу на диске (при наличии в конце программы сегментов с неинициализированными данными)
0Ch	2	Максимальное число параграфов, требуемых программе дополнительно к ее образу на диске (по умолчанию FFFFh)
0Eh	2	Смещение сегмента стека от начала программы в параграфах
10h	2	Содержимое регистра SP при входе в программу
12h	2	Контрольная сумма
14h	2	Содержимое регистра IP при входе в программу
16h	2	Смещение сегмента команд от начала программы в параграфах
18h	2	Относительный адрес первого элемента настройки
1Ah	2	Число перекрытий (0 для резидентной части программы)
1Ch	...	Таблица настройки переменной длины

Запись ключа за логическими пределами файла. Как известно, DOS выделяет место под файлы целыми кластерами, в результате чего за логическим концом файла практически всегда имеется свободное пространство (до конца кластера). При копировании файла на другой диск реально переносятся только байты, соответствующие самому файлу, так как число копируемых байтов определяется логической длиной файла. Байты последнего кластера файла, находящиеся за логическими пределами файла, не копируются. Если в них записать ключ, то при копировании ключ исчезнет. Методика работы с программой не отличается от уже описанной. После записи рабочей программы на жесткий диск она устанавливается с помощью специальной установочной программы (хранящейся на дискете). Установочная программа открывает файл с рабочей программой, перемещает указатель файла на его конец и записывает ключ (одно



или несколько слов) за прежними пределами файла. Затем с помощью средств DOS файл укорачивается до прежней длины. В результате ключ оказывается физически прилегающим к файлу, но логически за его пределами. При использовании этого метода установочная программа должна перед записью ключа проанализировать длину файла. Если файл занимает целое число кластеров, его предварительное удлинение не нужно, иначе некуда будет записать часть следующего кластера, иначе некуда будет записать ключ. То же получится, если, скажем, при длине ключа 2 байта файл занимает целое число кластеров минус 1 байт. В этом случае файл также требует удлинения.

Рабочая программа после запуска выполняет те же операции, что и установочная (за исключением удлинения файла) и проверяет, записан ли известный ей ключ за концом файла. Ключевая дискета с нестандартным форматом. Достаточно надежный способ защиты программ от переноса на другие компьютеры заключается в использовании ключевой дискеты. В этом случае рабочая программа, находящаяся на жестком диске, перед началом работы проверяет наличие на дисковом диске с ключевой информацией. Для того, чтобы ключевую дискету нельзя было размножить с помощью команды DISKCOPY, осуществляющей копирование на физическом уровне, ключевая информация записывается на дорожке с нестандартным форматом, расположенной к тому же за пределами рабочего пространства диска. Такая ключевая дискета подготавливается специальной установочной программой, которая с помощью функции 05h прерывания BIOS 13h, форматирует, например, дорожку номер 40 (или 80) с размером сектора 256 байтов вместо 512 и записывает на нее заданный ключ. Рабочая программа перед началом работы выполняет чтение нестандартной дорожки и при отсутствии самой дорожки или ключа на ней аварийно завершается. Такой способ удобен тем, что пользователь, приобретя программный продукт с ключевой дискетой, может свободно переносить его с компьютера на компьютер и даже запускать его на нескольких компьютерах сразу, но не имеет возможности передать программный пакет третьему лицу (не лишая себя ключевой дискеты).

Очевидно, что для ужесточения защиты рассмотренные методы могут использоваться совместно в разнообразных комбинациях.

#### 4.5. Задачи по защите программ от копирования и несанкционированного использования

**Задача 4.13.** Защита программы от копирования путем привязки к ее местоположению на жестком диске.

Написать рабочую программу WORK1.COM, выводящую на экран некоторый текст. В начале выполнения программа определяет номер своего первого кластера и сравнивает его с числом, записанным в ее полях данных в процессе установки на дискете данной копии программы. Если эти числа не совпадают, программа аварийно завершается с выдачей соответствующего сообщения.

Написать установочную программу INSTALL1.EXE, которая определяет начальный кластер файла рабочей программы WORK1.COM и записывает его в поле данных рабочей программы.

Вызвать рабочую программу WORK1.COM, убедиться в том, что в неустановленном состоянии она аварийно завершается.

Установить конкретную копию программы WORK1.COM с помощью программы INSTALL1.EXE. Убедиться, что установленная рабочая программа функционирует нормально.

Скопировать рабочую программу в другой каталог. С помощью команд DOS COMP или FC убедиться в полной тождественности двух копий программ WORK1.COM. Проверить работу "незаконно скопированной" (неустановленной) и исходной, установленной копий программы WORK1.COM. Установить (с помощью установочной программы INSTALL1.EXE) вторую копию рабочей программы и проверить ее работу.

#### Программа WORK1.COM

;Рабочая программа. Перед запуском требует установки на жестком диске  
;Основные фрагменты программы

```

muproc proc
    jmp short entry ;Короткий переход - обход слова
                    ;cluster
cluster dw 0 ;Место для номера первого кластера
entry:
;Откроем файл с рабочей программой, чтобы он попал в таблицу файлов
mov AH,30h
mov AL,2;Чтение/запись
mov DX,offset fname
int 21h
jnc gol ;Файл открылся нормально
                    ;AX=дескриптор открытого файла
                    ;Файл не открылся
jmp notopen

gol:
;Найдем в JFT номер блока описания нашего файла в SFT
mov DI,18h ;ES:DI->JFT
add DI,AX ;ES:DI->наш элемент в JFT

```

```

mov     DI,ES:[DI]
xor     CH,CH
;Получим доступ к системной таблице файлов
;CL=индекс SFT
;CX=индекс SFT
mov     AH,52h
int     21h
les     DI,ES:[BX+4]
cmp     CX,ES:[DI+4]
jbe     here
sub     CX,ES:[DI+4]
;Да
;Нет, вычтем число блоков в этой
;SFT
;ES:DI->вторая SFT
les     DI,ES:[DI]
here:
;Наши ту SFT, в которой наш индекс
mov     AX,59
mul     CL
add     DI,AX
add     AX,ES:[DI+08h]
;Размер блока описания файла
;Теперь AX=длина CL блоков
;Прибавим размер заголовка
;ES:DI->первый блок описания файла
;ES:DI->блок описания нашего файла
;AX=истинный номер первого
;кластера файла
;Сравним с записанным в программе
;Номера кластеров не совпадают,
;программа не установлена
found:  cmp     AX,cluster
jne     notins

```

;Программа установлена, можно с ней работать  
 ;Выведем сообщение mes об этом  
 ;Далее должна идти содержательная часть  
 ;рабочей программы. У нас ее нет  
 ;Завершим программу обычным образом  
 outprog: mov AX,4C00h
 int 21h
 ;Аварийные сообщения
 ;Программа не установлена, выведем сообщение mes1
 notins:

jmp outprog
 ;Файл не открылся, выведем сообщение mes2
 notopen:

jmp outprog
 outprog: endp

;Поля данных в том же сегменте  
 fname db 'work1.com',0 ;Имя файла рабочей программы
 mes db 'Программа установлена и будет работать нормально',10,13
 mes1 db 'Программа не установлена и не может быть запущена',10,13
 mes2 db 'Файл WORK1.COM не открылся',10,13

## Программа INSTALL1.EXE

;Установочная программа, предназначенная для установки  
 ;на жестком диске рабочей программы WORK1.COM  
 ;Основные фрагменты программы  
 ;Откроем файл с рабочей программой, чтобы он попал в таблицу файлов

```

mov     AH,3Dh
mov     AL,2;Чтение/запись

```

```

mov     DX,offset fname
int     21h
jnc     got
jmp     notopen
;Файл открылся нормально
;Файл не открылся
;Сохраним дескриптор
got:    mov     handle,AX
;Найдем в JFT номер блока описания нашего файла в SFT
mov     DI,18h
add     DI,AX
mov     CL,ES:[DI]
xor     CH,CH
;Получим доступ к системной таблице файлов
mov     AH,52h
int     21h
les     DI,ES:[BX+4]
cmp     CX,ES:[DI+4]
jbe     here
sub     CX,ES:[DI+4]
;Да
;Нет, вычтем число блоков в этой
;SFT
;ES:DI->вторая SFT
les     DI,ES:[DI]
;Наши ту SFT, в которой наш индекс
here:   mov     AX,59
mul     CL
add     DI,AX
add     AX,ES:[DI+08h]
;Размер блока описания файла
;Теперь AX=длина CL блоков
;Прибавим размер заголовка
;ES:DI->первый блок описания файла
;ES:DI->блок описания нашего файла
;AX=номер первого кластера файла
;Сохраним номер первого кластера файла
mov     word ptr buffer,AX
;Сдвинем указатель в файле WORK1.COM на начало кластера
mov     AH,42h
mov     AL,0
mov     BX,handle
mov     CX,0
mov     DX,2
int     21h
;Функция установки указателя
;Режим - от начала файла
;Дескриптор открытого файла
;Байт номер 2
;от начала файла
;Запишем номер кластера в предназначенное для него поле
;в файле WORK1.COM
mov     AH,40h
mov     BX,handle
mov     CX,2
mov     DX,offset buffer
int     21h

```

;Выведем сообщение mes о нормальном завершении установки  
 ...  
 ;Завершим программу обычным образом  
 outprog:
 ...  
 ;Аварийное сообщение об отказе открыть файл
 notopen:

```

jmp     outprog
;Поля данных
buffer db 2 dup (?)

```





```

mov     bx, handle      ;Первый блок заголовка
mov     cx, 512
mov     dx, offset buffer
mov     int 21h          ;Прочтали из заголовка
int     21h             ;размер заголовка в параграфах
mov     ax, word ptr buffer+8 ;Умножим на 16 сдвигом
                                ;АХ=размер заголовка в байтах
                                ;Сохраним в стеке
mov     cl, 4
mov     ax, cx
shl     ax
push    ax              ;Установим указатель файла за концом заголовка
                                ;(на начало сегмента данных)
mov     ah, 42h
mov     al, 0
mov     bx, handle      ;Режим - от начала файла
mov     cx, 0           ;Старшая половина указателя
mov     dx, 0           ;Заберем из стека размер заголовка
pop     dx
int     21h             ;Запишем ключ (номер первого кластера файла) на диск
mov     ah, 40h
mov     bx, handle
mov     cx, 2
mov     dx, offset cluster
int     21h
;Выведем сообщение mes о нормальном завершении процесса установки
mov     ah, 40h
mov     bx, 1
mov     cx, meslen
mov     dx, mes
lea     dx, mes
int     21h
;Завершим программу обычным образом
outprog:
;Выведем аварийное сообщение об отказе открыть файл
notopen:
...
jmp     outprog
;Поля данных
buffer db 512 dup (?)
fname db 'work2.exe', 0 ;Имя файла в формате файловых функций DOS
handle dw 0
mes db 'Программа WORK2.EXE установлена на жестком диске', 10, 13
mes2 db 'Файл WORK2.EXE не открылся', 10, 13

```

Задача 4.15. Защита пространство кластера за пределами файла ключа в свободное пространство кластера за пределами файла с программой.

Написать рабочую программу WORK3.EXE, выводящую на экран некоторый текст. В начале выполнения программа считывает первое слово за логическим концом файла WORK3.EXE, считывает из него ключ и сравнивает с ключом, записанным в программе. Если ключи не совпадают, программа аварийно завершается с выдачей соответствующего сообщения.

Написать установочную программу INSTALL3.EXE, которая записывает за логическим концом файла рабочей программы WORK3.EXE заданный ключ, предварительно проанализировав длину файла WORK3.EXE и удливив его на два байта (т.е. фактически на целый кластер), если за пределами файла, но до конца последнего занятого файлом кластера, нет места для записи ключа.

Методика проверки созданного программного комплекса такая же, как и в задаче 4.13.

### Программа WORK3.EXE

Рабочая программа WORK3.EXE, требующая установки на жестком диске  
Основные фрагменты программы  
Откроем файл с рабочей программой

```

...
jnc     go1
jmp     notopen
mov     handle, ax
;Установим указатель на 2 байта за концом файла
mov     ah, 42h
mov     al, 2
mov     bx, handle
mov     cx, 0
mov     dx, 2
int     21h
;Выполним фиктивную запись 0 байтов, чтобы удлинить файл на два байта
mov     ah, 40h
mov     bx, handle
mov     cx, 0
mov     dx, offset key_rd ;Фиктивный параметр
int     21h
;Сдвинем указатель на прежний конец файла, т.е. на ключ
mov     ah, 42h
mov     al, 2
mov     bx, handle
mov     cx, -1
mov     dx, -2
int     21h
;Прочитаем ключ
mov     ah, 3Fh
mov     bx, handle
mov     cx, 2
mov     dx, offset key_rd
int     21h
;Сравним прочитанный ключ с записанным в программе
mov     ax, key
cmp     ax, key_rd
je      ok
;Ключи совпадают
;Ключи не совпадают, выведем сообщение mes1 о том,
;что программа не установлена
...
jmp     ahead
;Продолжим программу

```

ok: ;Выведем сообщение mes в нормальной работе

;Независимо от наличия или отсутствия ключа файла в рабочей программой ;следует вернуть в исходное состояние, укоротив на два байта, которые ;ны к нему прибавили

;Сдвинем указатель на прежний конец файла  
ahead: mov ah, 42h  
mov al, 2  
mov bx, handle  
mov cx, -1  
mov dx, -2  
mov int 21h

;Выполним фиктивную запись 0 байтов, чтобы зафиксировать ;текущую длину файла  
mov ah, 40h  
mov bx, handle  
mov cx, 0  
mov dx, offset key\_rd ;Фиктивный параметр  
mov int 21h

;Завершим программу обычным образом

;Выведем сообщение mes2 об отказе открыть файл  
notopen:

```

... jmp outprog
;урог endp
;Пояа данных
fname db 'work3.exe', 0 ;Имя файла
handle dw 0 ;Ячейка для дескриптора
key dw 1234h ;Ключ, записываемый в файл
key_rd dw 0 ;Ключ, читаемый из файла
mes db 'Программа установлена и будет работать нормально', 10, 13
mes1 db 'Программа не установлена и не может быть запущена', 10, 13
mes2 db 'Файл WORK3.EXE не открылся', 10, 13
mes3 db 'Файл WORK3.EXE не найден в таблице файлов'

```

### Программа INSTALL3.EXE

;Установочная программа, предназначенная для установки ;на жестком диске рабочей программы WORK3.EXE  
;Основные фрагменты программы  
;Откроем файл с рабочей программой, чтобы он попал в таблицу файлов

```

... jmp gol
;ur notopen
gol: mov handle, AX ;Сохраним дескриптор
;Получим индекс SFT из JFT, находящейся в PSP
mov DI, 18h ;ES:DI->JFT
add DI, AX ;ES:DI->наш элемент в JFT
mov CL, ES:[DI] ;BL=индекс SFT
xor CH, CH ;BX=индекс SFT
;Получим доступ к системной таблице файлов
mov AH, 52h

```

```

int 21h
les DI, ES:[BX+4] ;ES:DI->первая SFT
cmp CX, ES:[DI+4] ;Индекс в этой SFT?
jb here ;Да
sub CX, ES:[DI+4] ;Нет, вычтем число блоков в этой SFT
les DI, ES:[DI] ;ES:DI->вторая SFT
here: ;Нашли ту SFT, в которой наш индекс
mov AX, 59 ;Размер блока описания файла
mul CL
add DI, 6 ;ES:DI->первый блок описания файла
mov AX, ES:[DI+11h] ;ES:DI->блок описания нашего файла
mov DX, ES:[DI+13h] ;AX=младшее слово длины
les DI, ES:[DI+7] ;DX=старшее слово длины
mov BL, ES:[DI+4] ;ES:DI=указатель на DPB
;BL=номер старшего сектора
;кластера
inc BL ;BL=число секторов в кластере
xor BH, BH ;BX=число секторов в кластере
mov CL, 9 ;Сдвиг на 9 бит = умножение на 512
shl BX, CL ;BX=число байтов в кластере
mov SI, BX ;Сохраним размер кластера в SI
div BX ;DX:AX/BX. Частное в AX,
;остаток в DX
push DX ;Сохраним в стеке остаток
;Установим указатель на конец файла
mov AH, 42h
mov AL, 2 ;Режим установки от конца файла
mov BX, handle
mov CX, 0 ;0 байтов
mov DX, 0 ;от конца файла
int 21h
;Нужна ли коррекция длины файла? Файл надо удлинить,
;если остаток равен 0 или равен длине кластера-1.
pop DX ;Извлечен из стека остаток
dec SI ;SI=число байтов в кластере-1
cmp DX, 0 ;Остаток равен 0?
je incr ;Да, на коррекцию
cmp DX, SI ;Остаток равен длине кластера-1?
je incr ;Да, на коррекцию
jmp good ;коррекция не требуется
;Удлиним файл на два байта, чтобы он занял еще один кластер
incr: mov AH, 40h
mov BX, handle
mov CX, 2
mov DX, offset null
int 21h
jmp good ;На запись ключа за пределами
;новой длины файла
good:
;Запишем за пределами файла ключ
mov AH, 40h

```

```

mov     BX,handle
mov     CX,2
mov     DX,offset key
mov     21h
int     21h
;Сдадим указатель на прежний конец файла, чтобы "отсечь" ключ от файла
mov     AH,42h
mov     AL,2
mov     BX,handle
mov     CX,-1
mov     DX,-2
mov     21h
int     21h
;Выполним фиктивную запись 0 байтов, чтобы укоротить файл
;до прежней длины
mov     AH,40h
mov     BX,handle
mov     CX,0
mov     DX,offset key
mov     21h
int     21h
;Выведем сообщение mes о нормальной работе
...
;Завершим программу обычным образом
...
notopen:
;Выведем сообщение mes2 о невозможности открыть файл
...
jmp outprog
;Поля данных
fname    db     'work3.exe',0
handle   dw     0
key      dw     1234h
null1    dw     0
mes      db     'Программа WORK3.EXE установлена на жестком диске',10,13
mes2     db     'Файл WORK3.EXE не открылся',10,13

```

Задача 4.16. Защита программ от копирования с помощью ключевой дискеты с нестандартным форматированием.

Написать программу, форматирующую на дискете емкостью 360 Кбайт дополнительную 41-ю дорожку дискеты (с номером 40) так, чтобы на ней были секторы нестандартного размера (например не 512, а 256 байтов). Записать в первый сектор этой нестандартной дорожки некоторый ключ.

Написать программу, которая перед выполнением считывает ключ с нестандартной дорожки и анализирует его правильность.

### Программа INSTALL4.EXE

;Программа нестандартного форматирования дискеты  
и запись на нее ключа

;Основные фрагменты программы

;Установим тип дискеты (только для машин типа AT)

```
mov     AH,17h ;Функция установки типа дискеты
```

```

mov     AL,02
mov     DL,0
int     13
;Найдем и сохраним адрес таблицы параметров дискеты (вектор
;прерывания 1Eh)
mov     AX,351Eh
int     21h
mov     word ptr dpt,BX
mov     word ptr dpt+2,ES
mov     DI,BX
;Установим новый размер сектора - 256 байтов вместо 512
mov     byte ptr ES:[DI+3],1
;Отформатируем дополнительную, 41-ю дорожку
mov     AH,05
mov     CH,40
mov     DH,0
mov     DL,0
push    DS
pop      ES
lea     BX,afd
int     13h
;Записываем на дорожку ключ (размером минимум в целый сектор)
mov     AH,03
mov     AL,1
mov     CH,40
mov     CL,1
mov     DH,0
mov     DL,0
lea     BX,key
int     13h

```

;Восстановим таблицу параметров дискеты  
mov ES,word ptr dpt+2  
mov byte ptr ES:[DI+3],2

```

;Поля данных
dpt      dd     0
afd      db     40,0,1,1
          db     40,0,2,1
          db     40,0,3,1
          db     40,0,4,1
          db     40,0,5,1
          db     40,0,6,1
          db     40,0,7,1
          db     40,0,8,1
          db     40,0,9,1
key       dw     9999h
          db     254 dup (0)

```

### Программа WORK4.EXE

;Программа чтения нестандартно отформатированной дискеты

;Основные фрагменты программы

;Установим тип дискеты - только для AT



;Найдем и сохраним адрес таблицы параметров дискеты (вектор типа 1Eh)  
;Установим размер сектора нестандартной дорожки

```

;Прочитаем первый сектор 41-й дорожки
mov     AL,02
mov     CH,40
mov     CL,1
mov     DH,0
mov     DL,0
push    DS
pop     BX,buf
lea     13h
int     noform
jc      word ptr buf,9999h
cmp     nokey
jne     ;Не читается
;Сравним ключи
;Неправильный ключ

```

;Адрес буфера в ES:BX

;Выведем сообщение mes 0 о нормальной работе

outprog:  
;Завершим программу обычным образом, восстановив сначала  
;таблицу параметров дискеты

```

mov     ES,word ptr dpt+2
mov     byte ptr ES:[DI+3],2

```

noform:  
;Выведем сообщение mes2 о несоответствии форматов

jmp outprog

nokey:  
;Выведем сообщение mes3 о несоответствии ключа ожидаемому

jmp outprog

```

;Поля данных
dpt     dd     0
buf     db     256 dup (0)
mes     db     'Ключевая дискета установлена.',10,13
        db     'Программа может работать',10,13
mes1en  equ     $-mes
mes2     db     'Установлена НЕ ключевая дискета',10,13
mes2len  equ     $-mes2
mes3     db     'На дискете неправильный ключ',10,13
mes3len  equ     $-mes3

```

## 5. Ввод информации с клавиатуры терминала

### 5.1. Системные процедуры обработки прерываний от клавиатуры

Работая на компьютере, пользователю постоянно приходится вводить с клавиатуры команды и данные. Если пользователь не запускал никаких программ, активной программой системы является командный процессор COMMAND.COM. Именно он воспринимает команды, адресованные DOS, проверяет их правильность и либо выполняет их сам (если введена команда DOS из числа внутренних), либо запускает запрошенную командой программу (одну из внешних команд DOS или прикладную программу, не входящую в состав DOS). Для уверенной работы с машиной полезно понимать, каким образом вводятся, куда падают и как обрабатываются символы, вводимые с клавиатуры. Процесс взаимодействия системы с клавиатурой продемонстрирован на рис. 5.1.

Работой клавиатуры управляет специальная электронная схема - контроллер клавиатуры. В его функции входит распознавание нажатой клавиши и помещение закрепленного за ней кода в свой выходной регистр (порт), обычно с номером 60h. Код клавиши, поступающий в порт, называется скен-кодом и является, по существу, порядковым номером клавиши, хотя последовательность скен-кодов не всегда совпадает с порядком расположения клавиш на клавиатуре. При этом каждой клавише присвоены как бы два скен-кода, отличающиеся друг от друга на 80h. Один скен-код (меньший, код нажатия) засылается контроллером в порт 60h при нажатии клавиши, другой (большой, код отпущения) - при ее отпущении.

Скен-код однозначно указывает на нажатую клавишу, однако по нему нельзя определить, работает ли пользователь на нижнем или верхнем регистре, а также вводит ли он латинские или русские буквы. С другой стороны, скен-коды присвоены всем клавишам клавиатуры, в том числе управляющим клавишам <Shift>, <Ctrl>, <Alt>, <Caps Lock> и др. Таким образом, очевидно, что определение введенного символа должно включать в себя не только считывание скен-кода нажатой

клавиши, но и выяснение того, не были ли перед этим нажаты, например, клавиши **<Shift>** (верхний регистр) или **<Caps Lock>** (фиксация верхнего регистра). Всем этим анализом занимается программа обработки прерываний от клавиатуры.

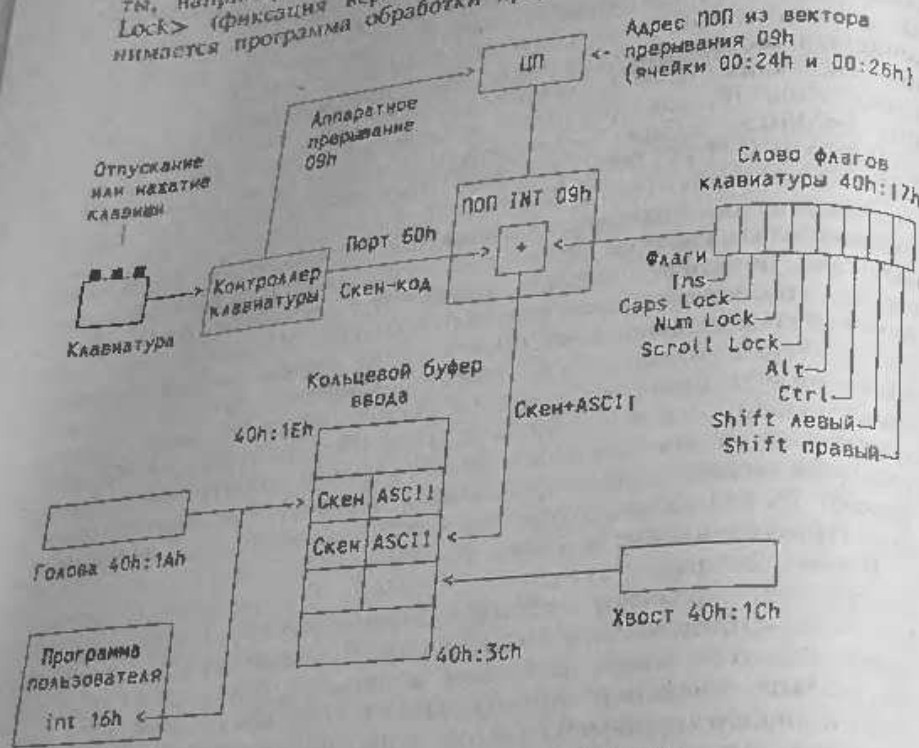


Рис. 5.1. Процесс взаимодействия системы с клавиатурой.

Нажатие (а также и отпускание) любой клавиши вызывает сигнал аппаратного (внешнего) прерывания, заставляющий процессор прервать выполняемую программу и перейти на программу обработки прерывания (ПОП) от клавиатуры. Эта программа хранится по фиксированному адресу в постоянном запоминающем устройстве BIOS, являясь, таким образом, элементом "встроенного", или "защитного" программного обеспечения.

Процессор вместе с сигналом прерывания получает еще и номер вектора прерывания. За клавиатурой закреплен вектор с номером 09h. Адрес программы обработки прерываний от клавиатуры располагается, таким образом, в векторе 09h, занимающем слова с адресами 24h и 26h.

Получив тип прерывания и определив по нему адрес векто-

Поскольку программа обработки прерываний от клавиатуры вызывается через вектор 09h, ее иногда называют программой INT 09h (INT - от английского Interrupt, прерывание).

Программа INT 09h, помимо порта 60h, работает еще с двумя областями оперативной памяти: кольцевым буфером ввода-вывода, располагаемым по адресам от 40h:1Eh до 40h:3Dh, куда в том состоянии (словом флагов) клавиатуры, находящимся по адресу 40h:17h, где фиксируется состояние управляющих клавиш (**<Shift>**, **<Caps Lock>**, **<Num Lock>** и др.).

Программа INT 09h, получив управление в результате прерывания от клавиатуры, считывает из порта 60h скен-код и анализирует его значение. Если скен-код принадлежит одной из нажатых, в слове флагов клавиатуры устанавливается бит (флаг), соответствующий нажатой клавише. Например, при нажатии клавиши **<Shift>** в слове флагов устанавливается бит 0, при нажатии левой клавиши **<Shift>** - бит 1, при нажатии любой клавиши **<Ctrl>** - бит 2, а при нажатии **<Alt>** пока клавиши (по одиночке или в любых комбинациях) остаются нажатыми. Если управляющая клавиша отпускается, программа INT 09h получает скен-код отпускания и сбрасывает соответствующий бит в слове флагов.

Кроме состояния указанных клавиш, в слове флагов фиксируются еще режимы **<Scroll Lock>**, **<Num Lock>**, **<Caps Lock>** и **<Insert>**, а в 101-клавишной клавиатуре на компьютерах PC/AT также состояния клавиш **<SysRq>**, **<Ctrl>**-левая, **<Alt>**-левая и режим паузы (**<Ctrl>**/**<Num Lock>**).

При нажатии любой другой клавиши программа INT 09h считывает из порта 60h ее скен-код нажатия и по таблице трансляции скен-кодов в коды ASCII формирует двухбайтовый код, старший байт которого содержит скен-код, а младший - код ASCII. При этом если скен-код характеризует клавишу, то код ASCII определяет закрепленный за ней символ. Поскольку за каждой клавишей закреплено, как правило, не менее двух символов ("a" и "A", "l" и "!", "-" и "\_" и т.д.), то каждому скен-коду соответствуют, как минимум, два кода ASCII. В процессе трансляции программа INT 09h анализирует состояние флагов, так что если нажата, например, клавиша Q (скен-код 10h, код ASCII буквы Q - 51h, а буквы q - 71h), то формируется двухбайтовый код 1071h, но если клавиша Q нажата при нажатой клавише **<Shift>** (смена регистра), то результат трансляции составит 1051h. Тот же код 1051h получится, если при нажатии клавиши Q был включен режим **<Caps Lock>** (заглавные буквы), однако при включенном режиме **<Caps**

Lock> и нажатой клавишей <Shift> образуется код 1071h, помещаемый в такой ситуации клавиша <Shift> на время нажатия переводит клавиатуру в режим нижнего регистра (строчные буквы).

Полученный в результате трансляции двухбайтовый код записывается программой INT 09h в кольцевой буфер ввода, который служит для синхронизации процессов ввода данных с клавиатуры и приема их выполняемой компьютером программой. Объем кольцевого буфера составляет 15 слов, при этом дисциплина его обслуживания такова, что коды символов извлекаются из него в том же порядке, в каком они в него поступали. За состоянием буфера следят два указателя. В хвостовом указателе (слово по адресу 40:1Ch) хранится адрес первой свободной ячейки, в головном указателе (40:1Ah) - адрес самого старого кода, принятого с клавиатуры и еще не востребованного программой. В начале работы, когда буфер пуст, оба указателя - и хвостовой, и головной, указывают на первую ячейку буфера.

Программа INT 09h, сформировав двухбайтовый код, помещает его в буфер по адресу, находящемуся в хвостовом указателе, после этого этот адрес увеличивается на 2, указывая опять на первую свободную ячейку. Каждое последующее нажатие на какую-либо клавишу добавляет в буфер очередной двухбайтовый код и смещает хвостовой указатель.

Выполняемая программа, желая получить код нажатой клавиши, должна вызвать прерывание INT 16h, которое активизирует драйвер клавиатуры BIOS. Драйвер считывает из кольцевого буфера содержимое ячейки, адрес которой находится в головном указателе, и увеличивает этот адрес на 2. Таким образом, программный запрос на ввод с клавиатуры фактически выполняет прием кода не с клавиатуры, а из кольцевого буфера.

Хвостовой указатель, перемещаясь по буферу в процессе занесения в него кодов, доходит, наконец, до конца буфера (адрес 40h:3Ch). В этом случае при поступлении очередного кода адрес в указателе не увеличивается, а, наоборот, уменьшается на длину буфера. Тем самым указатель возвращается в начало буфера, после чего продолжает перемещаться по буферу до его конца, опять возвращается в начало и так далее по кольцу. Аналогичные манипуляции выполняются и с головным указателем.

Равенство адресов в обоих указателях свидетельствует о том, что буфер пуст. Если при этом программа вызвала прерывание INT 16h, то драйвер клавиатуры будет ждать поступления кода в буфер, после чего он будет передан в программу. Если же хвостовой указатель, перемещаясь по буферу в

процессе его заполнения, подошел к головному указателю "с обратной стороны" (это происходит, если оператор нажимает на клавиши, а программа не выполняет запросы к драйверу клавиатуры), прием новых кодов блокируется, а нажатие на клавиши возбуждает предупреждающие звуковые сигналы.

Если компьютер находится в пассивном состоянии ожидания команд DOS с клавиатуры, то за состоянием кольцевого буфера ввода следит командный процессор COMMAND.COM. Как только в буфере появляется код символа, командный процессор с помощью соответствующих системных программ переносит его в свой внутренний буфер командной строки, очищая при этом кольцевой буфер ввода, а также выводит символ на экран, организуя режим эхо-контроля. При получении кода клавиши <Enter> (0Dh) командный процессор предполагает, что ввод команды закончен, анализирует содержимое своего буфера и приступает к выполнению введенной команды. При этом командными процессор работает практически лишь с младшими байтами ASCII.

Если компьютер выполняет какую-либо программу, ведущую диалог с оператором, то, как уже отмечалось, ввод данных с клавиатуры (а точнее - из кольцевого буфера ввода) и вывод их на экран с целью эхо-контроля организует эта программа, обращаясь непосредственно к драйверу BIOS (INT 16h) или к соответствующей функции DOS (INT 21h). Может случиться, однако, что выполняемой программе не требуется ввод с клавиатуры, а оператор нажал какие-то клавиши. В этом случае вводимые символы накапливаются (с помощью программы INT 09h) в кольцевом буфере ввода и, естественно, не отображаются на экране. Так можно ввести до 15 символов. Когда программа завершится, управление будет передано COMMAND.COM, который сразу же обнаружит наличие символов в кольцевом буфере, извлечет их оттуда и отобразит на экране. Такой ввод с клавиатуры называют вводом с упреждением.

До сих пор речь шла о символах и кодах ASCII, которым соответствуют определенные клавиши терминала и которые можно отобразить на экране. Это буквы (прописные и строчные), цифры, знаки препинания и специальные знаки, используемые в программах и командных строках, например, !, \$, # и др. Однако имеется ряд клавиш, которым не назначены какие-то отображаемые на экране символы. Это, например, функциональные клавиши <F1>, <F2>...<F10>; клавиши управления курсором <Home>, <End>, <PgUp>, <PgDn>, <Стрелка вправо>, <Стрелка вниз> и др. Очевидно, что всем



этим клавишам назначены определенные скен-коды. Но как их скен-коды транслируются в коды ASCII?

В таблице трансляции, с которой работает программа INT 09h, всем таким скен-кодам соответствует нулевой код ASCII. Поэтому при нажатии, например, клавиши <F1> (скен-код 3Bh) в кольцевой буфер ввода поступает двухбайтовый код 3B00h, а при нажатии клавиши <Home> (скен-код 47h) - двухбайтовый код 4700h. Двухбайтовые коды, содержащие на месте кода ASCII ноль, называются расширенными кодами ASCII. Эти коды (и соответствующие им клавиши) широко используются для управления программами. Например, в оболочке DOS Norton Commander нажатие функциональных клавиш вызывает выполнение определенных операций: <F1> - вывод на экран справочника, <F5> - копирование файла, <F8> - его удаление и т.д.

Программы, работающие с расширенными кодами ASCII, должны, очевидно, считав из кольцевого буфера ввода младший байт и убедившись, что он равен нулю, считать далее и старший байт (в сущности скен-код) и в зависимости от его значения выполнить соответствующее действие.

Широкое использование возможностей интерактивных средств потребовало расширения возможностей ввода с клавиатуры управляющей информацией, которую программа должна легко отличать от вводимого текста. С этой целью в компьютерах типа IBM PC расширенные коды ASCII генерируются не только функциональными клавишами и клавишами управления курсором, но и всеми алфавитно-цифровыми клавишами, если они нажимаются вместе с клавишей <Alt>. Таким образом, если нажатие клавиши Q посылает в кольцевой буфер двухбайтовый код 1071h (или 1051h на верхнем регистре), то нажатие сочетания <Alt>/Q генерирует расширенный код ASCII 1000h, где 10 - скен-код клавиши, а 00 - признак расширенного кода ASCII. Расширенные коды ASCII генерируются также при нажатии клавиш <Alt>, <Ctrl> или <Shift> одновременно с функциональными клавишами <F1>...<F10>. В этом случае, однако, в старший байт расширенного кода ASCII помещается уже не скен-код клавиши, а некоторый код, специально назначенный этой комбинации клавиш. Естественно, этого кода нет среди "обычных" скен-кодов. Например, клавиша <F1>, скен-код которой равен 3Bh, может генерировать следующие расширенные коды ASCII:

<F1> - 3B00h  
 <Shift>/<F1> - 5400h  
 <Ctrl>/<F1> - 5E00h  
 <Alt>/<F1> - 6B00h

На рис. 5.2 приведены скен-коды клавиш, а на рис. 5.3 - 5.6 - информационные (младшие) байты расширенных кодов ASCII, образуемых программой обработки прерываний от клавиатуры при нажатии клавиш и их сочетаний.

Некоторые клавиши расширенной (101-клавишной) клавиатуры генерируют при нажатии не один, а два или даже несколько сигналов прерываний, каждое из которых сопровождается посылкой в порт 60h своего кода. Такие клавиши помечены на рис. 5.2 звездочками; соответствующие им последовательности скен-кодов приведены в табл. 5.1. Некоторые клавиши (например, Ins или Del), если их нажать при нажатой управляющей клавише (Shift, Ctrl, Alt), генерируют специфическую последовательность скен-кодов; эти последовательности также включены в приводимый перечень.

Таблица 5.1. Последовательности скен-кодов некоторых клавиш и их сочетаний.

Клавиша	Скен-последовательность
Pause	E1h 10h 45h E1h 90h C5h
Ins	E0h 52h
Shift-Ins	E0h AAh E0h 52h
Home	E0h 47h
Shift-Home	E0h AAh E0h 47h
PgUp	E0h 49h
Shift-PgUp	E0h AAh E0h 49h
Del	E0h 53h
Shift-Del	E0h AAh E0h 53h
End	E0h 4Fh
Shift-End	E0h AAh E0h 4Fh
PgDn	E0h 51h
/	E0h 35h
Shift-/	E0h AAh E0h 35h
Стрелка вверх	E0h 48h
Shift-Стрелка вверх	E0h AAh E0h 48h
Стрелка влево	E0h 4Bh
Shift-Стрелка влево	E0h AAh E0h 4Bh
Стрелка вниз	E0h 50h
Shift-Стрелка вниз	E0h AAh E0h 50h
Стрелка вправо	E0h 4Dh
Shift-Стрелка вправо	E0h AAh E0h 4Dh
Enter	E0h 1Ch
Alt. правая	E0h 38h
Ctrl. правая	E0h 1Dh
PrtScr	E0h 2Ah E0h 37h
Shift-PrtScr	E0h 37h
Ctrl-PrtScr	E0h 37h

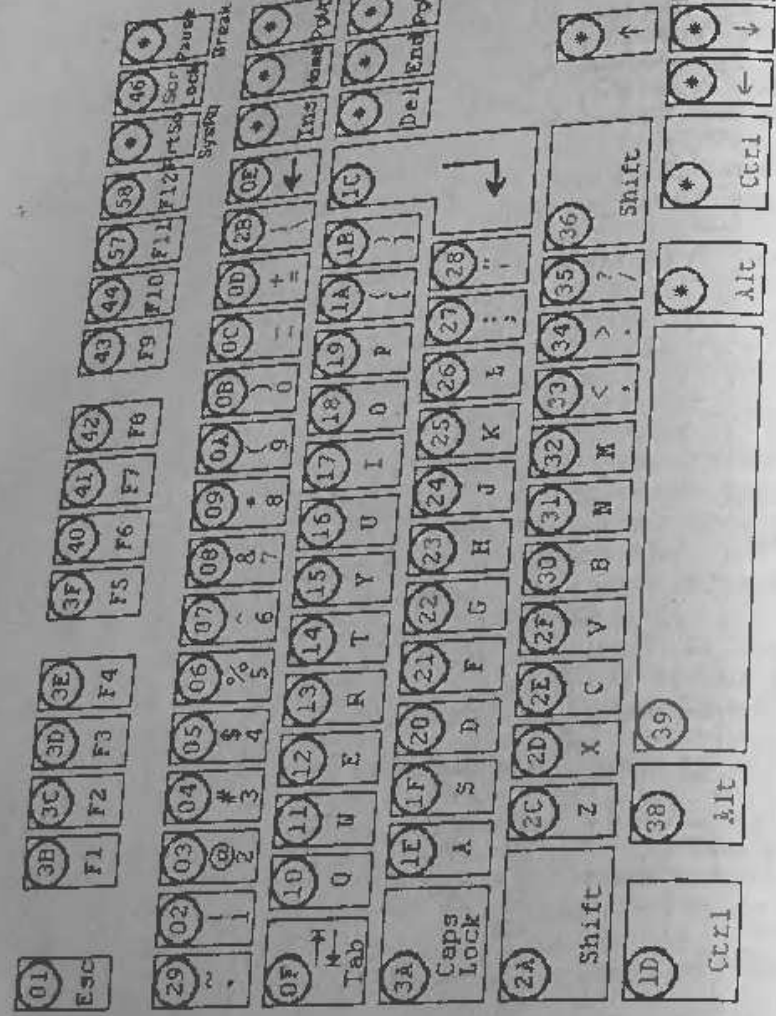


Рис 5.2. Скан-коды клавиш (числа шестнадцатеричные)  
 \* См. пояснения в тексте

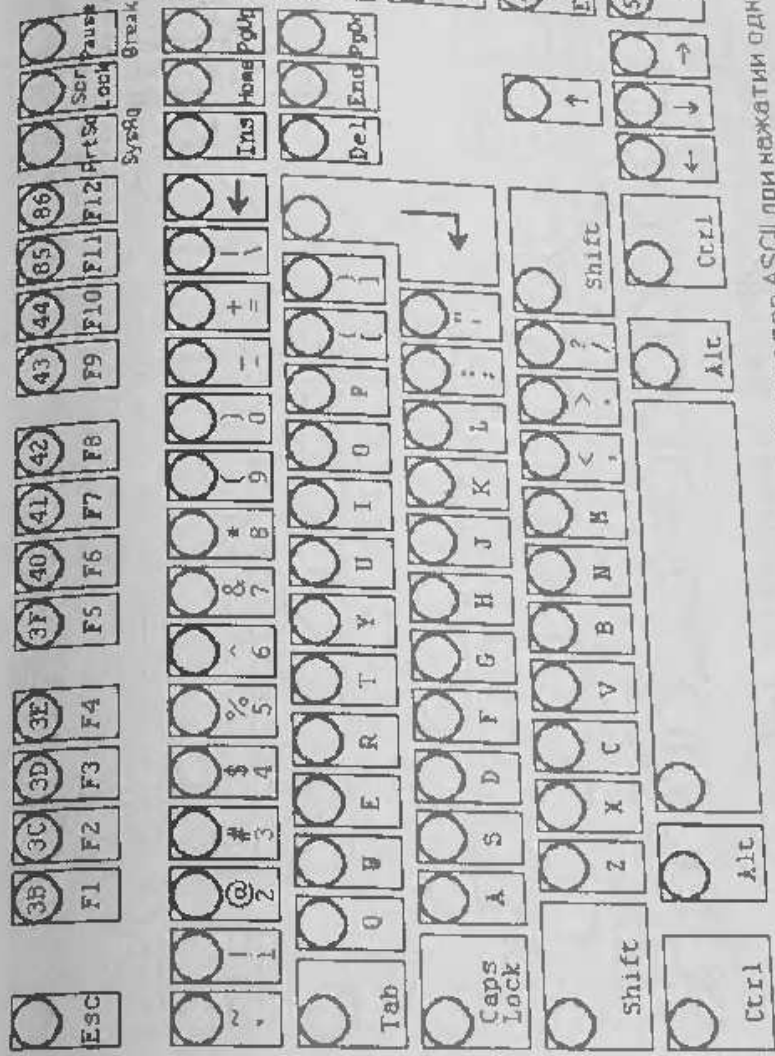


Рис 5.3. Информационные байты расширенных кодов ASCII при нажатии одной клавиши (числа шестнадцатеричные)

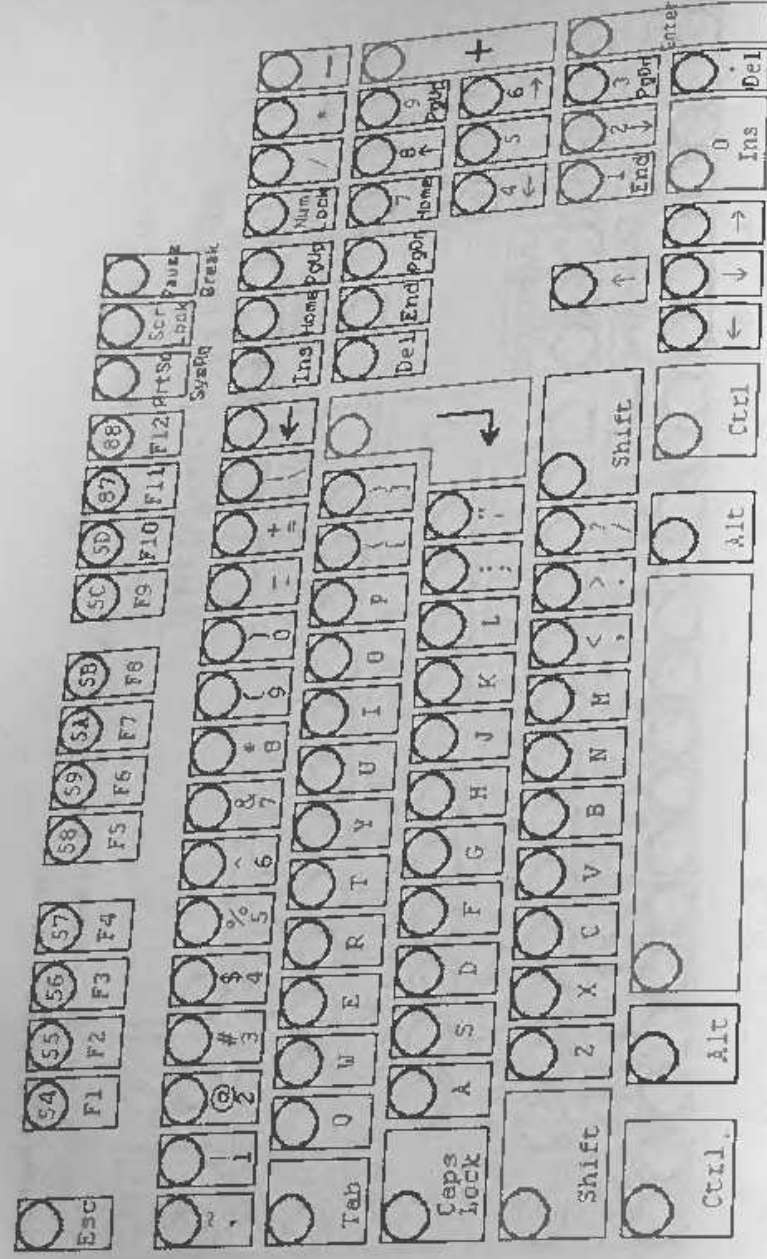


Рис 5.4 Информационные байты расширенных кодов ASCII при нажатии Shift/клавиша (числа шестнадцатеричные)

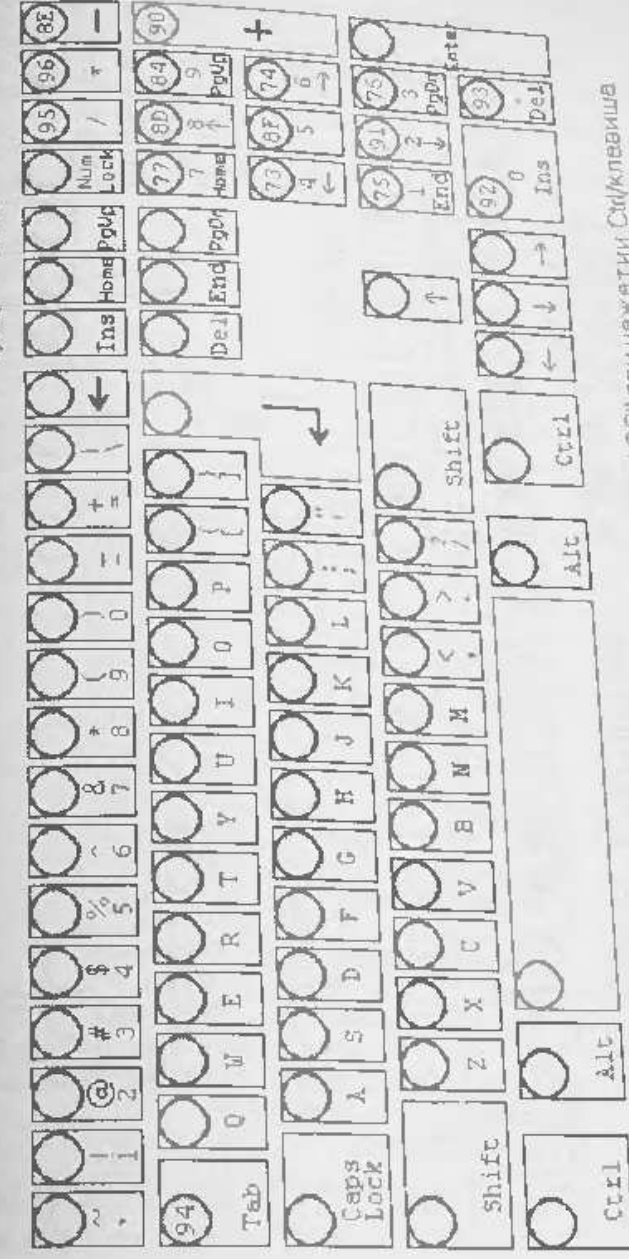
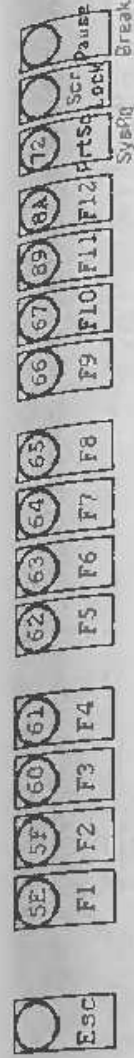


Рис 5.5 Информационные байты расширенных кодов ASCII при нажатии Ctrl/клавиша (числа шестнадцатеричные)



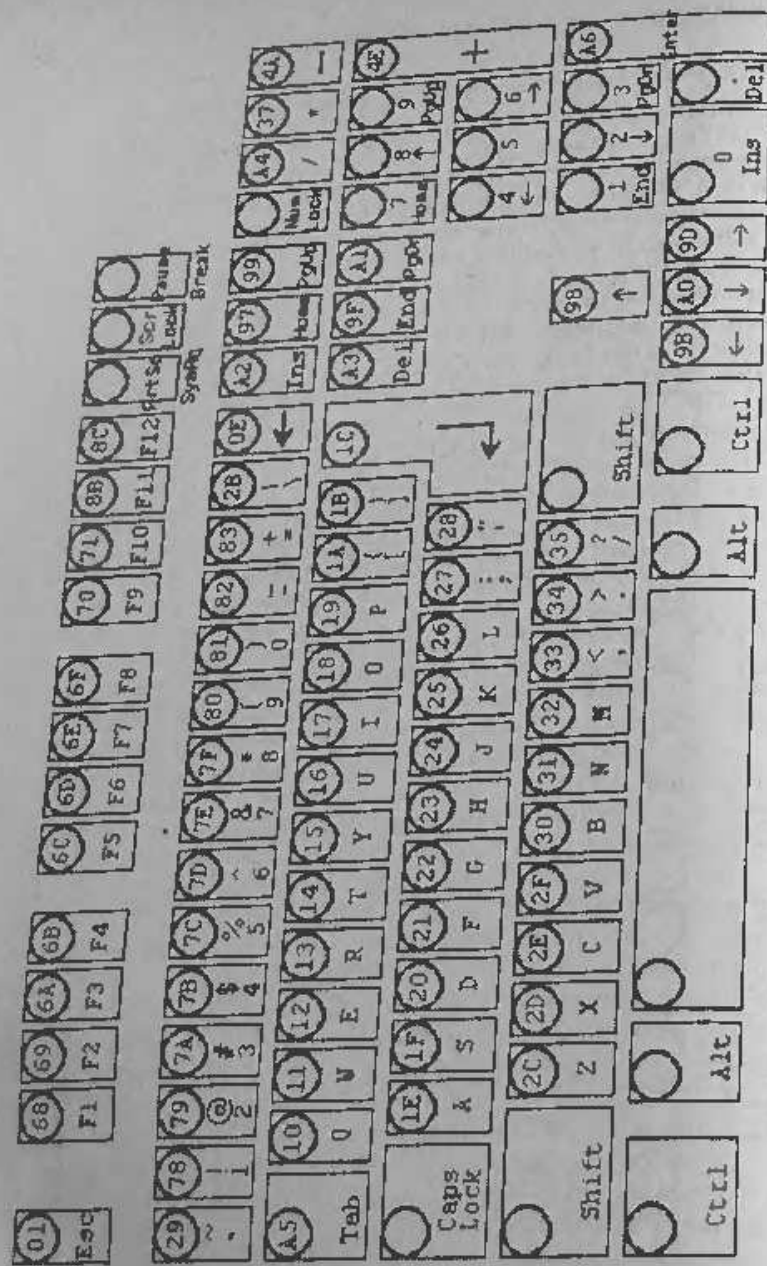


Рис. 5.6. Информационные байты расширенных кодов ASCII при нажатии A/клавиша (числа шестнадцатеричные)

5.2. Системные средства ввода данных с клавиатуры

Операционная система предоставляет несколько способов ввода данных с клавиатуры:

- обращение клавиатуры, как к файлу, с помощью прерывания DOS INT 21h с функцией 3Fh;
- использование группы функций DOS INT 21h из диапазона 1...Ch, обеспечивающих посимвольный ввод с клавиатуры в различных режимах;
- посимвольный ввод пульт...

Ввод с клавиатуры средствами файловой системы (INT 21h, функция 3Fh) осуществляется точно так же, как и чтение из закрепленного за стандартным устройством ввода (по умолчанию за клавиатурой). Число вводимых символов указывается в регистре CX, однако ввод завершается лишь после того, как нажата клавиша <Enter>, независимо от того, введено ли фактически меньше символов, чем было запланировано, или больше (последнее, естественно, может случиться лишь при неправильных действиях). Поэтому при вводе строк с клавиатуры нет необходимости заранее задавать их длину, достаточно загрузить в регистр CX максимальную длину строки, например, 80 байт. В любом случае в регистре AX возвращается число реально введенных байтов, при этом учитываются также и два байта (0Ah и 0Dh), поступающие во входной буфер при нажатии клавиши <Enter>.

Особая ситуация возникает, если попытаться ввести больше символов, чем затребовано функцией 3Fh. В процессе выполнения этой функции все вводимые символы тут же извлекаются из кольцевого буфера ввода и пересылаются в буфер DOS. Обнаружив во входном потоке коды клавиши <Enter>, DOS пересылает из этого буфера в буфер пользователя в программе точно затребованное число символов (естественно, без кодов <Enter>, которые располагаются в конце вводимой строки). Остальные символы остаются в буфере DOS, готовые к вводу. Фактически, если не принять специальных мер к очистке буфера, они поступят в программу при очередном запросе 3Fh, даже если оператор еще не начал вводить очередную порцию данных. Очевидно, что в этом случае будет нарушена синхронизация хода выполнения программы с работой оператора.

Поскольку дескриптор 0 закреплен не собственно за клавиатурой, а за стандартным устройством ввода, он обеспечивает перенаправление ввода. Пусть, например, программа PROCDAT содержит строки ввода через дескриптор 0. При запуске программы командой

## PROCSTAT

она будет ожидать ввода исходных данных с клавиатуры, однако запуск ее командой

PROCSTAT < RAWDAT.001

приведет к автоматическому вводу информации из файла RAWDAT.001, который в данном случае ищется в текущем каталоге текущего диска. Поиск и открытие файла осуществляет DOS.

Иногда требуется выключить механизм перенаправления (возможно, лишь для определенных операторов ввода). Для этого следует открыть консоль для ввода как файл (с именем CON) функцией 3Dh, получить выделенный системой дескриптор, а затем использовать его в операциях ввода 3Fh:

:Поля данных		:Имя устройства
keybd db	'CON', 0	:Новый дескриптор
handle dw	0	
:Программные строки		
:Открыть новый дескриптор		:Функция открытия
mov	AX, 3Dh	:Доступ для чтения
mov	AX, 0	:Адрес имени устройства
mov	DX, offset keybd	
int	21h	
mov	handle, AX	:Получили дескриптор

Второй способ получения данных с клавиатуры в программу, с помощью функций DOS из диапазона 1...Ch, несколько более громоздок, но обеспечивает более разнообразные возможности. Вообще все функции DOS разделяются на две группы: функции 1...Ch (все они обеспечивают ввод-вывод через стандартные дескрипторы) и все остальные функции с номерами 0 и Dh...6Ch. Различие между указанными группами функций заключается в том, что при их выполнении DOS работает на разных стеках - стеке ввода-вывода, если вызываются функции из первой группы и стеке, получившим название дискового, при выполнении функций второй группы. Наличие двух стеков обеспечивает частичную реентерабельность (повторную входимость) DOS. Это свойство очень важно при обработке аппаратных прерываний и будет детально рассмотрено в последующих главах. В программах, не связанных с обработкой внешних аппаратных прерываний, принадлежность функции той или иной группе не имеет значения, и при выборе для использования в программе следует руководствоваться чисто функциональными соображениями.

Для ввода с клавиатуры можно использовать 7 функций прерывания INT 21h:

01h - ввод символа с эхом;

06h - прямой ввод - вывод через консоль;  
 07h - ввод символа без эха и без отработки Ctrl/C;  
 08h - ввод символа без эха и с обработкой Ctrl/C;  
 0Ah - буферизованный ввод строки с эхом;  
 0Bh - проверка состояния стандартного устройства ввода;  
 0Ch - сброс входного буфера и ввод.

Функции 01h, 06h, 07h и 08h при каждом вызове вводят в программу один символ из кольцевого буфера ввода; при необходимости ввести группу символов (строку) функции следует использовать в цикле. Различаются эти функции наличием или отсутствием эха, а также реакцией на ввод с клавиатуры сочетания <Ctrl>/C. Функции 01h и 0Ah отображают вводимые символы на экране (эхо); функции 07h и 08h этого не делают, что дает возможность вводить данные тайком от окружающих (например, пароль или ключ). Второе важное различие <Ctrl>/C. При выполнении функций 01h и 0Ah DOS проверяет каждый введенный символ и, обнаружив во входном потоке код <Ctrl>/C (03h), аварийно завершает программу. Функции 06h и 07h пропускают код <Ctrl>/C в программу, не инициализируя по нему никаких специальных действий. Такой метод ввода используется прикладными программами, если перед завершением в них должны быть выполнены определенные пролов и проч.). Аварийное завершение такой программы средством DOS по коду <Ctrl>/C могло бы привести к нарушению ее работоспособности.

Функция 0Ah передает в буфер пользователя строку, введенную с клавиатуры; строка должна заканчиваться нажатием клавиши <Enter>. Длина строки может достигать 254 символов. Вводимые символы отображаются на экране; при вводе <Ctrl>/C происходит аварийное завершение программы.

Функция 0Bh позволяет проверить наличие в кольцевом буфере ввода ожидающих символов. При обнаружении символов программа должна извлечь их из буфера одной из функций ввода; если символов нет, программа может продолжить выполнение. Такая методика используется в программах, носящих циклический характер, если требуется обеспечить управление ходом выполнения программы с клавиатуры терминала. В каждом шаге цикла после выполнения запланированных действий проверяется состояние кольцевого буфера ввода; если в течение предыдущего шага цикла оператор нажал на какую-либо клавишу, программа анализирует введенный код и осуществит выход из цикла и переход в ту или иную точку; если же буфер оказывается пуст, циклическое выполнение продолжится.



Функция 0Bh чувствительна к <Ctrl>/C. Это дает возможность организовать с ее помощью аварийное завершение программы на тех ее участках, где выполняются чисто процессорные действия. Если, например, включить вызов функции 0Bh в цикл, то при отсутствии ввода с клавиатуры цикл будет выполняться обычным образом, но после ввода <Ctrl>/C программа аварийно завершится, хотя на выполняемом участке программы не используются функции ввода-вывода.

Функция 0Ch служит для организации ввода с предварительной очисткой кольцевого буфера.

Все функции, кроме 0Ch, вводят в программу наиболее стабильный из скопившихся в кольцевом буфере ввода символов, реагируя тем самым на возможность ввода с упреждением. В этом режиме оператор может нажимать на клавиши еще до выдачи программой запроса на ввод; коды нажатых клавиш (не более 15) будут накапливаться в кольцевом буфере ввода и извлекаться оттуда в программу по мере выполнения ею запросов на ввод. В отличие от этого, функция 0Ch сначала очищает кольцевой буфер и лишь затем ожидает ввода символа с клавиатуры. В результате коды всех ранее нажатых (по предположению - случайно) клавиш теряются. Обычно функция ввода 0Ch стоит в программе непосредственно вслед за функцией вывода на экран символьной строки с предложением оператору ввести данные. В результате из кольцевого буфера убирается весь "мусор" от случайных нажатий, в программу же поступает лишь то, что вводится оператором после запроса программы. При этом режим ввода (с эхом или без него и т.д.) определяется тем, какая именно функция ввода (01h, 07h, 08h или 0Ah) реализуется "внутри" функции 0Ch.

Все функции DOS ввода с клавиатуры допускают перенаправление ввода (из файла, последовательного порта, из вывода другой программы). Если требуется избавиться от этого качества, следует использовать файловую функцию ввода 3Fh и специально выделенный дескриптор.

Функции 01h, 07h, 08h и 0Ah являются синхронными, т.е. при отсутствии символа в кольцевом буфере ждут его ввода. Функция 06h позволяет определить состояние кольцевого буфера и при наличии в нем кода извлечь этот код и обработать его, а при отсутствии - продолжить выполнение программы.

Функции 01h, 06h, 07h и 08h позволяют вводить в программу расширенные коды ASCII. Для этого, обнаружив, что введенный код ASCII равен нулю, следует выполнить функцию повторно. Это дает возможность управления прикладными программами с помощью функциональных клавиш, а также сочетаний <Alt>/цифра, <Alt>/буква и др.

Функции 06h, 07h и 08h позволяют вводить в программу коды символов с помощью сочетания <Alt>/<цифра на цифровой клавиатуре> (в том числе некоторые из первых 32 символов кодовой таблицы и всю вторую половину кодовой таблицы).

Сравнительные характеристики функций DOS ввода с клавиатуры приведены в табл. 5.2.

Таблица 5.2. Сравнительные характеристики функций DOS ввода с клавиатуры.

	01h	06h	07h	08h	0Ah	0Bh	0Ch
Эхо	+	-	-	-	+	-	-
Реакция на ^C	+	-	-	-	+	-	+/-
Перенаправление	+	+	+	+	+	+	+/-
Ожидание символа	+	-	+	+	+	+	+
Расширенные коды ASCII	+	+	+	+	+	-	+/-
<Alt>+код	-	+	+	+	-	-	+
Очистка буфера	-	-	-	-	-	-	+/-

Работа с клавиатурой на уровне BIOS (INT 16h) позволяет считывать двухбайтовые коды, поступающие в кольцевой буфер ввода (код ASCII + скен-код) и анализировать слово флагов клавиатуры (нажатие клавиш <Shift>, <Caps Lock> и др.). Для ввода используются следующие функции прерывания INT 16h:

00h - чтение двухбайтового кода из входного буфера;

01h - чтение состояния клавиатуры и двухбайтового кода без извлечения его из буфера;

02h - чтение флагов клавиатуры.

Функция 00h позволяет в одном действии получить полный двухбайтовый код нажатой клавиши или комбинации клавиш, из которого, в частности, можно извлечь скен-код (некоторые программы идентифицируют нажатые клавиши не по кодам ASCII, а по их скен-кодам), а также получить значащую часть расширенного кода ASCII (при нажатии, например, функциональных клавиш). Функция 00h является синхронной: при ее выполнении программа останавливается в ожидании нажатия клавиши.

Функция 01h относится к числу асинхронных: определив состояние клавиатуры (точнее - буфера ввода), она возвращает управление программе. Состояние буфера возвращается в флаге ZF: если в буфере имеются ожидающие ввода в программу символы, ZF=0, если же буфер пуст, ZF=1. При наличии в буфере кода символа его можно проанализировать, так как он возвращается функцией в регистре AX (AH=скен-код, AL=код ASCII). Необходимо однако иметь в виду, что функция 01h,



копируя двухбайтовый код в регистр AX, не очищает при этом кольцевой буфер. Забрать символ с очистки буфера можно затем функцией 00h.

Функция 02h - чтение флагов клавиатуры - передает в программу содержимое слова флагов (ячейка 417h). Она может использоваться программами, работающими на уровне скен-кодов, для определения состояния клавиш <Shift>, <Caps Lock> и др.

### 5.3. Задачи по программированию ввода с клавиатуры

Задача 5.1. Ввод с клавиатуры и вывод на экран символьной информации. Вывести на экран через дескриптор стандартной ошибки служебное сообщение. Ввести с клавиатуры через дескриптор стандартного ввода символьную строку, вывести ее на экран через дескриптор стандартного вывода.

```

;Определения
stdin=0
stdout=1
stderr=2
;Основные фрагменты программы
;Выведем служебное сообщение msg
...
;Поставим запрос на ввод строки в буфер buf
mov     AH,3Fh
mov     BX,stdin
mov     CX,80
mov     DX,offset buf
int     21h
mov     actlen,AX
;Выведем введенное на экран
mov     AH,40h
mov     BX,stdout
mov     CX,actlen
mov     DX,offset buf
int     21h
;Завершим программу
...
;Поля данных
msg     db      'Вводите! '
msglen=$-msg
buf     db      80 dup (0)
actlen  dw      0

```

Отладив программу, изучить с ее помощью действие операторов перенаправления ввода и вывода на примере следующих операций (программа из примера 5.1 названа P5-1):

1) P5-1 > F1.TST (перенаправление вывода программы в создаваемый системой файл F1.TST;

2) P5-1 >> F1.TST (перенаправление вывод программы в имеющийся файл с добавлением новой строки к содержимому файла);

3) P5-1 < F2.TST (перенаправление ввода программы с получением данных для нее не с клавиатуры, а из текстового файла F2.TST, который надо предварительно создать с помощью какого-либо текстового редактора);

4) P5-1 < F2.TST > F3.TST (перенаправление и ввода, и из одного файла и заносит их в другой (в сущности, реализация копирования символьных файлов)).

Задача 5.2. Ввод с клавиатуры, обработка и вывод на экран символьной информации. Усложнить программу из примера 5.1, предусмотрев фильтрацию входного символьного потока, например, преобразование строчных латинских (или русских, или и тех, и других) букв в прописные.

```

;Определения stdin, stdout, stderr
...
;Основные фрагменты программы
;Выведем служебное сообщение msg
...
;Поставим запрос на ввод строки в буфер buf, отправив длину
фактически введенной строки в actlen
...
;Превратим строчные латинские буквы в прописные
mov     CX,actlen
mov     SI,0
filter: mov     AL,buf[SI]
        cmp     AL,'a'
        jb      nolet
        cmp     AL,'z'
        ja      noletter
        sub     AL,20h
        mov     buf[SI],AL
        inc     SI
        loop    filter
nolet:   ;И отправим назад в buf
        mov     buf[SI],AL
;Выведем введенное на экран
...
;Завершим программу
...
;Поля данных
msg     db      'Вводите! '
msglen=$-msg
buf     db      80 dup (' ')
actlen  dw      0

```

Отладив программу и используя ее вместе с предыдущей, изучить процедуру конвейеризации программ на примере

следующих операций (программа из примера 5.1 названа P5-1, из примера 5.2 - P5-2):

- 1) P5-1 и P5-2 (перенаправление вывода программы P5-1 на вход программы P5-2 с ее автоматическим запуском после завершения P5-1);
  - 2) P5-1 и P5-2 > FILE1.TXT (перенаправление вывода программы P5-1 на вход программы P5-2, а ее вывода - в файл FILE1.TXT, создаваемый системой автоматически);
  - 3) P5-1 < FILE2.TXT и P5-2 (перенаправление ввода программы P5-1 для получения входных данных из файла FILE2.TXT и передача выходных данных этой программы на вход программы P5-2 с ее автоматическим запуском после завершения P5-1);
  - 4) P5-1 < FILE2.TXT и P5-2 > FILE3.TXT (ввод исходных данных в программу P5-1 из существующего файла FILE2.TXT, запуск после завершения P5-1 программы P5-2 с перезаписью ей вывода P5-1 и перенаправлением вывода P5-2 в создаваемый системой файл FILE3.TXT).
- Файл FILE2.TXT, содержащий, в частности, строчные буквы, следует создать либо с помощью той же программы P5-1, либо с помощью любого текстового редактора.

**Задача 5.3.** Посимвольный ввод с эхом. Ввести в программу один символ с клавиатуры. Проанализировать его. Если введено "Е", завершить программу; в случае ввода любого другого символа повторить ввод. Обратит внимание на реакцию программы на ввод прописной или строчной буквы "Е". После отладки программы проверить ее реакцию на ввод <Ctrl>/C. Исследовать работу программы в случае перенаправления ее ввода или вывода в файл. Исследовать реакцию программы на ввод с клавиатуры <Ctrl>/C в случае перенаправления ее ввода и вывода. Исследовать реакцию программы на ввод с упреждением, для чего сразу после ввода команды активизации программы, но еще до ее фактического запуска нажать на какие-либо клавиши.

;Основные фрагменты программы

;Введем символ и проанализируем его на код "Е"

;Если введено "Е", завершим программу

```
again: mov     AH,01h           ;Функция ввода с эхо
        int     21h
        cmp     AL,'E'         ;Введено "Е"?
        je      outprog       ;Да
        jmp     again
```

outprog:

;Завершим программу

**Задача 5.4.** Посимвольный ввод без эха. Замкнуть в примере 5.3 функцию ввода с эхом 01h на функцию фильтрованного ввода без эха 08h. После отладки проверить реакцию программы на ввод <Ctrl>/C. Для удобства работы предусмотреть в программе вывод на экран запроса на ввод символа.

;Основные фрагменты программы

;Выведен запрос req на ввод символа

```
...
;Введем символ и проанализируем его на код "Е"
;Если введено "Е", завершим программу
```

```
again: mov     AH,08h           ;Функция ввода без эха
        int     21h
        cmp     AL,'E'
        je      outprog       ;Введено "Е"?
        jmp     again         ;Да
```

outprog:

;Завершим программу

```
...
;Поля данных
req      db      'Вводите команду:'
reqlen=$-req
```

**Задача 5.5.** Управление программой от функциональных клавиш. С помощью функции 08h организовать ввод в программу управляющих кодов от функциональных клавиш <F1>...<F10> или других клавиш, дающих расширенные коды ASCII (сочетания <Alt>/<буква>, <Alt>/<цифра> и др.). Вводимые коды использовать для управления ходом программы.

;Основные фрагменты программы

;Ожидаем нажатия клавиши

```
again: mov     AH,08h           ;Функция ввода без эха
        int     21h
        cmp     AL,0           ;Младший байт кода = 0?
        jne     again         ;Нет, повторить
        mov     AH,08h
        int     21h
        cmp     AL,59         ;Да, введем старший байт кода
        je      f1            ;Нажата <F1>?
        cmp     AL,84         ;Да
        je      shiftf1       ;Нажаты <Shift>/<F1>?
        cmp     AL,30         ;Да
        je      alta         ;Нажаты <Alt>/A?
        cmp     AL,120        ;Да
        je      altl         ;Нажаты <Alt>/I?
        jmp     again         ;Да
        ;Нажата незапланированная
```

;Вывод соответствующих сообщений

f1:

;Вывод сообщения mf1

```
...
        jmp     outpr
```

```

shiftf1:
;Вывод сообщения mshift1
    jmp outpr

alt1:
;Вывод сообщения malt1
    jmp outpr

alt1:
;Вывод сообщения malt1
    jmp outpr

outpr:
;Завершим программу
    ...
;Поля данных
mf1 db 'Введено <F1>'
mf1len=$-mf1
mshf1 db 'Введено <Shift>/<F1>'
mshf1len=$-mshf1
malt1 db 'Введено <Alt>/A'
malt1len=$-malt1
malt1len=$-malt1

```

**Задача 5.6.** Ввод с клавиатуры с предварительной очисткой буфера. Организовать цикл ввода в программу данных по ее запросу. Анализировать введенный символ. Если введено "Q", завершить программу. После отладки проверить реакцию программы на ввод с упреждением. Проверить возможность управления программой кодами, вводимыми через цифровую клавиатуру (при нажатой клавише <Alt>).

;Основные фрагменты программы  
;Выведен запрос req на ввод символа

```

;Очистим буфер ввода и поставим запрос на ввод с клавиатуры
again: mov AH,0Ch ;Функция ввода с очисткой буфера
        mov AL,01h ;Выберем функцию ввода 01h
        int 21h ;(можно 01h, 06h, 07h, 08h)

;Проанализируем введенное
cmp AL,'Q' ;Введено "Q"?
je outprog ;Да
jmp again

```

outprog:
;Завершим программу
 ...

```

;Поля данных
req db 10,13,'Введите команду:'
reqlen=$-req

```

**Задача 5.7.** Чтение двухбайтового кода из кольцевого буфера ввода. Программа, несмотря на примитивность, позволяет наглядно ознакомиться с принципами трансляции скен-кодов программой INT 09h. Для этого программу следует запустить в отладчике CodeView, установить режим индикации регистров процессора и, многократно выполняя строки программы, наблюдать, какие двухбайтовые коды образуются при нажатии различных клавиш и их комбинаций. Любопытно посмотреть коды таких клавиш, как <Esc>, <Tab>, <PgDn>, <Home>, <End>, клавиш со стрелками, а так же сочетание управляющих (<Ctrl>, <Alt>, <Shift>) и "обычных" клавиш.

```

;Основные фрагменты программы
;Будем ждать ввода символа
again: mov AH,00h
        int 16h
        jmp again

```

;Функция чтения двухбайтового кода

**Задача 5.8.** Управление циклической программой от клавиатуры с помощью функции чтения состояния клавиатуры. Организовать выход из циклического участка программы при нажатии на любую клавишу. При желании можно усложнить задачу, введя в программу анализ нажатой клавиши и переход на то или иное продолжение программы в зависимости от результатов анализа. В приведенном примере такой анализ отсутствует. Любопытно также удалить из программы строки извлечения символа из кольцевого буфера (функция 00h) и проанализировать работу такого варианта программы.

;Определение stdout=1 ;Дескриптор стандартного вывода  
;Основные фрагменты программы  
;Организуем цикл каких-нибудь действий с периодическим опросом клавиатуры.  
;Конкретные действия - вывод символа на экран с небольшой задержкой

```

again: mov AH,40h ;Функция вывода
        mov BX,stdout ;Дескриптор стандартного вывода
        mov CX,1 ;Один символ
        mov DX,offset sym ;Адрес символа
        int 21h

```

;Организуем небольшую задержку

```

;Получим состояние клавиатуры
mov AH,01h ;Функция чтения состояния
int 16h ;клавиатуры
jz again ;Если Z=1, символа нет

;Сообщим о выходе из цикла с помощью функции 40h
    ...

```

```

;Заберем введенный символ из кольцевого буфера ввода
mov AH,00h ;Получим символ в AX
int 16h ;и пусть он пропадет

```



Завершил программу

Поля данных  
куб db  
мат db

Программа завершена оператором

## 6. Вывод текстовой информации на экран терминала

### 6.1. Вideosистема компьютеров типа IBM PC

Вideosистема компьютера включает в себя ряд аппаратных и программных средств, позволяющих получать на экране терминала текстовые и графические изображения.

К аппаратным средствам можно отнести сам видеотерминал (монитор) - высококачественную монохромную или цветную электронно-лучевую трубку со схемами питания и управления, а также видеоконтроллер, или видеоадаптер - электронную плату, обеспечивающую вывод на экран текстовых и графических изображений, а также программное управление вideosистемой. Качество и возможности изображения в значительной степени определяются характеристиками видеоадаптера.

В настоящее время широко используется адаптер EGA (Enhanced Graphics Adapter, улучшенный графический адаптер). Адаптер обеспечивает два программно переключаемых режима - текстовый и графический.

В графическом режиме на экран выводится 16-цветное изображение с разрешением 640x350 точек, причем в памяти адаптера могут одновременно храниться два независимых изображения (два графические страницы).

В текстовом режиме изображение обычно состоит из 25 строк по 80 символов в строке, хотя имеется возможность увеличивать число строк до 40 за счет уменьшения высоты отображаемых символов. Каждый символ и фон под ним могут принимать любой из 16 цветов.

Поскольку таблицы, описывающие форму символов, загружаются в память адаптера программно, имеется возможность работать с символами любой конфигурации (например, греческие буквы). Обычно используется стандартная кодовая таблица символов, содержащая знаки английского и русского алфавитов, цифры, знаки препинания, специальные машинные знаки, символы псевдографики для рисования в текстовом режиме разнообразных рамок, а также некоторые математические символы.

В памяти адаптера одновременно может храниться до 8 текстовых страниц (8 "экранов"). Переключение как текстовых, так и графических страниц осуществляется программно.

Программные средства обслуживания экрана включают в себя видеодрайвер BIOS, к которому можно обратиться из прикладной программы с помощью прерывания Int 10h, и который обеспечивает нижний уровень управления (вывод символов, работа с курсором, переключение режимов видеoadapterа и т.д.), а также программы DOS, активизируемые с помощью прерывания Int 21h и предоставляющие более высокий уровень сервиса в текстовом режиме. Работа в графическом режиме не поддерживается DOS и может осуществляться только с помощью функций видеодрайвера BIOS.

## 6.2. Вывод на экран средствами DOS

DOS предоставляет следующие возможности вывода текстовой информации на экран:

- обращение к экрану как к файлу, с помощью прерывания DOS Int 21h с функцией 40h;
- использование группы функций DOS из диапазона 1...Ch (прерывание Int 21h), реализующих посимвольный вывод, а также вывод строк.

Вывод на экран средствами файловой системы (Int 21h, функция 40h) осуществляется точно так же, как и запись в файл. Используются предопределенные дескрипторы 1 или 2, закрепленные за стандартными устройствами вывода и ошибки, соответственно (по умолчанию - экраном). Число выводимых символов указывается в регистре CX, а адрес выводимой строки - в регистре DX. Коды 08h (забой), 0Ah (перевод строки), 0Dh (возврат каретки) и некоторые другие рассматриваются, как управляющие и приводят к выполнению соответствующих им действий. Число выводимых символов передается через регистр CX, однако, если в строке встречается <Ctrl>/Z (код 26), вывод прекращается. Дескриптор 1, закрепленный за стандартным устройством вывода, обеспечивает перенаправление вывода. Пусть, например, программа FILTXT содержит строки вывода через дескриптор 1. При запуске программы командой

```
FILTXT
```

ее вывод появится на экране, однако команда

```
FILTXT>MYFILE.DOC
```

приведет к автоматическому образованию файла MYFILE.DOC и записи в него всего вывода программы (на экран ничего не поступит); команда

```
FILTXT >> MYFILE.DOC
```

направит вывод программы FILTXT в тот же файл MYFILE.DOC, при этом новая запись добавится к уже существующим, увеличив тем самым длину файла MYFILE.TXT.

Предопределенный дескриптор 2 (стандартная ошибка) всегда связан с экраном и не может быть перенаправлен.

Если требуется исключить для каких-то операторов вывода или для всей программы в целом возможность перенаправления, то, помимо использования дескриптора стандартной ошибки 3Dh, получить выделенный системой дескриптор, а затем использовать его в операциях вывода 40h:

:Поля данных		'CON'.0	
screen	db	0	:Имя устройства
handlsr	dw	0	:Новый дескриптор
:Откроем	новый дескриптор		
	mov	AH,3Dh	
	mov	AL,1	:Функция открытия
	mov	DX,offset screen	:Доступ для записи
	int	21h	:Адрес имени устройства
	mov	handlsr,AX	
			:Получили дескриптор

Второй способ вывода на экран текстовой информации реализуется с помощью трех функций прерывания Int 21h:

- 02h - вывод символа;
- 06h - прямой ввод-вывод;
- 09h - вывод строки.

Функция 09h широко используется в системных программах (например, в драйверах) для вывода на экран информационных и диагностических сообщений. Перед вызовом прерывания адрес сообщения записывается в регистр DX; заканчивается сообщение символом "\$". В сообщение могут быть включены управляющие коды (возврата каретки, перевода строки, забоя и др.), а также Esc-последовательности. Так же, как и при выводе через дескриптор 1 (функция 40h), вывод функцией 09h поступает на стандартное устройство вывода и при запуске программы может быть перенаправлен на другие внешние устройства с помощью операторов перенаправления.

Если в процессе вывода сообщения на экран с клавиатуры поступает код <Ctrl>/C, срабатывает стандартная процедура обработки этого прерывания и вывод завершается (как и вся программа в целом). Для надежной обработки прерывания по <Ctrl>/C следует включать режим BREAK (командой DOS BREAK ON).

Функция 02h вызывает передачу на экран (точнее - на стандартное устройство вывода) одного символа, помещаемого в

регистр DL. Для вывода строки функцию следует использовать в цикле. В остальном она не отличается от функции 09h (перенаправление, обработка управляющих кодов, реакция на ввод с клавиатуры <Ctrl>/C).

Функция 06h (прямой ввод-вывод через консоль) используется в тех случаях, когда надо исключить стандартную реакцию системы на ввод с клавиатуры <Ctrl>/C. В остальном она действует так же, как функции 09h и 02h, однако обеспечивает не только вывод, но ввод. В случае вывода код ASCII передаваемого символа засылается в регистр DL; при вводе DL=FFh.

Средства DOS "в чистом виде" позволяют выводить на экран только черно-белый текст; возможности позиционирования текста на экране ограничиваются использованием символов возврата каретки (0Dh) и перевода строки (0Ah). Для вывода на экран средствами DOS цветных изображений следует использовать управляющие Esc-последовательности, реализуемые драйвером ANSI.SYS.

### 6.3. Управление экраном через ANSI-драйвер

Включение в систему ANSI-драйвера терминала (файл ANSI.SYS) дает пользователю дополнительные возможности управления экраном и клавиатурой. Если в символьной строке, выводимой на экран, встречается код клавиши <Esc> (27=1Bh), за которым следует символ [, то ANSI-драйвер перехватывает последующие символы и интерпретирует их, как команды управления экраном или клавиатурой. С помощью Esc-последовательностей можно очищать экран, перемещать по нему курсор, выбирать цвета фона и символа, изменять видеорежим, а также переопределять клавиши клавиатуры.

Группа Esc-последовательностей обеспечивает управление курсором. Так, Esc[2J очищает экран и перемещает курсор в левый верхний угол, т.е. действует подобно команде DOS CLS; последовательность Esc[строка;столбец] позиционирует курсор в заданной точке экрана т.д.

Управление цветом предполагает возможность задания трех параметров: цвета фона, цвета символа и дополнительной характеристики символа (мерцание, повышенная яркость и др.). Поэтому соответствующая Esc-последовательность имеет вид

Esc[код-1;код-2;код-3m

где код-1, код-2 и код-3 - коды трех указанных параметров в любом порядке. При необходимости можно указать только два или один параметр:

Esc[7m Символы становятся мерцающими;  
Esc[34;7m Голубые мерцающие символы;

Esc[47;7;34m Голубые мерцающие символы на белом фоне.

Esc-последовательность Esc[0m отменяет заданные ранее цвета и характеристики символов, возвращая комбинацию: белые символы на черном экране.

Esc-последовательности Esc[s (сохранение текущих координат курсора) и Esc[u (восстановление сохраненных координат курсора) используются для того, чтобы запомнить положение курсора, вывести что-то в другом месте экрана, а затем вернуть курсор в прежнюю позицию и продолжить вывод.

Следует иметь в виду, что ANSI-драйвер чувствителен к регистру клавиатуры, на котором вводится завершающая Esc-последовательность буква. Так Esc-последовательность Esc[2J завершается обязательно прописной буквой J, а Esc-последовательность Esc[0m - строчной буквой m.

Esc-последовательности часто используются в программах для формирования на экране цветных информационных кадров. В этом случае Esc-последовательности включаются в строки, выводимые на экран операторами вывода того языка, на котором написана программа (Паскаль, Си, Ассемблер и пр.). Это дает возможность выводить последовательные строки текста в разные места экрана, изменять их цвет, заставлять мигать или выделяться яркостью и т.д. Перечень Esc-последовательностей с кратким описанием приведен в табл. 6.1.

Таблица 6.1. Esc-последовательности для управления экраном и клавиатурой.

Последовательность	Действие
Esc[2J	Очистка экрана и перемещение курсора в левый верхний угол
Esc[K Esc[стр;позн	Очистка строки от курсора до конца строки Установка позиции курсора. Параметр стр обозначает Y-координату курсора в пределах 1-25 параметр поз - X - координату в пределах 1-80 (для видеорежима 80x25 символов). Вместо символа n в конце последовательности можно использовать символ f
Esc[кОдA Esc[кОдВ Esc[кОдС Esc[кОдD Esc[0n	Перемещение курсора на код строк вверх Перемещение курсора на код строк вниз Перемещение курсора на код позиций вправо Перемещение курсора на код позиций влево Вывод текущих координат курсора в формате Esc[стр;поз]
Esc[s	Сохранение текущих координат курсора в специальном буфере
Esc[u	Восстановление сохраненных в буфере координат курсора



Продолжение таблицы 6.1.

Последовательность	Действие
Esc[7h	Включение автоматического перевода курсора на следующую строку
Esc[7l	Выключение автоматического перевода курсора на следующую строку
Esc[код_1, код_2, код_3m	Выбор атрибутов символов. Возможные значения параметров код_1, код_2, код_3 приведены в табл. 6.2
Esc[кодh	Выбор видеорежима. Возможные значения параметра код приведены в табл. 6.3. Символ h можно заменить символом ?
Esc[код_ASCII;комp	Переопределение клавиш клавиатуры. Здесь код_ASCII - код ASCII переопределяемой клавиши; ком - заключенная в кавычки цепочка символов (команда), которая должна генерироваться при нажатии этой клавиши; латинская буква p - завершающий символ этой последовательности
Esc[0;код_ASCII;комp	Переопределение клавиши или сочетания клавиш, дающих расширенный код ASCII. Здесь код_ASCII - старший байт расширенного кода ASCII

Таблица 6.2. Возможные значения параметров Esc-последовательности Esc[код-1, код-2, код-3m задания атрибутов выводимого текста.

Код	Атрибуты выводимого текста
0	Нормальное изображение (белые символы на черном поле)
1	Выделение яркостью
4	Выделение подчеркиванием (только для монохромных дисплеев)
5	Выделение мерцанием
7	Инверсное изображение (черные символы на белом поле)
8	Скрытый текст (только для монохромных дисплеев)
30	Черные символы
31	Красные символы
32	Зеленые символы
33	Коричневые символы
34	Синие символы
35	Фиолетовые символы
36	Бирюзовые символы
37	Белые символы
40	Черный фон
41	Красный фон
42	Зеленый фон
43	Коричневый фон
44	Синий фон
45	Фиолетовый фон
46	Бирюзовый фон
47	Белый фон

Таблица 6.3. Возможные значения параметра Esc-последовательности Esc[кодh или Esc[кодl задания видеорежима.

Код	Видеорежим	Видеоадаптеры
0	40x25, 16-цветный текстовый режим	CGA, EGA, MCGA, VGA
1	40x25, 16-цветный текстовый режим	CGA, EGA, MCGA, VGA
2	80x25, 16-цветный текстовый режим	CGA, EGA, MCGA, VGA
3	80x25, 16-цветный текстовый режим	CGA, EGA, MCGA, VGA
4	320x200, 4-цветный графический режим	CGA, EGA, MCGA, VGA
5	320x200, 4-цветный графический режим	CGA, EGA, MCGA, VGA
6	640x200, 2-цветный графический режим	CGA, EGA, MCGA, VGA
7	80x25, монохромный текстовый режим	CGA, EGA, MCGA, VGA
14	640x200, 16-цветный графический режим	MDA, EGA, Hercules, VGA
15	640x350, монохромный графический режим	EGA, VGA
16	640x350, 16-цветный графический режим	EGA, VGA
17	640x480, монохромный графический режим	EGA, VGA
18	640x480, 16-цветный графический режим	MCGA, VGA
19	320x200, 256-цветный графический режим	VGA, MCGA

Все перечисленные выше функции DOS позволяют включать в состав выводимой строки Esc-последовательности. Например, для размещения в центре пустого экрана фразы "Ваши возможности", написанной красным цветом по белому полю, надо выполнить функцию 40h прерывания 21h (вывод в файл или устройство) с предопределенным дескриптором 1:

```
mov     AH,40h
mov     BX,1
mov     CX,message_length
mov     DX,offset message
int     21h
```

Функция 40h требует указания дескриптора в BX, длины строки в CX и адреса строки в DX. Саму строку message следует сформировать следующим образом:

```
message db 27,'[2J',27,'[12;32H',27,'[31;47m'
         db 'Ваши возможности:',27,'[0m'
```

В строку включены 4 Esc-последовательности:

Esc[2J - очистка экрана;  
Esc[12;32H - установка курсора (12-я строка, 32-й столбец);  
Esc[31;47m - установка цвета сообщения (красные символы по белому полю);  
Esc[0m - отмена цвета, чтобы остальные выводимые на экран строки оставались черно-белыми.

Еще проще вывести эту строку функцией 9 прерывания DOS (строку в этом случае следует завершить символом \$):

```

mov     AH, 09h
mov     SI, offset message
int     21h

```

#### 6.4. Логическая организация текстового видеобуфера

Текстовые страницы адаптера EGA располагаются в адресном пространстве компьютера (за пределами основной памяти) по следующим адресам:

```

страница 0 - B8000h...B8F40h
страница 1 - B9000h...B9F40h
страница 2 - BA000h...BAF40h
страница 3 - BB000h...BBF40h
страница 4 - BC000h...BCF40h
страница 5 - BD000h...BDF40h
страница 6 - BE000h...BEF40h
страница 7 - BF000h...BFF40h

```

Каждый символ занимает в буфере поле из двух байтов (рис. 6.1.). Младшие (четные) байты всех полей отводятся под коды ASCII отображаемых символов, старшие (нечетные) байты - под их атрибуты. Двухбайтовые коды символов записываются в видеобуфер в том порядке, в каком они должны появляться на экране: первые 80 двухбайтовых полей соответствуют первой строке экрана, вторые 80 полей - второй строке и т.д. Таким образом, переход на следующую строку определяется не управляющими кодами возврата каретки и перевода строки, а размещением кодов символов в другом месте буфера, в полях, соответствующих следующей строке. Вообще при формировании изображения непосредственно в видеобуфере, в обход программ DOS и BIOS, все управляющие коды ASCII теряют свои управляющие функции и отображаются в виде соответствующих им символов. Трактровка же, например, кода ASCII 9, как символа табуляции, или кода ASCII 10 - как символа перевода строки выполняется программы DOS, которые в данном случае не активизируются.

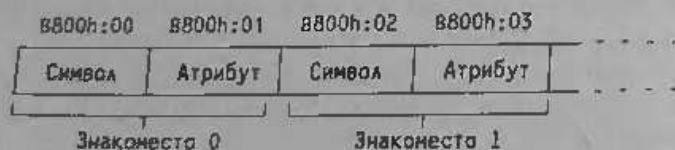


Рис. 6.1. Логическая организация текстового видеобуфера.

Атрибут символа определяет цвет символа и фона под ним, а также некоторые дополнительные характеристики

изображения на экране. Структура байта атрибутов приведена на рис. 6.2.

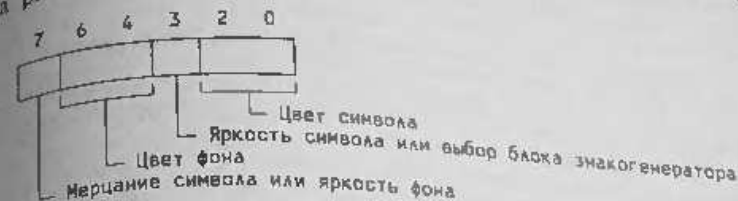


Рис. 6.2. Структура байта атрибутов.

В биты 0...2 байта атрибутов записывается код цвета символа, а бит 3 при исходной настройке видеоадаптера, действующей по умолчанию, управляет яркостью символа. Таким образом, каждый символ независимо от других может принимать любую из 16 возможных цветов. Соответствие кодов, записываемых в байте атрибута, конкретным цветам составляет палитру видеосистемы. Палитра устанавливается программно; по умолчанию действует назначение кодов, приведенное в табл. 6.4.

Таблица 6.4. Коды цветов, действующие по умолчанию

Значение кода	Цвет	Значение кода	Цвет
0	Черный	8h	Серый
1	Синий	9h	Голубой
2	Зеленый	Ah	Салатовый
3	Бирюзовый	Bh	Светло-бирюзовый
4	Красный	Ch	Розовый
5	Фиолетовый	Dh	Светло-фиолетовый
6	Коричневый	Eh	Желтый
7	Белый	Fh	Ярко-белый

Однако бит 3 байта атрибутов не обязательно управляет яркостью символов. С помощью подфункции 03 функции 10h драйвера BIOS (прерывание 10h) можно установить такой режим работы видеоадаптера, когда значение бита 3 (0 или 1) задает номер блока знакогенератора, определяющего конфигурацию символов, отображаемых на экране. В этом режиме на экран, в зависимости от значения бита 3 атрибута, выводится символ либо из одного, либо из другого блока, что позволяет увеличить количество различных символов до 512 (и пользоваться, например, в дополнение к русскому и английскому,

еще немецким и греческим алфавитами). Естественно, в этом режиме символы могут быть только 8 цветов.

Биты 4...6 байта атрибута задают цвет фона под данным символом. Что же касается последнего бита 7, то он, в зависимости от режима видеoadaptera, определяет либо яркость фона (и тогда фон может принимать 16 разных цветов), либо мерцание символа. Так, в режиме мерцания значение старшего полубайта атрибута 8h обозначает не серый фон, а черный фон при мерцающем символе (цвет которого по-прежнему определяется младшим полубайтом); значение Ch - не розовый, а красный фон при мерцающем символе и т.д.

Если бит 7 байта атрибута управляет мерцанием символов, то фон может принимать только 8 цветов, соответствующих левой половине приведенной выше таблицы. Для переключения назначения бита 7 предусмотрена подфункция 03h функции 10h драйвера BIOS (прерывание Int 10h). При включении компьютера устанавливается режим управления мерцанием.

### 6.5. Вывод на экран средствами BIOS

Рассмотрим теперь средства вывода на экран, реализуемые драйвером BIOS, программное обращение к которому осуществляется с помощью прерывания Int 10h. При работе в текстовом режиме обычно используются следующие функции драйвера:

- 02h - установить позицию курсора;
- 03h - получить позицию курсора;
- 05h - установить видеостраницу;
- 06h - инициализировать или прокрутить вверх окно;
- 07h - инициализировать или прокрутить вниз окно;
- 08h - прочитать символ и атрибут в позиции курсора;
- 09h - вывести символ и атрибут в позицию курсора;
- 0Ah - вывести символ в позицию курсора;
- 0Eh - вывести символ в режиме телетайпа;
- 10h, подфункция 03h - переключить бит мерцание/яркость;
- 13h - вывести строку в режиме телетайпа.

Функция 02h позволяет позиционировать текстовый курсор, задавая его местоположение в виде номера строки (0...24) и номера столбца (0...79). Видеодрайвер поддерживает 8 независимых курсоров - по одному на каждую страницу, причем функция 02h позиционирует курсор независимо от того, какая страница является активной.

Функция 03h позволяет получить и сохранить текущее положение курсора. Это дает возможность перейти временно в

другое место экрана, сформировать там изображение, а затем вернуться на старое место.

Функция 05h переключает видеостраницы дисплея. Если дисплей находится в текстовом режиме, то переключаются текстовые страницы (0...7), если установлен графический режим, то переключаются графические страницы (0...1).

Большая часть описываемых ниже функций вывода на экран, кроме подфункции переключения мерцания/яркости и функции вывода строки в режиме телетайпа, позволяют формировать изображение на любой видеостранице, как активной в настоящий момент, так и скрытой. Это даст возможность либо подготовить заранее несколько страниц и по мере необходимости быстро их переключать, либо, пока одна страница выводится на экран, готовить изображение на следующей.

С помощью функций 06h и 07h в заданном месте экрана дисплея создаются цветные прямоугольные окна заданного размера. Если в созданные ранее окна выведен какой-либо текст, то с помощью этих же функций можно прокручивать текст вверх или вниз. При этом текст, уходящий за край окна, пропадает, а из-под противоположного края появляются пустые строки с заданными атрибутами цвета. Для заполнения появляющихся строк текстом следует использовать подходящие функции DOS или BIOS, причем контроль местоположения, длины и цвета строк возлагается на программиста. Драйвер только прокручивает заданную прямоугольную область экрана (вместе с текстом в ней).

Функции 09h, 0Ah, 0Eh и 13h служат для вывода на экран отдельных символов и символьных строк (в цикле). Функции 09h и 0Ah не выполняют фильтрации управляющих символов, поэтому с их помощью можно выводить все символы кодовой страницы. Предусмотрен вывод одного и того же символа заданное число раз, что можно использовать при создании рамок и других орнаментов. Вывод символа не перемещает курсор, поэтому каждый раз перед применением функций 09h или 0Ah следует позиционировать курсор с помощью функции 02h. Различие функций 09h и 0Ah заключается в том, что первая позволяет вывести символ с любым атрибутом, а вторая использует прежний атрибут той позиции, куда выводится символ.

Функция 0Eh фильтрует управляющие коды 07h (звуковой сигнал), 08h (возврат на шаг), 10h (перевод строки) и 13h (возврат каретки), выполняя соответствующие им действия. Курсор перемещается после вывода каждого символа, что дает возможность выводить целые строки. Однако атрибут символа установить нельзя, выводимый символ приобретает прежний атрибут той позиции, куда он выводится. При необходимости вывода символа с новым атрибутом следует сначала вывести в



заданную позицию символ пробела с требуемым атрибутом (функцией 09h), а затем туда же послать символ с помощью функции 0Eh.

Важным свойством функции 0Eh является автоматический переход на следующую строку после завершения предыдущей, а также прокрутка экрана вверх на одну строку после заполнения самой нижней строки.

Функция 13h предназначена для вывода строк с указанием атрибутов как каждого символа в отдельности, так и всей строки. Функция может выполняться в четырех вариантах в зависимости от кода режима, указываемого в регистре AL. В режимах 0 и 1 атрибут символов указывается сразу для всей строки в регистре BL, причем в режиме 0 курсор не смещается в процессе вывода, а в режиме 1 - смещается на длину строки. В режимах 2 и 3 атрибуты символов включаются в выводимую строку, в которой, таким образом, чередуются коды атрибутов и коды символов, что усложняет формат строки, но позволяет устанавливать атрибуты для каждого символа независимо. Режим 2 отличается от режима 3 тем, что в первом случае курсор не смещается, а во втором смещается на длину строки.

При вызове функции 13h в регистре DX задаются координаты начала выводимой строки (в DH - строка экрана и в DL - столбец), а в регистре CX - длина выводимой строки, которая в режимах 2 и 3 оказывается за счет байтов с атрибутом в два раза больше длины строки, реально появляющейся на экране. Несколько необычно указывается адрес выводимой строки. Он должен быть помещен в регистры ES:BP (ES - сегментный адрес и BP - смещение в пределах сегмента).

Функция 13h выводит не все символы, так как коды 07h, 08h, 0Ah и 0Dh рассматриваются ею, как управляющие.

При выводе на экран средствами драйвера BIOS необходимо иметь в виду, что ввод с клавиатуры <Ctrl>/C не приводит к завершению программы. Следует опасаться бесконечных циклов вывода на экран - выход из них возможен только путем перезагрузки компьютера.

Подфункция 03h функции 10h (прерывание 10h), в отличие от описанных выше функций вывода символов и строк, воздействует сразу на весь экран, влияя на отображение тех символов, у которых установлен старший бит атрибута фона. Функция позволяет либо приписать этот бит яркости фона, давая тем самым возможность выводить на экран 16 цветов фона, либо назначить его атрибуту мерцания символа. В последнем случае цвет фона может принимать только 8 значений.

## 6.6 Задачи по программированию вывода на экран

**Задача 6.1.** Вывод на экран строки. Вывести на экран сообщение с помощью функции 09h прерывания INT 21h. Включить в сообщение управляющие коды ANSI-драйвера для управления положением курсора и цветом символов.

Определения cr, lf

...  
;Основные фрагменты программы

```
mov     AH,09h
mov     DX,offset msg
int     21h
```

;Функция вывода на экран  
;Адрес сообщения

Завершим программу

```
...
;Появ. данных
db
```

```
27,'[2J',27,'[12;10H',27,'[34;46m Начиная
27,'[6;10H Работать ',27,'[0m$'
```

**Задача 6.2.** Прямой вывод на экран (функция 06h, прерывание INT 21h). Вывести на экран строку символов с кодами от 1 до 31 (всего 31 символ). Проанализировать действие управляющих кодов (7, 8, 10, 13 и др.). Для большей наглядности предусмотреть после вывода каждого символа остановку программы (с помощью функций 7 или 8 прерывания INT 21h) до нажатия какой-либо клавиши.

...  
;Основные фрагменты программы

```
mov     CX,31
mov     DL,symb
mov     AH,06h
int     21h
inc     symb
```

;Всего столько кодов  
;Начнем с кода 1  
;Функция прямого вывода

Остановим программу с помощью функции 08h прерывания DOS (INT 21h)

```
...
loop    line
;Завершим программу
```

;На вывод следующего символа

```
...
;Появ. данных
symb db
```

1

;Код выводимого символа

**Задача 6.3.** Организация окна и вывод символов. Вывести на экран все 256 символов кодовой таблицы четырьмя строками по 64 символа.

...  
;Основные фрагменты программы

Очистим экран и зададим атрибуты символов с помощью окна

```
mov     AH,06h
mov     AL,0
mov     BH,31h
```

;Функция инициализации окна  
;Не прокручивать  
;Бирюзовый фон, синие символы

```

mov     CH,0
mov     CL,0
mov     DH,24
mov     DL,79
mov     INT,10h

; Будем выводить символы
; строками по 64 символа
; четыре строками по 64 символа
mov     CX,4
push    CX
1b72:   mov     CX,64
push    CX
1b71:   push    CX
; цикл
; Позиционируем курсор
mov     AH,02h
mov     BH,0
mov     DH,ROW
mov     DL,COL
int     10h

; Выведем очередной символ
mov     AH,0Ah

mov     AL,CHAR
mov     BH,0
mov     CX,1
mov     INT,10h

; Установим новые значения переменных и организуем циклы
inc     CHAR
inc     COL
; Следующий символ
; Следующий столбец
; Восстановим счетчик внутреннего цикла
mov     BH,11
; Снова с начала строки
; Вниз на две строки
add     ROW,2
; Восстановим счетчик внешнего цикла
mov     CX,1672
loop    1b72

; Завершим программу
...
; Поля данных
row     db     10
col     db     10
char    db     0

```

Задача 6.4. Вывод строки в режиме телетайпа. Вывести строку символов с указанием атрибутов.

```

; Основные фрагменты программы
mov     AX,DATA
mov     DS,AX
mov     ES,AX

; Выведем строку
mov     AH,13h
mov     AL,3
mov     BH,0
mov     CX,NUMSUM
mov     DH,24

; Функция вывода строки
; Формат строки (коды/атрибуты)
; Страница 0
; Число символов
; Строка

```

```

mov     DL,20
mov     BP,offset string
int     10h
; Столбец
; Адрес строки

; Завершим программу
...
; Поля данных
; Переменные коды и атрибуты
string  db     16,42h,16,42h,16,42h
        db     'S',28h,'T',28h,'R',28h,'I',28h,'N',28h,'G'
        db     28h,17,42h,17,42h,17,42h
stringlen=$-string
numsum=numsum/2

```

Задача 6.5. Организация окна. Задать окно размером в целый экран, изменив тем самым цвет экрана. Вывести в центр экрана окно меньшего размера другого цвета с текстом.

```

; Основные фрагменты программы
; Изменим цвет экрана
mov     AH,06h
mov     AL,0
mov     BH,30h
mov     CH,0
mov     CL,0
mov     DH,24
mov     DL,79
mov     INT,10h
; Функция инициализации окна
; Не прокручивать
; Бирюзовый фон
; Y левый верхний
; X левый верхний
; Y правый нижний
; X правый нижний

; Нарисуем пустое окно
mov     AH,06h
mov     AL,0
mov     BH,14h
mov     CH,10
mov     CL,30
mov     DH,14
mov     DL,49
mov     INT,10h
; Функция инициализации окна
; Не прокручивать
; Синий фон, красные символы
; Y левый верхний
; X левый верхний
; Y правый нижний
; X правый нижний

; Выведем в окно текст
mov     AX,seg text
mov     ES,AX
mov     AH,13h
mov     AL,0
mov     BH,0
mov     BL,14h
mov     CX,TEXTLEN
mov     DH,12
mov     DL,36
mov     BP,offset text
int     10h
; Подготовим сегментный
; регистр ES
; Функция вывода строки
; Режим
; Страница
; Атрибуты символов строки
; Длина строки
; Y
; X
; Адрес строки

; Завершим программу
...
; Поля данных
text     db     'Внимание!'
textlen=$-text

```

; Длина строки

**Задача 6.6.** Работа с видеостраницами. Сформировать на видеостраницах 0 и 1 два изображения, выводить их попеременно на экран. Для этого копированием подсоединить к тексту задачи 6.3 те же строки, изменив номер страницы и введя какое-либо отличие (размер окна, надпись, атрибут символов). Далее в цикле переключать страницы.

;Основные фрагменты программы  
;Сформируем изображение на странице 0

```
...
;Активизируем страницу 1
mov     AH,05h           ;Функция переключения страницы
mov     AL,1             ;Страница 1
int     10h
```

;Сформируем изображение на странице 1

;Будем переключать страницы  
toggle:

```
;Активизируем страницу 0
mov     AH,05h           ;Функция переключения страницы
mov     AL,0             ;Страница 0
int     10h
```

;Введем небольшую задержку

```
...
;Активизируем страницу 1
mov     AH,05h           ;Функция переключения страницы
mov     AL,1             ;Страница 1
int     10h
```

;Введем небольшую задержку

;Анализ буфера клавиатуры функцией 06h прерывания DOS (INT 21h)  
и завершение программы по нажатию клавиши

```
mov     AH,06h           ;Функция ввода без ожидания
mov     DL,0FFh          ;Ввод
int     21h
jnz     outpr            ;Z=1, символ есть, на выход
jmp     toggle           ;Z=0, символа нет
```

outpr:

;Перед завершением программы восстановим в качестве текущей  
;видеостраницу 0

```
mov     AH,05h           ;Функция переключения страницы
mov     AL,0             ;Страница 0
int     10h
```

;Завершим программу

```
...
;Поля данных
text    db      'Внимание!'
textlen=$-text           ;Длина строки
```

**Задача 6.7.** Прокрутка окна. Ввести в пример 6.5 прокрутку текста в цикле вверх и вниз на одну строку.

;Основные фрагменты программы

;Сформируем изображение на странице 0

```
...
;Прокрутим окно вверх
updown: mov     AH,06h
         mov     AL,1
         mov     BH,14h
         mov     CH,11
         mov     CL,31
         mov     DH,13
         mov     DL,48
         int     10h
         ;Функция прокрутки вверх
         ;Прокрутить на 1 строку
         ;Те же атрибуты
         ;Y левый верхний
         ;X левый верхний
         ;Y правый нижний
         ;X правый нижний
```

;Введем небольшую задержку

```
...
;Прокрутим окно вниз
mov     AH,07h
mov     AL,1
mov     BH,14h
mov     CH,11
mov     CL,31
mov     DH,13
mov     DL,48
int     10h
;Функция прокрутки вниз
;Прокрутить на 1 строку
;Те же атрибуты
;Y левый верхний
;X левый верхний
;Y правый нижний
;X правый нижний
```

;Введем небольшую задержку

;Анализ буфера клавиатуры функцией 06h прерывания DOS (INT 21h)  
и завершение программы по нажатию клавиши

```
...
jmp     updown
;Завершим программу
...
```

**Задача 6.8.** Переключение мерцание - яркость. Заполнить участок экрана надписями разного цвета. В цикле переключать значение бита мерцание - яркость. Цвета и расположение строк подобраны в этой задаче таким образом, что наглядно демонстрируется влияние установки бита мерцание - яркость на характеристики изображения.

;Основные фрагменты программы

;Инициализируем оба сегментных регистра DS и ES

```
...
;Очистим экран
mov     AH,40h
mov     BX,1
mov     CX,homeLen
mov     DX,offset home
int     21h
```

;Выведем строку line1 с позиции 12,10

```
mov     AH,13h           ;Функция вывода строки
mov     AL,0             ;Атрибут в BL
mov     BH,0             ;Страница 0
mov     BL,41h           ;Синий по красному
mov     CX,11len         ;Длина строки
```



```

mov     DH,12             ;Координата Y
mov     DX,10             ;Координата X
mov     BP,offset line1   ;Адрес строки
int     10h
;Выведем строку line2 с позиции 14,10
mov     AH,13h            ;Функция вывода строки
mov     AL,0              ;Атрибут в BL
mov     BH,0              ;Страница 0
mov     BL,0C1h           ;Синий по розовому
mov     CX,line           ;Длина строки
mov     DH,14             ;Координата Y
mov     DL,10             ;Координата X
mov     BP,offset line2   ;Адрес строки
int     10h
;Выведем строку line3 с позиции 12,35
...
;Выведем строку line4 с позиции 14,35
...
toggle:
;Включим мерцание
mov     AH,10h            ;Группа подфункций
mov     AL,03h            ;Подфункция мерцание - яркость
mov     BL,1              ;Мерцание
int     10h
;Введем задержку, заметно большую периода мерцания
call    delay
;Включим яркость
mov     AH,10h            ;Группа подфункций
mov     AL,03h            ;Подфункция мерцание - яркость
mov     BL,0              ;Яркость
int     10h
;Введем задержку, заметно большую периода мерцания
call    delay
;Анализ буфера клавиатуры функцией 06h прерывания DOS (INT 21h)
;и завершение программы по нажатию клавиши
...
jmp     toggle            ;Z=0, символа нет
;Завершим программу
...
delay    proc
;Подпрограмма задержки
mov     CX,200
cxloop1: mov     BX,0
delay1:  inc     BX
jne     delay1
loop    cxloop1
delay    endp
;Поля данных
home     db      27,'[2J]' ;Очистка экрана
home len equ     $-home
line1    db      ' Синий по красному '
line2    db      ' Синий по розовому '
line3    db      ' Желтый по синему '

```

```

line4    db      ' Желтый по голубому '
line len equ $-line4
;Все строки одной длины

```

### Задача 6.9. Вывод символов в видеобuffer.

```

;Основные фрагменты программы
;Очистим экран с помощью ANSI-драйвера
mov     AH,09h
lea     DX,mes
int     21h
;Настроим сегментный регистр ES на страницу 0 видеобufferа
mov     AX,0B800h
mov     ES,AX
;Выведем символы
mov     AH,14h
mov     AL,'*'            ;Красный по синему
mov     ES:0,AX           ;Код звездочки
mov     ES:2,AX           ;Первая позиция на экране
mov     ES:4,AX           ;Следующая позиция
mov     AH,75h           ;Следующая позиция
mov     AL,1              ;Фиолетовый по белому
mov     BX,(160*24)+(77*2) ;Код позиции
mov     ES:[BX],AX        ;3 позиции от конца буфера
mov     ES:2[BX],AX
mov     ES:4[BX],AX
;Остановим программу функцией 8h DOS в ожидания нажатия на любую клавишу
;Завершим программу
...
;Поля данных
mes     db      27,'[2J]'

```

### Задача 6.10. Вывод строки в видеобuffer терминала.

```

;Основные фрагменты программы
;Очистим экран
...
;Настроим сегментный регистр ES на страницу 0 видеобufferа
...
;Выведем всю строку
mov     SI,offset text    ;Адрес строки
mov     DI,12*80*2+33*2   ;В центр экрана
mov     CX,textlen        ;Длина строки
movsb
;Анализ буфера клавиатуры функцией 06h прерывания DOS (INT 21h)
;и завершение программы по нажатию клавиши
...
;Завершим программу
...
;Поля данных
text     db      5 dup (16,74h) ;Код ASCII 16, красный по белому
          db      'T',7Ch,'e',7Ch,'s',7Ch,'t',7Ch ;Розовый по белому
          db      5 dup (17,74h) ;Код ASCII 17, красный по белому
textlen equ $-text

```

### Задача 6.11. Вывод строки на страницу 7 видеобuffers.

```

;Основные фрагменты программы
;Настроим сегментный регистр ES на страницу 7 видеобuffers
mov     AX,0BF00h
mov     ES,AX

;Очищать страницу 7 нет необходимости
;Выведем всю строку
mov     SI,offset text      ;Адрес строки
mov     DI,12*80*2+33*2    ;8 центр экрана
mov     CX,textlen         ;Длина строки

mov     movsb

jmp     ;Введем задержку
switch:
;Переключим страницу
mov     AH,05h             ;Функция выбора страницы
mov     AL,07h             ;Страница 7
int     10h

;Еще раз введем задержку
...
;Переключим страницу
mov     AH,05h             ;Функция выбора страницы
mov     AL,0               ;Страница 0
int     10h

;Анализ буфера клавиатуры функцией 06h прерывания DOS (INT 21h)
;и завершение программы по нажатию клавиши
...
jmp     switch
;Завершим программу
...

;Поля данных
text    db      5 dup (16h,04h) ;Код ASCII 16h, красный по черному
        db      't',4Eh,'e',4Eh,'s',4Eh,'t',4Eh ;Желтый по красному
        db      5 dup (16h,04h) ;Код ASCII 16h, красный по черному
textlen=$-text

```

### 6.7. Системные средства управления шрифтами

Для отображения символов на экране где-то в компьютере должно быть записано, в какую именно фигуру на экране преобразуется каждый код ASCII. Эта информация (кодированные страницы) составляет содержание файла EGA.CPI. Например, в русифицированной версии DOS 4.01 файл EGA.CPI включает 6 кодовых страниц с условными номерами 437 (США), 850 (США и все европейские страны), 860 (Португалия), 863 (Канада), 865 (Дания и Норвегия) и 866 (Россия).

Каждая страница состоит из трех таблиц, в которых описаны изображения всех 256 символов ASCII для шрифтов с размером каждого символа 8x16, 8x14 и 8x8 точек. Каждый символ требует для своего описания соответственно 16, 14 или 8 байтов. На рис.6.3 даны для примера некоторые извлечения из

файла EGA.CPI, показывающие принцип описания формы символа (для размеров символа 8x14 и 8x8 точек). Слева от изображения символов приведены последовательности байтов в кодовой таблице.

Символы размером 8x14 точек

Код символа 01h		Код символа 51h	
00		00	
00		00	
00		7C	
7E		C6	
81		C6	
A5		C6	
81		D6	
8D		DE	
99		7C	
81		0C	
7E		0E	
00		00	
00		00	

Символы размером 8x8 точек

Код символа 01h		Код символа 51h	
7E		3C	
81		66	
A5		66	
81		66	
8D		6E	
99		3C	
81		0E	
7E		00	

Рис. 6.3. Извлечения из кодовой таблицы файла EGA.CPI.

Для того, чтобы символы, описанные в той или иной таблице, могли отображаться на экране, сама таблица должна быть загружена в память знакогенератора видеоадаптера. На уровне команд пользователя это можно сделать с помощью команд MODE; существуют и специальные программы (напр., FONT8X8.COM или VGA866.COM), которые позволяют загрузить имеющиеся в них кодовые страницы без помощи программы MODE. Это позволяет сэкономить память за счет того, что загружаются только шрифты нужного размера.

На уровне программ загрузка кодовых таблиц (для адаптера EGA) осуществляется с помощью функций видеодрайвера BIOS (Int 10h):

функция 11h, подфункция 10h - загрузка шрифта пользователя с перепрограммированием контроллера на новый размер символов;

функция 11h, подфункция 11h - загрузка шрифта 8x14 с перепрограммированием контроллера на новый размер символов;  
 функция 11h, подфункция 12h - загрузка шрифта 8x8 с перепрограммированием контроллера на новый размер символов;  
 функция 11h, подфункция 03h - установка номера блока знакогенератора.

Память знакогенератора EGA включает четыре блока с номерами 0...3, в которых могут одновременно храниться четыре полных набора символов (по 256 символов в наборе). Обычно используется только один блок 0, в который при загрузке машины заносится стандартная таблица символов с размером 8x14 точек. С помощью подфункции 12h функции 11h в блок 0 можно загрузить таблицу символов размером 8x8 точек и работать далее в таком "уплотненном" режиме экрана. Эту операцию и выполняет опция EGA Lines программы Norton Commander. Подфункция 11h функции 11h возвращает экран в прежний режим.

Подфункция 10h функции 10h служит для загрузки в блок памяти знакогенератора таблицы шрифтов пользователя. Подфункция позволяет задать высоту символа (в регистре BH), адрес таблицы (в регистрах ES:BP), число определяемых символов (в регистре CX), а также код, присваиваемый первому символу таблицы (в регистре DX). Таким образом, имеется возможность переопределить не все символы стандартной таблицы, а только часть, и при этом расположить новые символы в произвольном месте таблицы. Если, например, требуется задать 32 новых символа, для них можно выделить коды 128 - 159, отвечающие в стандартной таблице прописным русским буквам. В этом случае следует задать CX=32 и DX=128.

Кодовая таблица, адрес которой заносится в регистры ES:BP, представляет собой просто перечень кодов, определяющих последовательные строки символов (см. приведенные выше извлечения из файла EGA.CPI). Так, например, строки

```
usertab db 0FFh,81h,81h,81h,81h,81h,81h,0FFh
db 00,7Eh,7Eh,7Eh,7Eh,7Eh,7Eh,00
db 00,00,3Ch,3Ch,3Ch,3Ch,00,00
db 00,00,00,18h,18h,00,00,00
```

определяют изображения четырех прямоугольников последовательно уменьшающегося размера (шрифт 8x8).

Таблицы нестандартных символов используются в игровых и обучающих программах для построения несложных изображений. Достоинства такой методики по сравнению с графическим режимом - простота программирования и более высокая скорость вывода на экран.

Наличие в знакогенераторе адаптера EGA нескольких блоков памяти позволяет хранить в памяти до четырех кодовых таблиц, из которых в любой момент активными могут быть две (в общей сложности до 512 символов). Кодовые таблицы загружаются в память знакогенератора с помощью таблицы загрузки функции 11h. Номер блока, в который загружается данная таблица, указывается в регистре BL. Если контроллер уже настроен на требуемый размер шрифта, и перепрограммировать его не требуется, вместо подфункции 10h используется подфункция 00h, вместо 11h - 01h и вместо 12h - 02h.

Подфункция 03h функции 11h позволяет задать те два из четырех блоков памяти знакогенератора, которые находятся в активном состоянии и могут использоваться программой. Номер этих блоков указывается в битах 0...1 и 2...3 регистра BL. Выбор блока осуществляется (для каждого символа в отдельности) битом 3 атрибута символа, который в этом случае уже не управляет яркостью символа. Если бит 3 атрибута символа равен 0, выбирается блок, указанный при вызове подфункции 03h в битах 0...1 регистра BL. Если бит 3 атрибута равен 1, выбирается блок, указанный в битах 2...3 регистра BL. Для того, чтобы вернуться к режиму 16 цветов и, соответственно, к набору из 256 символов, следует при вызове подфункции 03h в обоих полях регистра BL указать один и тот же номер блока.

## 6.8. Задачи по программной смене шрифтов

**Задача 6.12.** Перепрограммирование видеоконтроллера на шрифт 8x8. Вывести на экран служебное сообщение с помощью любой подходящей функции DOS. Организовать смену шрифта на 8x8 точек. Вывести на экран строку текста в режиме 8x8. Выйти из программы в режиме 8x8 и восстановить режим 8x14 средствами программы Norton Commander (опция EGA Lines).

Основные фрагменты программы

;Выведем на экран строку текста mes1 какой-либо функцией DOS (40h, 09h)

...  
;Остановим программу

```
mov AH,08h
int 21h
```

;Ввод без эха с ожиданием

;Сменим шрифт на 8x8

```
mov AH,11h
mov AL,12h
mov BL,0
int 10h
```

;Функция работы со шрифтами

;Подфункция загрузки шрифта 8x8

;Блок знакогенератора

;Остановим программу

...  
;Выведем на экран вторую строку текста mes2

...



```
;Остановки программы
```

```
;Заверши программу
```

```
mes1 db 'Тестовая строка, выводимая в режиме 8x14',10,13
mes2 db 'Вторая строка, выводимая в режиме 8x8',10,13
```

**Задача 6.13.** Установка шрифта пользователя. Создать небольшую таблицу с определением нестандартных символов. Загрузить ее в знакогенератор. С помощью подходящей функции DOS вывести нестандартные символы на экран. Для задания местоположения и цвета символов можно воспользоваться Esc-последовательностями. При завершении программы не восстанавливать стандартную таблицу символов. Выйдя из программы, вывести на экран пересопределенные символы какими-либо средствами DOS (командой ECHO либо просто вводом в командной строке). Перейти в обычный режим средствами Norton Commander и проанализировать результат.

```
;Основные фрагменты программы
```

```
;Настроим регистры ES:BP для загрузки шрифта пользователя
```

```
mov AX,seg newchar
mov ES,AX
mov BP,offset newchar
```

```
;Сменим шрифт на 8x8
```

```
mov AH,11h
mov AL,12h
mov BL,0
int 10h
```

```
;Функция работы со шрифтами
```

```
;Подфункция загрузки шрифта 8x8
```

```
;Блок 0 знакогенератора
```

```
;Загрузим шрифт пользователя
```

```
mov AH,11h
mov AL,10h
mov BH,8
mov BL,0
mov CX,4
mov DX,128
int 10h
```

```
;Функция работы со шрифтами
```

```
;Подфункция загрузки нашего шрифта
```

```
;Высота 8 точек
```

```
;Блок 0 знакогенератора
```

```
;Число символов в таблице
```

```
;Код первого символа
```

```
;Выведем строку string функцией 09h прерывания Int 21h
```

```
;Заверши программу
```

```
;Поля данных
```

```
;Таблица нового шрифта
```

```
;Первый символ (код ASCII 128, вместо русской буквы А)
```

```
newchar db 11111111B
db 10000001B
db 10000001B
db 10000001B
db 10000001B
db 10000001B
db 10000001B
db 11111111B
```

```
;Второй символ (код ASCII 129, вместо русской буквы Б)
```

```
db 00000000B
db 01111110B
db 01000010B
db 01000010B
db 01000010B
db 01000010B
db 01111110B
db 00000000B
```

```
;Третий символ (код ASCII 130, вместо русской буквы В)
```

```
db 00000000B
db 00000000B
db 00111100B
db 00100100B
db 00100100B
db 00111100B
db 00000000B
db 00000000B
```

```
;Четвертый символ (код ASCII 131, вместо русской буквы Г)
```

```
db 00000000B
db 00000000B
db 00000000B
db 00011000B
db 00011000B
db 00000000B
db 00000000B
db 00000000B
```

Ниже описана тестовая строка для вывода на экран. В строке выполняется очистка экрана, позиционирование курсора, установка атрибута символов (красный по черному), вывод самих символов, возврат исходного цвета экрана, контрольный вывод символов при прямом задании их кодов ASCII. Для функции 09h строка завершается знаком \$

```
string db 27,'[2J',27,'[25;1H',27,'[31;40m'
db 'А И Б СИДЕЛИ НА ТРУБЕ. а и б сидели на трубе',27,'[0m'
db 10,10,13,128,129,130,131,132,133,134,135,10,13,'$'
```

## 7. Вывод графической информации на экран терминала

### 7.1. Графические возможности видеодрайвера BIOS

Как уже отмечалось выше, графический адаптер EGA обеспечивает хранение и отображение двух графических страниц с разрешением 640x350 цветных точек (пикселей). Адаптер поддерживает 64 цвета, хотя в каждый момент времени изображение на экране может содержать только 16 цветов. Этот набор из 16 цветов, выводимых на экран (цветовая палитра), задается программно и может легко изменяться. При загрузке машины устанавливается стандартная палитра (см. табл. 6.1)

Фоновый цвет всего экрана может принимать любое из 16 текущих значений палитры, причем на обеих страницах фона изменяется одновременно. Собственно говоря, фона как такового нет. Под фоном понимается совокупность всех еще не покрашенных точек. Однако сменить цвет фона можно, что, естественно, не затронет выведенного на экран изображения. Ниже этот вопрос будет рассмотрен подробнее.

Помимо произвольных изображений, рисуемых пиксел за пикселом, на экране в графическом режиме можно отображать и текстовую информацию, используя любые функции DOS и BIOS, выводящие на экран символьные строки, например, INT 21h с функциями 40h или 09h, INT 10h с функциями 09h или 0Ah и т.д. Для позиционирования строк на экране можно использовать функцию 02h драйвера BIOS (прерывание INT 10h), устанавливающую местоположение курсора в текстовых координатах (номер строки от 0 до 24 и номер столбца от 0 до 79), хотя сам курсор в графическом режиме на экран не выводится. Таким образом, символы, выводимые на экран в графическом режиме, попадают в точности в те же места, что и в текстовом; текстовые строки нельзя выводить с указанием графических координат. Нельзя также изменять их направление (т.е. выводить на экран вертикально или наклонно). Конфигурация символов берется из кодовых таблиц; при необходимости отображения символов другого размера или начертания, следует подготовить соответствующие кодовые таблицы или рисовать

символы вручную. Многие прикладные программы широкого назначения (графический пакет системы Turbo Pascal, графический редактор PaintBrush) имеют такие средства. При программировании графического изображения можно использовать следующие функции видеодрайвера BIOS (прерывание INT 10h):

- 00h - установка видеорежима;
- 05h - установка видеостраницы;
- 0Ch - вывод пиксела;
- 0Fh - получение видеорежима;
- 10h, подфункция 00h - установка цветового регистра;
- 10h, подфункция 01h - установка цвета края экрана;
- 10h, подфункция 02h - установка цветовых регистров палитры;
- 11h, подфункция 21h - загрузка таблицы шрифтов пользователя в графическом режиме.

Кроме перечисленных, имеется еще ряд функций более ограниченного использования (например, чтение пиксела в заданной графической позиции).

Функция 00h позволяет переключать режимы видеосистемы. Для адаптера EGA с видеобуфером объемом 256 Кбайт текстовый режим имеет код 03, графический - 10h.

Программа, активно использующая оба режима, может с помощью функции 0Fh определить текущий видеорежим (а также и текущую видеостраницу) и, в зависимости от полученного результата, оценить необходимость переключения режима или страницы.

Функция 05h переключает видеостраницу. Номер страницы заносится в регистр AL. В режиме 10h (с адаптером EGA) можно использовать страницы 0 и 1. Изображение, выведенное на ту или иную страницу, не пропадает при переключении активной страницы, поскольку в видеоадаптере для каждой страницы имеются отдельные блоки памяти.

Для рисования изображения предусмотрена единственная функция 0Ch - вывод пиксела (напомним, что в DOS вообще нет функций, поддерживающих графику). В регистр AL засылается значение цвета пиксела, в регистр BH - номер страницы (не обязательно активной), в регистры CX и DX - X- и Y-координаты пиксела, соответственно. X-координата изменяется в пределах от 0 до 639, Y-координата - от 0 до 349. Точка с координатами (0,0) соответствует левому верхнему углу экрана.

Цвета пикселей на экране, как и цвет фона, задаются соответствующими цветовыми регистрами, определяющими цветовую палитру видеосистемы. Всего в адаптере EGA имеются 17 цветовых регистров, каждый из которых может содержать коды 64

цветов от 0 до 63. Регистры с 0 по 15 определяют возможные цвета пикселей, а регистр 16 - цвет рамки вокруг рабочей части экрана (выбег развертки). Эта рамка обычно не видна, так как ей присваивают цвет, совпадающий с цветом фона.

Код цвета, указываемый в программе (например, в регистре AL для функции 0Ch), определяет собственно не цвет, а номер цветового регистра, используемого системой при выводе на экран данного пиксела. Цвет же пиксела определяется числом, хранящимся в регистре.

Каждый цветовой регистр содержит 6 значащих разрядов, которые определяют интенсивность красного, зеленого и синего цветов, дающих при смешивании требуемый цвет. Разряды 0, 1 и 2 закреплены за цветами красный, зеленый и синий с интенсивностью 2/3 максимальной, разряды 3, 4 и 5 - за теми же цветами с интенсивностью 1/3:

Номера разрядов	5	4	3	2	1	0
Веса разрядов	32	16	8	4	2	1
Цвет	к	з	с	к	з	с

Таким образом, число 1(С) - записанное в регистр, определяет синий цвет, число 2(З) - зеленый, число 3(З+С) - сине-зеленый (бирюзовый), число 9(с+С) - ярко-синий, число 7(К+С+З) - белый, число 63(к+с+з+К+С+З) - ярко-белый. Регистр, содержащий 0, определяет черный цвет.

Обычно при инициализации в регистры заносится следующий ряд чисел:

Регистр	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Содержимое	0	1	2	3	4	5	20	7	56	57	58	59	60	61	62	63	0

Эта последовательность образует стандартную цветовую палитру. Заметим, что в исходном состоянии и фон (регистр 0), и выбег развертки (регистр 16) оказываются черного цвета, так как в них записан 0.

При указании в программе кода цвета б (а в действительности - номера регистра б) на экран выводится коричневое изображение, так как число 20, записанное в регистр б, определяет коричневый (красный с примесью зеленого) цвет. Если, однако, в цветовой регистр б записать, например, 40, то все обращения к коду цвета б приведут к выводу на экран ярко-красного изображения; запись в регистр б числа 12 сменит цвет всех коричневых точек на экране на малиновый (красный с примесью синего).

Таким образом, изменение содержимого цветовых регистров позволяет практически мгновенно менять цвета изображения на экране, настраивая, например, изображение на холодную сине-зеленую гамму или на теплую красно-желтую. Перенастройка

стандартной палитры иногда используется при выводе на экран географических карт или псевдообъемных изображений в системах автоматизации проектирования и других научно-технических приложениях. Возможность настройки цветовой палитры предусмотрена, в частности, в пакете мортонских утилит.

Цветовой регистр 0 определяет цвет фона. Цвет фона - это тот цвет, которым заполняется экран при инициализации графического режима, когда во все ячейки видеобuffers записывается 0. Таким образом, фон - это та часть экрана, в которой начальный код 0 еще не заменен каким-либо кодом цветного изображения. Указание в программе кода цвета 0 при выводе какого-то элемента изображения приводит к использованию системы регистра фона, в результате чего этот элемент оказывается невидимым. Для того, чтобы вывести при цветном фоне черное изображение, следует в какой-либо цветовой регистр записать 0, и при выводе изображения указывать номер этого регистра.

В то время, как переключение цветов осуществляется практически мгновенно, вывод пикселей средствами BIOS требует немалого времени, до нескольких сотен микросекунд на пиксел при медленном процессоре. Зарисовывание всего экрана может занять 1 - 2 минуты. Поэтому графические пакеты обычно работают не с функциями BIOS, а непосредственно с графическим адаптером.

## 7.2. Задачи по программированию графического режима

**Задача 7.1.** Вывод на экран графического изображения. Установить графический режим и вывести в середину экрана цветной прямоугольник. После отладки программы определить скорость вывода точек в графическом режиме.

Основные фрагменты программы

Программа draw вывода прямоугольника

Входные параметры - x, y, xsize, ysize, forcolor

Программа использует xcnt и ycnt, модифицируя эти переменные

proc			
draw	mov	AX,y	;Инициализация
	mov	ycnt,AX	;ycnt
	mov	CX,ysize	;Счетчик по Y
	push	CX	;Сохраним его в стеке
	mov	AX,x	;Инициализация
	mov	xcnt,AX	;xcnt
	mov	CX,xsize	;Счетчик по X
	push	CX	;Сохраним его в стеке
	mov	AH,0Ch	;Функция вывода пиксела
	mov	AL,forcolor	;Цвет пиксела
	mov	BH,0	;Страница 0
	mov	CX,xcnt	;Текущий графический столбец



```

mov     DX,ycrnt      ;Текущая графическая строка
int     10h           ;Сместимся вправо
inc     xcrnt         ;Восстановим счетчик по X
pop     CX            ;Сместимся вниз
loop    loopx         ;Восстановим счетчик по Y
inc     ycrnt
pop     CX
loop    loopy
ret
draw    endp
;Начало основной программы

;Установим графический режим
mov     AH,0
mov     AL,10h
int     10h
;Нарисуем прямоугольник пикселами
call    draw
;Завершим программу

;Поля данных
x       dw     200
y       dw     100
xcrnt   dw     0
ycrnt   dw     0
xsize   dw     200
ysize   dw     100
forcolor db    14

```

;X-координата начала  
 ;Y-координата начала  
 ;Текущая X-координата  
 ;Текущая Y-координата  
 ;Размер прямоугольника по X  
 ;Размер прямоугольника по Y  
 ;Желтый цвет

**Задача 7.2.** Изучение цветовой палитры. Вывести на экран ряд из 16 прямоугольников всех возможных цветов (прямоугольник, цвет которого определяется цветовым регистром 0, будет невидим). Под прямоугольниками вплотную к ним нарисовать длинную полосу любого цвета, т.е. с указанием номера любого цветового регистра (от 1 до 15). Далее в цикле заносить в этот регистр числа 0, 1, 2, ... 63, изменяя тем самым цвет полосы. Для удобства наблюдения следует, во-первых, включить в цикл остановку программы в ожидании нажатия на клавишу и, во-вторых, при каждой смене цвета выводить на экран содержимое модифицируемого цветового регистра. Отладив программу, наблюдать смену цветов, определяемых содержимым модифицируемого цветового регистра, при желании анализируя состав каждого цвета.

```

;Основные фрагменты программы
draw    proc
;Подпрограмма вывода прямоугольника

draw    endp
;Начало основной программы

```

Установим графический режим

```

;Нарисуем прямоугольнички
mov     CX,16
push    CX
call    draw
inc     forcolor
add     x,40
pop     CX
loop    rect
;Нарисуем длинную полосу. Для этого изменим значения
;параметров, используемых подпрограммой draw
mov     x,0
mov     y,50
mov     xsize,635
mov     ysize,40
mov     forcolor,4
;Нарисуем полосу
call    draw
;Остановим программу
mov     AH,0ch
mov     AL,0bh
int     21h
;Сменим в цикле цвет в регистре 4 от 0 до 63
mov     CX,64
push    CX
mov     AH,10h
mov     AL,0
mov     BL,4
mov     BH,palette
int     10h
;Выведем номер цвета. Для этого:
;1. Установим где-нибудь текстовый курсор для вывода сообщения mes
mov     AH,02h
mov     BH,0
mov     DH,10
mov     DL,10
int     10h
;2. Преобразуем номер цвета в адрес его символического
;представления в таблице table
mov     AL,palette
xor     AH,AH
add     AX,AX
mov     SI,AX
mov     BX,offset table
mov     AX,[BX][SI]
mov     number,AX
;3. Выведем номер на экран
mov     AH,40h
mov     BX,1
mov     CX,meslen
mov     DX,offset mes
int     21h

```

;Счетчик цветов  
 ;Сохраним его в стеке  
 ;Нарисуем прямоугольнички  
 ;Следующий цвет  
 ;Следующее начало  
 ;Восстановим счетчик

;Рисуем регистром 4  
 ;Функция ожидания ввода  
 ;Фильтрованный ввод без эха

;Счетчик цветов  
 ;Сохраним его  
 ;Функция 10h  
 ;Подфункция установки цвета  
 ;Регистр 4, обычно - красный  
 ;Номер цвета

;Функция вывода  
 ;Стандартный дескриптор вывода  
 ;Длина сообщения  
 ;Адрес строки

```

;Изменим цвет
inc palette
pop cx
;Следующий цвет
;Восстановим счетчик цветов
;Остановим программу
...
loop back
;Завершим программу

;Поля данных
x dw 0
y dw 20
xcnt dw 0
ycnt dw 0
xsize dw 35
ysize dw 40
foreground db 0
palette db 0
mes db 0
number dw 0
mes len=$-mes
;Таблица возможных номеров цветов в символической форме
;Обратите внимание на то, что все числа занимают по два байта
table
db 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
db 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
db 20, 21, 22, 23, 24, 25, 26, 27, 28, 29
db 30, 31, 32, 33, 34, 35, 36, 37, 38, 39
db 40, 41, 42, 43, 44, 45, 46, 47, 48, 49
db 50, 51, 52, 53, 54, 55, 56, 57, 58, 59
db 60, 61, 62, 63

```

**Задача 7.3.** Изменение цвета фона. Вывести на экран ряд из 16 прямоугольников всех возможных цветов (см. пример 7.2). В цикле (с остановкой) заносить в регистр 0 числа 0, 1, 2, ..., 63, изменяя тем самым цвет фона. Сравнить цвета фона с цветами изображения прямоугольников, нарисованных пикселями. Обратит внимание на то, что самый левый прямоугольник, цвет которого задан регистром 0, при любом цвете фона остается невидимым.

```

;Основные фрагменты программы
;Подпрограмма вывода прямоугольника
...

```

```

;Начало основной программы
...

```

```

;Установим графический режим
...

```

```

;Нарисуем прямоугольники
...

```

```

;Остановим программу
...

```

```

;Сменим в цикле цвет фона от 0 до 63

```

```

mov cx, 64 ;Счетчик цветов
back: push cx ;Сохраним его

```

```

mov ah, 10h ;Функция 10h
mov al, 0 ;Подфункция установки цвета
mov bl, 0 ;Регистр 0 - фона
mov bh, palette ;Номер цвета
int 10h
;Выведен номер цвета
...
;Завершим программу
...

```

**Задача 7.4.** Настройка цветовой гаммы. Вывести на экран ряд из 16 прямоугольников всех возможных цветов (см. задачу 7.2). С помощью функции 10h с подфункцией 02h задать последовательно четыре разных палитры, перебрав в итоге все 64 цвета. Одна палитра может составлять, например, условно синюю гамму, другая - красную, третья - зеленую и четвертая - желтую. Цвет фона, как и цвет края экрана в каждой палитре, лучше сделать черным (в этом случае, правда, удастся реализовать лишь 60 цветов).

```

;Подпрограмма вывода прямоугольника
...

```

```

;Начало основной программы
...

```

```

;Установим графический режим
...

```

```

;Нарисуем прямоугольники цветами стандартной палитры
...

```

```

;Остановим программу
...

```

```

;Подготовим регистр ES
mov ax, seg plt_blue
mov es, ax

```

```

;Установим цветовую палитру синих тонов
mov ah, 10h ;Функция 10h
mov al, 2 ;Подфункция установки палитры
mov dx, offset plt_blue ;Адрес палитры
int 10h

```

```

;Остановим программу
...

```

```

;Установим цветовую палитру красных тонов
...

```

```

;Остановим программу
...

```

```

;Установим цветовую палитру зеленых тонов
...

```

```

;Остановим программу
...

```

```

;Установим цветовую палитру желтых тонов
...

```

```

;Остановим программу
...

```

```
;Завершим программу
```

```
;Поля данных
```

```
;Синяя гамма
plt_blue db 00,01,03,08,09,13,17,25
          db 29,31,33,41,43,49,57,59,00
```

```
;Красная гамма
plt_red db 00,04,05,12,20,28,32,36
         db 37,40,44,45,52,53,60,61,00
```

```
;Зеленая гамма
plt_grn db 00,02,07,10,11,16,18,19
         db 26,27,34,35,42,48,51,58,00
```

```
;Желтая гамма
plt_yel db 00,14,22,06,23,24,30,38
         db 39,46,47,54,55,56,62,63,00
```

Задача 7.5. Вывод текста в графическом режиме. Установить графический режим и цвет фона. Вывести на экран текстовые строки с помощью различных функций.

```
;Основные фрагменты программы
;Установим графический режим
```

```
;Установим цвет фона (например, синий)
```

```
mov AH,10h
mov AL,0
mov BL,0
mov BH,1
int 10h
```

```
;Функция 10h
```

```
;Установка цветового регистра
```

```
;Регистр фона
```

```
;Цвет фона
```

```
;Установим текстовый курсор
```

```
mov AH,02h
mov BH,0
mov DH,5
mov DL,20
int 10h
```

```
;Функция установки курсора
```

```
;Страница 0
```

```
;Номер текстовой строки
```

```
;Номер текстового столбца
```

```
Выведем строку string через дескриптор
```

```
;Установим текстовый курсор на две строки ниже
```

```
;Выведем символ средствами BIOS
```

```
mov AH,09h
mov AL,23
mov BH,0
mov BL,4
mov CX,25
int 10h
```

```
;Функция вывода символа
```

```
;Произвольный код символа
```

```
;Страница 0
```

```
;Цвет красный
```

```
;Длина строки
```

```
;Остановим программу
```

```
;Завершим программу
```

```
;Поля данных
```

```
string db 'Строка, выводимая средствами DOS'
```

```
stringlen equ $-string
```

Задача 7.6. Установка шрифта пользователя в графическом режиме средствами BIOS. Предусмотреть в полях данных программы две таблицы с определением нестандартных символов. Перейдя в графический режим, вывести на экран с помощью какой-либо функции строку, содержащую коды переопределенных символов. Наблюдать их отображение на экране в исходном состоянии компьютера, когда действует кодовая страница, определяемая конфигурацией системы. После этого заменить системную кодовую страницу на страницу пользователя. Для этого адрес таблицы с определением символов загрузить соответствующей функцией BIOS в вектор 43h. Вывести ту же строку.

Завершив программу, организовать вывод на экран переопределенных символов какими-либо средствами DOS (командой ECHO из заранее подготовленного командного файла либо просто в командной строке). Для выполнения последнего задания следует перед запуском программы выгрузить оболочку DOS Norton Commander, которая автоматически восстанавливает исходную кодовую страницу.

```
;Основные фрагменты программы
;Установим графический режим (10h)
```

```
;Установим цвет фона (например, синий)
```

```
mov AH,10h
mov AL,0
mov BL,0
mov BH,1
int 10h
```

```
;Функция 10h
```

```
;Установка цветового регистра
```

```
;Регистр фона
```

```
;Цвет фона
```

```
;Установим цвет символов (например, желтые)
```

```
mov AH,10h
mov AL,0
mov BL,07h
mov BH,14
int 10h
```

```
;Функция 10h
```

```
;Установка цветового регистра
```

```
;Регистр белого цвета
```

```
;Новый цвет (желтый)
```

```
;Установим текстовый курсор куда-нибудь в середину экрана
```

```
;Выведем строку str функцией 09h прерывания 21h
```

```
;Загрузим шрифт пользователя
```

```
mov AH,11h
mov AL,21h
mov BL,2
mov CX,14
lea BP,newchar1
push DS
pop ES
int 10h
```

```
;Функция работы со шрифтами
```

```
;Загрузка вектора 43h
```

```
;25 строк на экран
```

```
;14 линий развертки на символ
```

```
;Адрес таблицы описания 256
```

```
;символов (обе половины)
```

```
;ES:BP->нашу таблицу
```

```
;Установим текстовый курсор на другое место экрана
```





старший бит байта отображается слева, а младший - справа (рис. 7.2). Таким образом, пересылка по любому адресу видеобуфера одного байта данных приводит к формированию восьми соседних точек изображения, лежащих на горизонтальной прямой. При этом установленные в 1 биты пересылаемого байта соответствуют тем точкам, которые надо покрасить, а нулевые биты - тем точкам, которые надо либо погасить, либо оставить без изменения.



Рис. 7.2. Байт изображения

Операции над содержимым битовых плоскостей выполняются непосредственно в видеопамати, а в промежуточном буфере, образованном четырьмя восьмиразрядными регистрами-фиксаторами (рис. 7.3). Каждый регистр соответствует одной битовой плоскости. Чтение байта из видеобуфера приводит к переносу из битовых плоскостей в регистры-фиксаторы всех четырех байтов с указанным адресом. Эти данные хранятся в фиксаторах и могут быть перенесены назад в видеобуфер в процессе записи байта изображения в неизменном виде (например, в случае копирования изображения в другое место экрана), либо с частичным изменением. Конкретная форма заносимого в буфер байта изображения довольно сложным образом зависит от содержимого четырех объектов: регистров-фиксаторов, байта, поступающего из процессора, регистра маски цвета и регистра маски битов.

Поскольку минимальной пересылаемой порцией данных является байт, заполнение экрана изображением осуществляется группами по 8 точек. Естественно, предусмотрена возможность маскирования отдельных битов байта и воздействия на меньшее число точек экрана, в том числе на одну. Однако физически и в этом случае происходит считывание из видеобуфера целого байта с его последующей перезаписью на то же место.

Адаптер EGA обеспечивает три режима записи видеоданных. Номер режима (0, 1 или 2) устанавливается в регистре режима. Доступ к которому осуществляется через порт 3CFh. Всего

через этот порт адресуются 9 управляющих регистров с номерами от 0 до 8; регистр режима имеет среди них номер 5.

Маска цвета

0				x	x	x	x	x	3
0				x	x	x	x	x	2
1				1	1	0	0	1	1
1				0	1	0	0	1	0

2 Битовые плоскости (A000:xxxx)

Ход цвета 3 - бирюзовый

			0	1	0	x	x
			1	1	0	x	x
			1	1	0	x	x
			0	1	0	x	x

Регистры - фиксаторы

			0	0	0	1	1
--	--	--	---	---	---	---	---

Маска битов

			*	*	*	0	1
--	--	--	---	---	---	---	---

Байт из процессора

Рис. 7.3. Взаимодействие аппаратных составляющих видеоадаптера в процессе записи байта изображения по адресу A000:xxxx. \* - старое содержимое бита не изменяется, - содержимое бита может быть любым

Для выбора требуемого регистра используется порт 3CEh (графический адресный регистр), куда заносится номер адресуемого регистра данных:

mov	DX, 3CEh	: Порт номера адресуемого регистра
mov	AL, 5	: Номер регистра режима
out	DX, AL	: Установка адресации к регистру режима
mov	DX, 3CFh	: Порт данных
mov	AL, 0	: Режим записи 0
out	DX, AL	: Установка режима записи

При инициализации графического режима 10h (функцией 0 прерывания Int 10h) устанавливается режим записи 0, являющийся наиболее универсальным. Рассмотрим сначала программирование адаптера в этом режиме.

В процедуре записи видеоданных, помимо регистров - фиксаторов, принимают участие еще два регистра: восьмиразрядный регистр маски битов (bit mask register) и четырехразрядный регистр маски цвета (map mask register) (см. рис. 7.3). Первый

определяет расположение окрашиваемых точек в записываемом байте, второй - их цвет. Регистр маски битов, как и регистр режима, адресуется через порт 3CFh, но имеет номер 8 (устанавливаемый предварительно в регистре режима 3CEh). Регистр маски цвета (наряду с четырьмя другими регистрами) адресуется через порт 3C5h (регистр данных "секвенсера") и имеет номер 2. Для обращения к этому регистру следует вывести его номер в порт 3C4h (адресный регистр секвенсера), после чего выполнять запись через порт 3C5h.

Если установлены в 1 все биты в регистре маски цвета (что соответствует ярко-белому цвету), и в регистре маски битов (что соответствует размаскированию всего байта), то содержимое записываемого по какому-либо адресу видеобуфера байта целиком переносится в битовые плоскости - биты с 1 отображаются в ярко-белые точки, биты с 0 - в черные. Запись 1 и 0 в битовые плоскости в этих условиях осуществляется принудительно, независимо от предыдущего содержимого адресуемого байта в видеобуфере, которое, таким образом, затирается. Если же байт из процессора полностью замаскирован (т.е. в регистр маски битов записано число 00h), в видеобуфер записывается не информация из пересылаемого байта, а содержимое регистров-фиксаторов. При этом, поскольку все биты маски установлены, разрешается прохождение всей информации из регистров-фиксаторов в битовые плоскости, в результате чего в пиксели адресуемого байта переносится цветное изображение, хранящееся в регистрах-фиксаторах.

Менее определенная ситуация возникает, если в регистр маски цвета записывается не число 0Fh, а произвольный код требуемого цвета. В этом случае содержимое пересылаемого байта переносится лишь в те битовые плоскости, которым в регистре маски цвета соответствуют битам регистра маски цвета, соответствующие сброшенным битам регистра маски цвета, не изменяют своего содержимого. То же справедливо и для замаскированных битов маски битов - содержимое соответствующих им битов регистров-фиксаторов переносится лишь в размаскированные маской цвета битовые плоскости. В результате на экране возникает некоторая суперпозиция имевшегося и записываемого изображения. Лишь в случае чистого черного экрана на него будут правильно выводиться точки любого цвета, поскольку черному экрану отвечают нули во всех битовых плоскостях, и плоскости, которые для заданного цвета должны содержать нули, окажутся заполнены заранее правильно. На рис. 7.3 показаны возможные варианты записи изображения в видеобуфер.

В силу отмеченных особенностей функционирования аппаратуры адаптера вывод на экран нового цветного изображения с

полным затиранием старого требует выполнения над каждым байтом изображения двух последовательных операций:

1. Стирание старого изображения, для чего в регистр маски цвета заносится код 0Fh (ярко-белый цвет), в регистр маски битов заносится код 0FFh, чем размаскируется байт данных, и по заданному адресу выводится число 0 (отсутствие цвета, т.е. черный цвет во всех точках). Запись байта в видеобуфер осуществляется командой пересылки:

```
mov byte ptr ES:[BX],0FFh
```

Здесь предполагается, что сегментный регистр ES настроен на начало видеобуфера, а в регистр BX занесен номер адресуемого байта.

2. Запись нового изображения, для чего в регистр маски цвета заносится код требуемого цвета (1 - синий, 2 - зеленый, 3 - бирюзовый и т.д.), в регистре маски битов остается код 0FFh (все биты размаскированы), и по заданному адресу видеобуфера выводится число 0FFh, переносимое код цвета во все точки адресуемого байта изображения. Для выборочной окраски отдельных точек пересылаемый в видеобуфер байт данных должен содержать 1 лишь в битах, соответствующих окрашиваемым точкам. Неокрашенные точки останутся черными. Их можно окрасить в другой цвет, повторив операцию записи для этих точек (см. следующий абзац).

Более сложной выглядит последовательность операций в тех случаях, когда новое изображение накладывается на старое не целым байтом, а лишь в отдельных точках. Основное отличие этой процедуры от уже рассмотренных заключается в необходимости выполнять перед каждой записью чтение соответствующего байта из видеобуфера. Чтение заполняет регистры-фиксаторы содержимым текущего байта изображения, в результате чего при записи этого байта на прежнее место изображение (в точках, оставляемых без изменения) не разрушается. Если опустить операцию чтения, в фиксаторах останется (и будет записано в видеобуфер) изображение, относящееся к последнему прочитанному байту. Таким образом, процедура вывода изображения в самом общем случае выглядит следующими образом:

1. Чтение байта из видеобуфера. Эта операция не требует каких-либо действий с регистрами и выполняется одной командой пересылки:

```
mov AL,ES:[BX]
```

2. Очистка тех точек изображения, которые предполагается окрасить в другой цвет. Для этого в регистр маски цвета заносится код 0Fh, в регистр маски битов - последовательность 1,



отвечающая новому изображению, и по заданному адресу послается число 0 (отсутствие цвета, т.е. стирание).

3. Запись нового изображения, для чего в регистр маски цвета заносится код требуемого цвета, маска битов остается прежней, и по заданному адресу записывается число 0FFh (либо число, совпадающее с кодом маски).

Режим 0, кроме основных процедур, описанных выше, предоставляет ряд дополнительных возможностей: логические операции над содержимым регистров-фиксаторов, ускоренный способ заполнения областей экрана заданным цветом и др.

Для быстрого копирования областей экрана используется режим записи 1. В этом режиме содержимое регистров-фиксаторов просто записывается по заданному адресу видеобuffers. Регистр маски битов не действуют.

Режим 1 отличается от режима 0 тем, что не надо стирать предыдущее изображение - это происходит автоматически при переносе информации из регистров-фиксаторов в битовые плоскости (если в регистре маски цвета записано число 0Fh). Копирование реализуется двумя командами пересылки байта:

```
mov     AL,ES:[SI]
mov     ES:[DI],AL
```

; чтение копируемого байта  
; запись его по новому адресу

Любопытно отметить, что регистр AL выступает здесь в качестве "фиктивной переменной". В результате выполнения первой команды изображение копируется из видеобuffers в регистры-фиксаторы; в результате выполнения второй - из регистров-фиксаторов (не из AL!) в видеобuffers.

Последний режим записи, режим 2, также обеспечивает ускоренное заполнение экрана. В этом режиме цвет указывается в четырех младших битах пересылаемого байта (старшая половина байта не используется). Этим цветом заполняется весь адресуемый байт видеобuffers, либо его часть (в соответствии с содержимым регистра маски битов). В регистре маски цвета по-прежнему должно быть число 0Fh.

#### 7.4. Задачи на прямое программирование адаптера EGA в графическом режиме

Задача 7.7. Запись байтами в режиме 0 с полным затиранием предыдущего изображения. Заполнять в цикле прямоугольную область экрана разными цветами, сравнить скорость заполнения экрана при прямом программировании и при использовании драйвера BIOS (Int 10h, функция 0Ch).

; Основные фрагменты программы  
; Установим графический видеорежим

Подготовим адресацию к видеобuffers

```
...
; Установим режим записи 0
mov     DX,3CEh
mov     AL,5
mov     DX,AL
inc     DX
```

```
mov     AL,0
out     DX,AL
mov     BX,0
mov     CX,80*175
```

begin;  
; Сначала очистим байт изображения

```
; Установим полную маску цвета
mov     DX,3C4h
mov     AL,2
out     DX,AL
inc     DX
```

```
mov     AL,0Fh
out     DX,AL
```

```
; Установим маску битов
mov     DX,3CEh
mov     AL,8
out     DX,AL
inc     DX
```

```
mov     AL,0FFh
out     DX,AL
; Очистим байт (все пиксели байта)
mov     byte ptr ES:[BX],0
```

; Теперь выведем байт изображения

```
; Установим цвет
mov     DX,3C4h
mov     AL,2
out     DX,AL
inc     DX
```

```
mov     AL,color
out     DX,AL
```

; Выведем байт пикселей (выводятся все пиксели байта)

```
mov     byte ptr ES:[BX],0FFh
inc     BX
loop    begin
```

; Остановим программу с возможностью выхода в DOS при нажатии заданной клавиши с помощью, например, функции 0Bh прерывания 21h (выход из программы по <Ctrl>/C)

```
...
inc     color
add     stbyte,80*20
jmp     newcolor
```

; Адресный регистр контроллера  
; Регистр режима  
; Установка адресуемого регистра  
; DX=3CEh - регистр данных  
; контроллера  
; Режим 0  
; Установка режима записи  
; Начнем с байта 0  
; Столько байтов всего (175  
; графических строк)

; Адресный регистр секвенсера  
; Регистр маски цвета  
; Установка адресуемого регистра  
; DX=3C5h - регистр данных  
; секвенсера  
; Все биты ("ярко-белый")  
; Установка маски цвета

; Адресный регистр контроллера  
; Регистр маски битов  
; Установка адресуемого регистра  
; DX=3CEh - регистр данных  
; Все пиксели байта размаскировать  
; Установка маски битов

; Адресный регистр секвенсера  
; Регистр маски цвета  
; Установка адресуемого регистра  
; DX=3C5h - регистр данных  
; секвенсера  
; Цвет  
; Установка цвета

; К следующему байту  
; Повторить вывод байта

; Остановим программу с возможностью выхода в DOS при нажатии заданной клавиши с помощью, например, функции 0Bh прерывания 21h (выход из программы по <Ctrl>/C)  
; Сменим цвет  
; Сдвиг на 20 строк  
; Повторим с новым цветом

```

;Поля данных
color db
stbyte dw

```

```

1
0

```

```

;Код цветового регистра
;Номер начального байта

```

Задача 7.8. Запись пикселями с частичным затиранием предыдущего изображения. Заполнить часть экрана произвольным цветом. После этого вывести на экран отдельные цветные точки или их комбинации.

```

;Основные фрагменты программы
;Подготовим адресацию к видеобufferу

```

```

;Установим видеорежим

```

```

;Установим режим записи 0

```

```

;Выведем синий прямоугольник
xor     BX, BX
mov     CX, 80*125

```

```

;Начнем с байта 0
;125 строк

```

```

;При рисовании по черному очистка не нужна
;Установим маску битов 0FFh

```

```

;Установим синий цвет

```

```

;Выведем байт пикселей (выводятся все пиксели байта)
begin:  mov     byte ptr ES:[BX], 0FFh

```

```

inc     BX
;К следующему байту
;Повторить вывод байта

```

```

;Набросаем CX байтов цветных точек
loop    begin

```

```

mov     CX, 100
mov     BX, 80*5+1
;Байт 1 строки 5

```

```

;Установим маску битов на 3 пиксела в байте -
;два самых левых и самый правый (число 31h)

```

```

mov     DX, 3CEh
mov     AL, 8
out     DX, AL
inc     DX
mov     AL, 0C1h
out     DX, AL

```

```

;Адресный регистр контроллера
;Регистр маски битов
;Установка адресуемого регистра
;DX=3CEh - регистр данных
;Маска
;Установка маски битов

```

```

;Каждый выводимый пиксел надо предварительно очищать
;Установим полную маску цвета (0FFh), не зная, какой был цвет
;и какие битовые плоскости надо стирать

```

```

dots:

```

```

;Прочитаем и снова запишем подчищенный байт
mov     AL, ES:[BX]
;Чтение обязательно
mov     byte ptr ES:[BX], 0
;Очистка

```

```

;Установим цвет color в регистре маски цвета

```

```

;Маска битов прехняя

```

```

;Теперь запишем заданные пиксели байта

```

```

mov     byte ptr ES:[BX], 0FFh ;Вывод байта
add     BX, 162 ;Через столько байтов (2 строки + 2 байта)
inc     color ;Следующий цвет

```

```

loop    dots
;Повторить
;Основные фрагменты программы

```

```

;Завершим программу

```

```

;Поля данных
color db

```

```

1

```

```

;Цвет

```

Задача 7.9. Копирование байтов в режиме 1. Вывести на экран произвольный байт изображения. Скопировать его в другое место экрана. Использовать отложенный алгоритм для заполнения экрана в режимах 0 и 1.

```

;Основные фрагменты программы
;Установим графический видеорежим

```

```

;Подготовим адресацию к видеобufferу

```

```

;Установим режим записи 0

```

```

;Выведем байт изображения (установив маски битов и цвета)

```

```

;Установим режим записи 1

```

```

;Будем копировать байт
;Установим маску цвета 0Fh

```

```

;Теперь собственно копирование
mov     CX, 350*26

```

```

copy:   mov     AL, ES:[BX]
add     BX, 3
mov     ES:[BX], AL
loop    copy

```

```

;Столько копий (приблизительно
;на весь экран)
;Чтение байта
;Новое место (на 3 байта правее)
;Запишем на новое место
;Повторим CX раз

```

Задача 7.10. Запись в режиме 2. Заполнить несколько перекрывающихся или накладывающихся прямоугольных областей экрана разными цветами.

```

;Основные фрагменты программы

```

```

;Подпрограмма гест вывода прямоугольника в режиме 2. Параметры:

```

```

;lg=ширина (байтов)
;ht=высота (пикселей)
;xo=X-координата начала (байтов)
;yo=Y-координата начала (пикселей)
;ху=временная переменная

```

```

rget    proc
mov     AL, yo
mov     BL, 80
mul     BL
add     AX, xo
mov     ху, AX
mov     CX, hi

```

```

;Y-координата начала (пикселей)
;Число байтов в строке
;AL*BL->AX
;Прибавим смещение вправо
;Сохраним
;Высота прямоугольника (точек)

```

168

```

mov     SI, 0
push    CX
12:     mov     CX, len
mov     BX, xy
mov     AL, color
mov     ES:[BX][SI], AL
11:     mov     BX
inc     SI
loop    SI, 80
add     CX
pop     CX
loop    12
ret
endp

```

; Начало основной программы  
; Установим графический видеорежим

...  
; Подготовим адресацию к видеобufferу

...  
; Установим режим записи 2

...  
; Установим маску битов (0FFh)

...  
; Выведем 1-й прямоугольник

```

mov     x0, 15
mov     y0, 50
mov     color, 1
mov     hi, 240
mov     len, 50
call    rect

```

; Выведем 2-й прямоугольник

```

mov     x0, 20
mov     y0, 90
mov     color, 14
mov     hi, 160
mov     len, 40

```

call rect  
; Выведем 3-й прямоугольник

...  
; Остановим программу

...  
; Завершим программу

...  
; Поля данных

x0	dw	0
y0	dw	0
len	dw	0
hi	dw	0
xy	dw	0
color	db	0

; Сдвиг от угла  
; Сохраним в стеке  
; Ширина прямоугольника (байтов)  
; Сдвиг от 0,0  
; Цвет  
; Вывод в видеобuffer  
; К следующему байту  
; Цикл по строке  
; Прибавим длину строки  
; Высоту из стека  
; Цикл по строкам  
; Возврат в программу

; Задание начальных значений

; Вызов подпрограммы

; Задание начальных значений

; Вызов подпрограммы

; X-координата начала (байтов)  
; Y-координата начала (пикселей)  
; Ширина (байтов)  
; Высота (пикселей)  
; Временная переменная  
; Цвет

## 8. Управление памятью и процессами

### 8.1. Системные средства распределения памяти

Система MS-DOS воспринимает в качестве выполнимых программ файлы двух типов: .COM и .EXE. Программы типа .COM являются односегментными: и программные строки, и области данных, и стек размещаются в единственном сегменте и не могут, следовательно, в сумме превышать 64 Кбайт. Такие программы хранятся на диске в файлах типа .COM в виде абсолютного образа памяти и переносятся в оперативную память в процессе загрузки практически без изменений.

Программы типа .EXE не имеют таких жестких ограничений. Они могут включать любое число сегментов программы, данных и стека и их суммарный размер ничем не ограничен. Файл типа .EXE, кроме собственно программы, включает еще специальную системную область - заголовок, в котором хранятся характеристики программы (длина, контрольная сумма, адрес точки входа и др.), а также информация о настройке программных кодов при их загрузке в память (карта перемещений). Необходимо считывать и обрабатывать эту информацию, удлиняет процесс загрузки, но программа оказывается более мобильной, так как ее отдельные сегменты могут быть загружены в несвязные участки памяти и, в частности, совместно использоваться несколькими процессами, что важно при организации многозадачных режимов.

Компоновщик (редактор связей, построитель задачи) LINK, обрабатывая объектный модуль программы, полученный в результате трансляции исходного текста, образует файл с расширением .EXE. Если программа предназначена для выполнения в формате .COM (для чего она должна быть написана с соответствующими ограничениями), полученный на выходе компоновщика файл .EXE следует преобразовать в формат .COM. Это преобразование выполняется с помощью системной утилиты EXE2BIN.

В настоящее время большая часть прикладных программ разрабатывается в формате .EXE. Формат .COM используется в



основном при написании резидентных программ, драйверов и обработчиков прерываний.

Любая программа содержит специальную системную таблицу префикс программного сегмента (PSP), имеющую размер 256=100h байтов.

В программах типа .COM место под PSP резервирует программист, начиная программу директивой `ORG 100h`; PSP включается, таким образом, в состав единственного сегмента программы. Заполняется область PSP функцией `Ehss` при загрузке программы в память. В программах типа .EXE место под PSP не резервируется; он "пристраивается" к программе в процессе ее загрузки.

Содержимое PSP используется DOS в процессе выполнения программы; к нему приходится также обращаться в прикладных программах. Некоторые поля, включенные в PSP, в настоящее время потеряли свое значение. В таблице 8.1 приведено содержимое префикса программного сегмента. Назначение и возможности использования полей PSP будут описываться по мере рассмотрения соответствующих программных средств.

Таблица 8.1. Содержимое PSP.

Смещение	Число байтов	Описание
00h	2	Команда <code>INT 20h</code> для совместимости с CP/M
02h	2	Сегментный адрес первого свободного байта за пределами программы
05h	1	Вызов функции 5 CP/M ( <code>FAR JMP</code> на 00000h)
06h	2	Размер первого сегмента для .COM-файлов (CP/M)
08h	2	Остаток от <code>FAR JMP</code> в байте 05h
0Ah	4	Адрес перехода после завершения программы (вектор 22h)
0Eh	4	Адрес обработчика <code>Ctrl/C</code> (вектор 23h)
12h	4	Адрес обработчика критической ошибки (вектор 24)
16h	2	Сегментный адрес родительского PSP
18h	20	Таблица файлов задания JFT
2Ch	2	Сегментный адрес блока окружения программы
2Eh	4	SS:SP программы при входе в последний вызов <code>INT 21h</code>
32h	2	Число элементов в JFT (по умолчанию 20)
34h	4	Адрес JFT (по умолчанию PSP:0018h)
40h	2	Версия DOS в формате функции 30h
50h	3	Команды <code>INT 21h/RETf</code> (вызов DOS и возврат)
5Ch	16	Первый FCB, заполняется первым аргументом команды данной программы в формате записи каталога
6Ch	16	Второй FCB, заполняется вторым аргументом команды
80h	1	Длина аргументов команды вызова данной программы
81h	127	Аргументы команды вызова данной программы и 00h

При загрузке в память программы (как .COM, так и .EXE) DOS, независимо от фактического размера программы, выделяет ей по умолчанию всю наличную память (рис. 8.1). Такой алгоритм выделения памяти определяется (для программ типа .EXE) значением поля со смещением `Ch` в заголовке файла загрузочного модуля (см. табл. 4.4). По умолчанию компонент `LINK` этого значения можно изменить, сократив размер выделяемой программе памяти.



Рис. 8.1. Выделение памяти загружаемой программе.

Корректная программа обычно освобождает лишнюю для ее функционирования память и работает далее лишь в пределах закрепленного за ней адресного пространства. Это особенно важно для драйверов устройств и других программ, резидентных в памяти. Если драйвер после загрузки и инициализации не освободит лишнюю память, система перестанет функционировать, так как некуда будет загружать пользовательские программы. С другой стороны, активной программе может на время потребоваться дополнительная память для размещения, например, результатов вычислений. В этом случае программа может поставить запрос к DOS на выделение требуемого блока памяти, причем DOS в ответ сообщает об объеме свободной памяти, что позволяет программе "приспособиться" к конкретным условиям работы. Такой механизм важен для нормального функционирования программ, активно использующих оперативную память, так как в реальных условиях ввиду широкого использования резидентных программ (как системных, так и прикладных), объем свободной памяти может изменяться в широких пределах.

Программа, загруженная в память, включает три важных для программирования компонента: окружение, префикс программного сегмента и собственно программу, которая (в случае

файла .EXE) может состоять из нескольких сегментов и в принципе занимать несколько несвязных участков памяти (рис. 8.2).

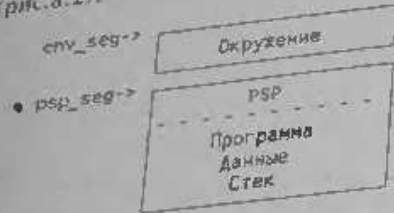


Рис. 8.2. Программа с ее окружением в памяти.  
env\_seg - сегментный адрес окружения; psp\_seg - сегментный адрес префикса программного сегмента (идентификатор программы, ID).

В процессе начальной загрузки DOS создает начальное окружение, в котором будут работать активизируемые программы и, прежде всего, командный процессор COMMAND.COM. Окружение представляет собой область памяти, в которой в виде символьных строк записаны значения переменных, называемых переменными окружения. Формат окружения приведен на рис. 8.3.

```
*переменная_1 = значение_1',0
*переменная_2 = значение_2',0
*переменная_3 = значение_3',0
...
*переменная_n = значение_n',0,0,1,0
*диск:\путь\имя_файла.расширение',0
```

Рис. 8.3. Формат окружения (все числа занимают по 1 байту).

Имеется ряд переменных окружения, имена которых зарезервированы и известны системе, однако пользователь может включать в окружение и свои переменные для использования их прикладными программами. Окружение служит для передачи программ (как системным, так и прикладным) требуемых параметров. Параметры (в виде значений определенных переменных окружения) заносятся в окружение с помощью системной команды SET. Системные и прикладные программы могут анализировать текущий состав окружения и извлекать из него относящиеся к ним параметры. Например, системная команда DIR ожидает найти в своем окружении (т.е. в окружении командного процессора COMMAND.COM) переменную DIRCMD с перечнем ключей, и, если такая переменная имеется, команда

DIR организует свой вывод в соответствии с указанными ключами. Пусть в начале сеанса на машине мы выполнили команду SET DIRCMD=/A:-D /O:-S

После этого команда DIR без указания ключей будет выводить на экран содержимое указанного каталога без упорядочивания размера (ключ /O:-S).

Механизм программного анализа окружения и извлечения параметров будет рассмотрен позже.

Размер окружения задается на этапе конфигурирования системы с помощью команды SET COMSPEC, которая устанавливает местоположение и некоторые характеристики командного процессора. Например, команда

```
SET COMSPEC=C:\DOS_62\COMMAND.COM/P/E:400
```

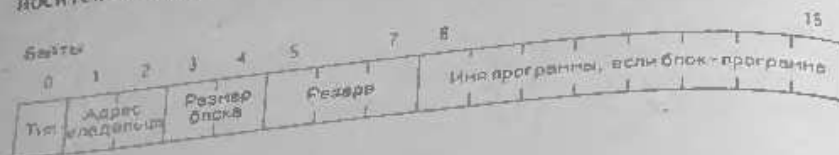
говорит системе, что командный процессор находится в каталоге DOS-62, является резидентным (ключ /P) и работает в окружении размером 400 байтов (ключ /E). Указанная команда позволяет увеличить размер начального окружения, что нужно в тех случаях, когда предполагается включать в окружение большое количество переменных.

Полная спецификация файла с программой, включаемая системой в окружение, позволяет, найдя в памяти окружение некоей неизвестной программы, узнать имя этой программы (в самой загруженной в память программе, так же, как и в файле загрузочного модуля, нет информации о ее имени).

Обычно DOS размещает окружение над программой вплотную к ней. Если, однако, при загрузке программы перед ней обнаруживается свободный участок (он мог возникнуть, если запущенная перед этим резидентная программа после загрузки освободила свое окружение), окружение данной программы размещается в этой "дырке" в памяти. В любом случае сегментный адрес окружения помещается системой в PSP программы. Поскольку окружение всегда начинается на границе параграфа, его местоположение однозначно описывается сегментным адресом (см. табл. 8.1).

DOS выделяет оперативную память участками произвольной длины, которые обычно называют блоками. Размер блока задается в параграфах и в принципе может иметь значение от 0 до FFFFh, т.е. все адресное пространство может быть отдано одному блоку. DOS ведет учет занятой и свободной памяти с помощью 16-байтовых структур - блоков управления памятью (Memory Control Block, MCB). Каждый MCB располагается в

памяти непосредственно перед тем блоком, к которому он относится. Формат MCB приведен на рис. 8.4.



Значения типов  
 'M' - '0Fh' - промежуточный блок  
 'Z' - '5Ah' - последний блок

Рис. 8.4. Формат блока управления памятью MCB.

Рассмотрим назначение полей MCB на примере распределения памяти в реальной вычислительной системе (рис. 8.5).

Сегментные адреса		DOS (около 64K)				
102Ch	'M'	102F	0001	Мусор		МСБ окружения VGA866
102Dh	0, 0, 1, 0, 'VGA866.COM'					Окружение VGA866
102Eh	'M'	102F	01B7	'VGA866'		МСБ программы VGA866
ID=102Fh	PSP ----- VGA866.COM					Резидентная программа VGA866.COM
11E6h	'M'	11E7	0116	'COMMAND'		Резидентная программа COMMAND.COM
ID=11E7h	PSP ----- COMMAND.COM					
12FDh	'M'	11E7	0010	Мусор		МСБ окружения COMMAND.COM
12FEh	'PATH'=C:\DOS', 0 'PROMPT=\$PSG', 0, 0 1, 0, 'COMMAND.COM'					Окружение COMMAND.COM
130Eh 130Fh	'Z'	0000	BCF1	Мусор		МСБ свободной памяти
Свободная память						

Рис. 8.5. Пример реального распределения памяти.

Использовалась только обычная память, поэтому около 64 Кбайт в начале памяти занимает DOS. В файл CONFIG.SYS была включена директива  
 INSTALL=VGA866.COM

загрузки единственной резидентной программы (русификатора экрана) еще на этапе конфигурирования системы, до загрузки COMMAND.COM. Такой метод загрузки резидентной программы хорош тем, что окружение программы оказывается минимального размера (всего 16 байтов). На рисунке показано минимальное количество наиболее характерных полей приведенных структур данных.

Начало памяти занимают программы DOS. Вообще говоря, внутри DOS тоже можно выделить отдельные блоки с сопровождающими их MCB, однако для пользователя они редко представляют интерес и на рисунке не показаны. Свободная память (после DOS) начинается с сегментного адреса 102Ch. При загрузке программы VGA866.COM в памяти сначала строится MCB окружения этой программы, ее окружение, еще один MCB, описывающий блок с программой и, наконец, загружается с диска сама программа VGD866. Оба MCB имеют идентификаторы 'M', так как они описывают промежуточные (не последние) блоки памяти. "Владельцем" всех этих структур считается программа, которая идентифицируется для системы сегментным адресом ее PSP. Ввиду важности этого адреса он носит специальное название "идентификатор программы" (ID или PID). ID программы указывается в поле владельца блока памяти и в MCB окружения, и в MCB самой программы. В MCB окружения указан размер блока окружения (1 параграф), а поле имени программы заполнено мусором, так как окружение не является программой (см. рис. 8.4).

Окружение VGA866 имеет минимальный размер и содержит только четыре байта со служебной информацией и имя программы. Путь к программе отсутствует, так как программа загружена не командным процессором (которого еще нет в памяти), а самой DOS.

MCB программы VGA866 содержит размер блока с программой (1B7h параграфов) и имя программы без пути и расширения. Последнее естественно, так как в MCB для имени программы зарезервировано всего 8 байтов (см. рис. 8.4).

Начиная с сегментного адреса 102Fh располагается сама программа русификатора экрана, начинающаяся, как и любая программа, с ее PSP.

Загрузив программы, указанные в директивах INSTALL файла CONFIG.SYS, DOS приступает к загрузке командного процессора. Свое окружение (начальное окружение) будет строить сам командный процессор, поэтому оно будет расположено за



программой COMMAND.COM. Перед программой COMMAND.COM в памяти строится MCB блока с этой программой. Владелец блока с программой является сама программа, поэтому в MCB указывается ее ID (11E7h). Размер резидентной части COMMAND.COM составляет 116h параграфов.

Вслед за программой COMMAND.COM располагается блок начального окружения, предваряемый MCB. Окружением программы владеет сама программа, поэтому в поле владельца окружения указан ID COMMAND.COM (11E7h). В конкретном примере окружение имеет размер 10h параграфов (160 байтов). Окружение не является программой, поэтому поле имени программы в MCB заполнено мусором.

Начальное окружение включает всего две переменные PATH и PROMPT, значения которых завершаются двумя нулевыми байтами, за которыми следуют два байта со служебной информацией и имя программы-владельца (COMMAND.COM).

Следующий и последний MCB располагается по сегментному адресу 130Eh. Идентификатор 'Z' указывает, что этот MCB описывает последний блок, занимающий, очевидно, всю оставшуюся память. Владелец этого блока является он сам (0000), то-есть блок свободен, и его размер составляет 8CF1h параграфов.

Для динамического управления памятью используются следующие функции DOS:

48h - назначить блок памяти;

49h - освободить блок памяти;

4Ah - изменить размер назначенного блока памяти.

С помощью функции 48h программа может затребовать у DOS до 1 Мбайт памяти, хотя практически разумно получать память сегментами по 64 Кбайт. Размер требуемого блока (в параграфах) указывается в регистре BX. В случае успешного завершения функции сегментный адрес выделенного блока памяти возвращается в AX; программа, переслав этот адрес в сегментный регистр данных (обычно ES), может работать с выделенной памятью, которая с точки зрения структуры программы представляет собой дополнительный сегмент данных. Если DOS не смогла выделить память (о чем говорит установленный флаг CF), в регистре BX возвращается число свободных параграфов, и программа может проанализировать это значение с целью определения дальнейшей стратегии.

При каждом выделении блока памяти DOS создает блок управления памятью, размещаемый непосредственно перед выделяемым блоком. Следует заметить, что работа с блоками управления памятью является исключительной прерогативой DOS;

случайное разрушение блока приводит к выдаче сообщения "Ошибка распределения памяти" и останову системы.

Для освобождения блока памяти, выделенного программе с помощью функции 48h, используется функция 49h. Нельзя освободить только часть выделенной памяти (для этого используется функция изменения размера блока 4Ah).

Для изменения размера блока памяти, ранее выделенного программе функцией 48h, а также для изменения собственного размера 4Ah. Новый размер (в параграфах) изменяемого блока памяти (с точки зрения структуры программы - сегмента или группы сегментов) передается в регистре BX, а сегментный адрес этого блока - в регистре ES. Функция 4Ah обычно используется для сокращения размера программы до реально необходимого (вспомним, что при загрузке DOS выделяет программе свой реальный размер, а также своей базовый сегментный адрес. Любая программа (и .EXE, и .COM) начинается с префикса программного сегмента, причем при загрузке программы базовый адрес PSP находится в регистрах ES и DS (а для программ .COM также и в регистрах CS и SS). Если программа явным образом не модифицирует содержимое ES, то этот регистр уже оказывается настроен должным образом для вызова функции 4Ah).

Методика определения размера программы зависит от типа программы (.EXE или .COM), а также от того, надо ли сохранить в памяти всю программу или только ее часть. Если в программе .EXE, состоящей из трех сегментов - программного, данных и стека - предусматривается освобождение лишней памяти, то для определения размера программы следует включить в программу фиктивный пустой сегмент, расположив его в самом конце программы, после всех остальных сегментов. Однако этого недостаточно. Дело в том, что транслятор и компоновщик следят за классами сегментов и сегменты одного класса размещают в памяти друг за другом в порядке, соответствующем их порядку в программе. Сегменты, для которых класс не указан, принадлежат к безымянному классу. Если не указать для нашего фиктивного сегмента класс, он будет находиться в одном классе с другим сегментом без класса, именно, сегментом данных и будет загружен в память сразу после него, то-есть до сегмента стека. Следовательно, нашему фиктивному сегменту надо присвоить какой-то (произвольный) класс. Учитывая, что при загрузке программы ее начальный адрес (т.е. адрес PSP) заносится в регистр ES, для определения размера программы потребуются следующие строки:

```

text    segment 'code'
        mov     AX, fict
        mov     BX, ES
        sub     AX, BX
        ends
text    data
        segment
        ends
data    stack
        segment stack 'stack'
        ends
stack   fict
        segment 'abcd'
        ends
fict    ends

```

;Сегментный адрес конца программы  
;Сегментный адрес начала программы  
;AX=размер программы в параграфах

;Фиктивный сегмент  
;нулевой длины

Размер односегментной программы .COM определяется иначе. Он равен разности значений счетчика текущего адреса в конце и начале программы плюс размер PSP. Однако при использовании функции 4Ah в программе .COM необходимо иметь в виду, что при загрузке программы весь остаток пространства 64-Кбайт сегмента, не занятый собственно программой и ее данными, отдается стеку, а указатель стека инициализируется числом FFFFh и указывает, таким образом, на последнее слово сегмента. Для того, чтобы сократить размер стека до разумной величины, перед освобождением лишней памяти указатель стека следует переместить в конец специально выделенной области стека:

```

text    segment 'code'
        org     100h
main    proc
        ...
        mov     SP, offset newstk
        endp
main    ...
        dw      64 dup (?)
newstk=$
text    ends

```

;Данные  
;Область стека

Размер программы в параграфах будет равен  $(newstk - main + 100h + 0Fh) / 16$ . Здесь к вычисленному размеру программы добавлен размер PSP (100h байтов) и число Fh для округления до следующего параграфа. Деление всей суммы на 16 дает число параграфов.

Рассмотрим программу типа .COM, в которой по ходу ее выполнения вычисляется объем некоторого буфера, с которым программа будет работать, и динамически выделяется требуемая память. Программа должна содержать следующие блоки:

- определения объема требуемой памяти в параграфах;

- переноса указателя стека на новое место;
- освобождения лишней памяти;
- выделения требуемого объема памяти в виде одного или нескольких сегментов;
- использования выделенных сегментов так, как это требуется программе (чтение файла, хранение данных и т.д.);
- освобождения выделенной памяти;
- завершения программы.

Структурная схема программы может выглядеть следующим образом:

```

text    segment 'code'
        org     100h
proc
        mov     SP, offset newstk
        mov     AH, 4Ah
        mov     BX, (newstk - main + 10Fh) / 16
        int     21h
        add     AX, 0Fh
        mov     DX, 0
        mov     BX, 16
        div     BX
        mov     BX, AX
        mov     AH, 48h
        int     21h
        mov     ES, AX
        mov     word ptr ES:[DI], 1234h
        mov     AX, word ptr ES:[SI]
        mov     AH, 49h
        int     21h
        ...
        endp
main    proc
        dw      ...
        db      ...
        ...
        dw      128 dup(0)
newstk=$
text    ends

```

;Функция изменения размера блока  
;памяти  
;Фактический размер  
;программы  
;Определим из каких-то соображений объем памяти, которую надо динамически выделить программе. Пусть эта величина (в байтах) оказалась в AX  
;Для округления числа параграфов  
;Старшая половина делимого  
;Делитель (размер параграфа)  
;Частное в AX, остаток (в DX) нам не нужен  
;Выделим динамически требуемую память  
;Требуемое число параграфов  
;Функция выделения памяти  
;Настроим ES на выделенный сегмент  
;Поработаем с выделенным блоком  
;Пример записи в него, если в DI требуемое смещение:  
;Пример чтения из него, если в SI требуемое смещение:  
;Освободим выделенный блок. ES уже указывает на него  
;Функция освобождения памяти  
;Завершим программу обычным образом

Структура аналогичной программы типа .EXE будет отличаться только методикой определения фактического размера программы (а также, естественно, наличием отдельных сегментов данных и стека).

## 8.2. Организация дочерних процессов

Программы загружаются в память для выполнения с помощью функции DOS Exec (Int 21h, функция 4Bh), которая играет роль системного загрузчика. Если пользователь запускает программу, вводя командную строку с клавиатуры, то функцию Exec вызывает командный процессор COMMAND.COM. В других случаях функция Exec может быть вызвана оболочкой DOS или другим программным интерфейсом, либо загруженной и выполняемой программой, в том числе пользовательской. Такая динамическая загрузка и запуск дочерних программ позволяет организовать иерархические программные комплексы, в которых родительский процесс, в зависимости от конкретных условий, инициирует те или иные дочерние процессы, а те, в свою очередь, могут вызывать к жизни процессы следующего уровня и т.д. При этом надо иметь в виду, что система MS-DOS не является системой реального времени и не поддерживает параллельные процессы. В иерархическом программном комплексе все его составляющие выполняются только поочередно, друг за другом.

Как уже отмечалось, программа, загруженная в память, занимает, как правило, 4 блока памяти (окружение программы с его блоком управления памятью MCB и сама программа с ее MCB); самостоятельным элементом программы является ее PSP (рис. 8.6).

Окружение для командного процессора, создаваемое в процессе начальной загрузки, чаще всего содержит переменные COMSPEC, PROMPT и PATH, которые заносятся в окружение из файла AUTOEXEC.BAT, и может иметь, например, следующий вид:

```
COMSPEC=C:\DOS\COMMAND.COM
PROMPT=$P$G
PATH=C:\; C:\DOS; C:\TOOLS
```

Первая строка описывает путь к файлу командного процессора COMMAND.COM. Системные и прикладные программы, загружаясь в память, могут затирать транзитную часть командного процессора, и после завершения такой программы командный процессор надо снова загрузить с диска. Система (конкретно - резидентная часть COMMAND.COM) берет имя и местонахождение файла с командным процессором из значения переменной окружения COMSPEC.

Строка с переменной PROMPT задаст вид системного запроса, который в приведенном примере состоит из полного пути к текущему каталогу и изображения стрелки (если переменная PROMPT не определена, устанавливается стандартный системный запрос, который включает лишь имя текущего диска и стрелку; работать с таким запросом крайне неудобно).

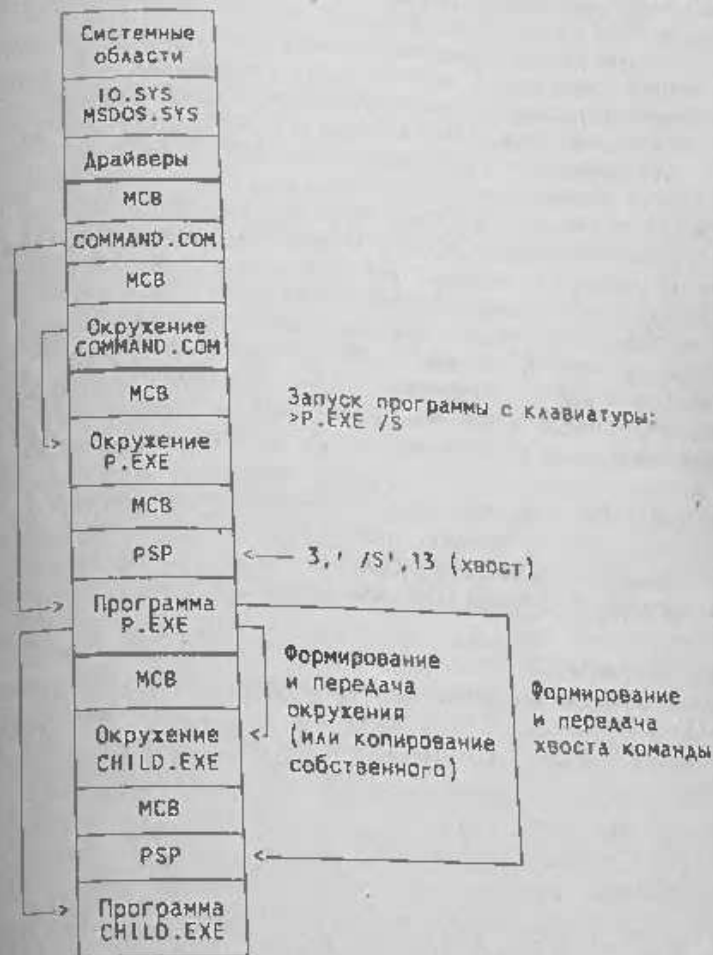


Рис. 8.6. Запуск дочернего процесса.

Строка с переменной PATH определяет пути, которые автоматически просматриваются DOS при поиске на диске запущенной пользователем программы.



Пользователь может включить в окружение строки определения дополнительных переменных с помощью команды SET. При вводе команды с клавиатуры значения этих переменных будут действовать только в данном сеансе работы на компьютере (до его перезагрузки). Если же включить команду SET в файл AUTOEXEC.BAT, она будет автоматически выполняться каждый раз при загрузке системы и значения дополнительных переменных будут определены всегда.

После того, как выполнена начальная загрузка DOS, активной программой (текущим процессом) становится командный процессор, который фактически ничего не делает, ожидая ввода команды с клавиатуры. Получив команду оператора на запуск программы (например, программы P.EXE на рис. 8.6), COMMAND.COM, активизируя эту программу (с помощью функции DOS Exec), передает ей свое окружение. Практически это осуществляется путем копирования строк окружения в новую область памяти, которая обычно располагается перед загружаемой программой. Если загруженная программа по ходу своего выполнения в свою очередь активизирует дочерний процесс следующего уровня (с помощью той же функции Exec), она еще раз копирует свое окружение, передавая его запускаемой ею программе. При этом родительский процесс может передать дочернему свое окружение в неизменном виде, а может включить в него новую информацию. Таким образом, каждая программа, загруженная в память, имеет собственную копию окружения, причем эти копии могут совпадать, а могут и различаться. После завершения программы и возврата управления родительскому процессу окружение дочерней программы уничтожается, и родительский процесс продолжает работать со своим собственным окружением.

Многие программы не используют переменные окружения и передача им окружения носит формальный характер. С другой стороны, с помощью этого механизма родительские процессы могут передавать дочерним значительные объемы информации (максимальный размер блока окружения составляет 32 Кбайт).

Вторая структура данных, формируемая функцией Exec при активизации вызываемой программы, PSP занимает объем 256 байтов и фактически включается в состав загружаемой программы. В слово PSP с адресом 2Ch (относительно его начала) DOS помещает сегментный адрес окружения программы, а в область, начинающуюся с адреса 80h - параметры командной строки, или "хвост команды" (/S на рис.8.6). В байте по адресу 80h находится длина параметров команды без учета завершающего команду кода 13, а затем располагаются сами параметры в виде символьной строки. С помощью хвоста командной строки запускаемой программе часто передаются ключи,

управляющие ее работой, а также имена рабочих файлов. Так, программа P.EXE могла быть запущена командой  
P.EXE A:\WORKFILE.001/S

в которой программе, помимо ключа /S, определяющего режим ее работы, передается еще и имя и местоположение рабочего файла WORKFILE.001. Если пользовательскую программу предполагается запускать с передачей ей каких-то параметров, то в программе должны быть строки извлечения хвоста команды из PSP и его анализа.

Для загрузки и выполнения дочерней программы родительская программа должна вызвать прерывание Int 21h с функцией 4Bh при следующем состоянии регистров:

AH=4Bh;

AL=00h (подфункция загрузки дочерней программы);

DS:DX=адрес строки со спецификацией файла с дочерней программой;

ES:BX=адрес блока параметров.

Спецификация файла с дочерней программой должна быть однозначной и не может содержать шаблоны групповых операций. Спецификация должна включать в явной форме расширение .COM или .EXE. Если имя файла дано без указания пути, дочерняя программа ищется в текущем каталоге текущего диска.

Командный файл .BAT нельзя запустить с помощью функции Exec непосредственно; вместо этого следует, используя Exec, вызвать копию командного процессора COMMAND.COM и передать ему имя командного файла в хвосте команды после ключа /S.

Блок параметров служит для передачи функции Exec необходимой информации и включает в себя адреса следующих объектов:

окружения (сегментный адрес, однословная ячейка);  
параметров командой строки (полный адрес, двухсловная ячейка);  
двух блоков управления файлами (FCB) (полные адреса, двухсловные ячейки).

Блок параметров и связанные с ним структуры данных выглядят, например, следующим образом:

parmbk	dw	envirseg	;Сегмент окружения
	dd	cmdtail	;Адрес хвоста команды
	dd	fcbl	;Адрес первого FCB
	dd	fcbl2	;Адрес второго FCB
chname	db	'CHILD.EXE',0	;Имя дочернего процесса
cmdtail	db	9,' FILE.TXT',0Dh	;Хвост команды
fcbl	db	37 dup (0)	;Первый FCB

```

fcb2      db      37 dup (0)          ;Второй FCB
envirseg  segment para "ENVIR"       ;Сегмент окружения
          db      "COMSPEC=D:\DOS\COMMAND.COM", 0
          db      "WORKFILE=A:\FILES", 0
          db      0
envirseg  ends

```

Адрес окружения составляет одно слово и содержит только сегментный адрес, поскольку блок окружения всегда начинается на границе сегмента. Если дочерний процесс должен обладать специфическим окружением, в родительской программе предусматривается и заполняется необходимой информацией соответствующий сегмент (вырожденный на границу параграфа). Если же дочерний процесс должен наследовать окружение родительского, то специальный сегмент не создается, а в качестве адреса окружения в блоке параметров указывается 0.

В приведенном выше примере сегмент окружения заполнен в родительской программе явным образом и содержит две переменные: системную переменную COMSPEC с описанием местоположения и имени командного процессора и индивидуальную переменную дочернего процесса WORKFILE, служащую, например, для передачи дочерней программе CHILD.EXE имени каталога с вспомогательными файлами. Этот пример несколько нетипичен в том отношении, что спецификация командного процессора обычно задается не в конкретной программе, а еще при запуске системы в файле AUTOEXEC.BAT командой SET COMSPEC=..., откуда она переписывается в окружение командного процессора. По мере активизации дочерних процессов значение переменной COMSPEC передается из окружения и окружение переменной COMSPEC индивидуальной переменной (WORKFILE в приведенном примере), то она может быть включена в окружение дочернего процесса программным образом (как это показано в примере), либо установлена путем ввода с клавиатуры команды SET WORKFILE=...

В любом случае для использования информации, передаваемой через окружение, дочерняя программа извлекает из слова 2Ch в PSP физический адрес окружения и ищет затем в блоке окружения интересующие ее переменные.

Хвост команды используется для передачи дочерней программе параметров вызова, чаще всего имен рабочих файлов, а также ключей, определяющих режим работы программы. В процессе загрузки дочерней программы DOS перешлет хвост команды из блока параметров функции Exec на его "законное" место - в префикс программного сегмента дочерней программы, где он будет располагаться начиная с адреса 80h. Для извлечения его из PSP и анализа в дочерней программе должны быть предусмотрены соответствующие программные строки.

В случае запуска программы с клавиатуры параметры командной строки вводятся вслед за именем запускаемой программы вручную; при программной (динамической) активизации дочернего процесса эти параметры определяются в родительской программе, как это показано в примере.

В блоке параметров хвост команды должен иметь тот же формат, что и в PSP, т.е. начинаться с байта - счетчика, за которым следует символьная строка параметров. Завершается строка символом возврата каретки (0Dh), который не входит в счет байтов. Если в конкретном вызове дочерней программы параметры ей не передаются, хвост команды должен иметь вид

```

db      0,0Dh

```

Можно поступить еще проще, указав в блоке параметров ноль вместо адреса хвоста команды.

Блоки управления файлами (FCB), как уже отмечалось, используются в настоящее время редко, так как операции над файлами удобнее выполнять с помощью дескрипторов. Однако формат блока параметров функции Exec требует указания адресов двух FCB и, соответственно, наличия в родительской программе самих FCB размером 37 байтов каждый. Если, однако, заведомо известно, что загружаемая программа не будет использовать FCB, место под эти блоки можно не выделять. В поля блока параметров, предназначенные для адресов двух FCB, в этом случае записываются нули.

В простейшем варианте (наследуется окружение родителя, а хвост команды и FCB отсутствуют) структуры данных для вызова функции Exec приобретают следующий вид:

```

segment Bk      dw      7 dup (0)
cbname         db      "CHILD.EXE", 0

```

Перед активизацией дочернего процесса родительская программа должна освободить достаточный для загрузки дочерней программы объем памяти. При нехватке памяти или при отсутствии запрашиваемого файла функция Exec возвращает установленный бит CF.

Все файлы и устройства, открытые родительской программой с помощью выделения дескрипторов, дублируются в дочерней программе, т.е. дочерняя программа наследует все открытые дескрипторы родительской. Таким образом, обе программы могут совместно использовать одни и те же файлы, при этом операции ввода - вывода, выполняемые в дочерней программе, отражаются на состоянии дескрипторов родительской.

Активизировав дочерний процесс с помощью функции Exec, DOS приостанавливает выполнение родительской программы до завершения дочерней. Она может, в свою очередь, загружать

другие программы с помощью той же самой процедуры, передавая управление через много уровней, пока в системе не исчерпается память.

DOS позволяет дочернему процессу передать в вызвавший его родительский процесс код возврата (завершения). Этот код может принимать значение от 0 до 255. Обычно 0 считается кодом успешного завершения, а все остальные значения говорят об ошибках. Значение кода возврата формируется в дочерней программе по мере ее выполнения и передается DOS с помощью функции 4Ch, которой обычно завершается любая, в том числе и дочерняя программа, чтобы передать управление родительскому процессу:

```
mov     AH,4Ch      ;Функция завершения процесса
mov     AL,errcode  ;Код завершения
int     21h         ;Передача управления родительскому процессу
```

DOS сохраняет полученный ею код возврата завершенного процесса в одной из системных таблиц до тех пор, пока его не затребует родительская программа. После этого код возврата сбрасывается.

Если программа запускается с клавиатуры (командой оператора), родительским процессом является COMMAND.COM, который и принимает код завершения (и, впрочем, ничего с ним не делает). Для анализа кода завершения программу следует запускать не с клавиатуры, а из командного файла, содержащего строки анализа системной переменной ERRORLEVEL:

```
if errorlevel 3 goto error3
if errorlevel 2 goto error2
if errorlevel 1 goto error1
rem Продолжение командного файла при отсутствии ошибок
```

```
...
:error1
rem Действия при получении код возврата 1
:error2
rem Действия при получении код возврата 2
:error3
rem Действия при получении код возврата 3
и т.д.
```

Если же программа запускается динамически из другой прикладной программы, то для получения кода завершения родительская программа сразу же после возврата в нее управления из дочерней должна выполнить функцию DOS 4Dh, извлекающую код завершения из соответствующей системной ячейки и передающей его вызывающей программе:

Родительская программа  
Поля данных  
parmblk dw

```
7 dup (0)
```

сбросит  
программные строки  
освободит память за реальными пределами программы

```
'8:\info_01.exe'.0
```

;Блок параметров. Вызов дочерней программы без передачи окружения или хвоста команд  
;Спецификация дочерней программы

Подготовим вызов дочерней программы

```
mov     AH,4Ch
```

```
mov     AL,0
```

;Функция Ehex  
;Режим активизации дочерней

программы

```
mov     AX,seg parmblk
```

```
mov     ES,AX
```

```
mov     BX,offset parmblk
```

```
mov     DX,offset cname
```

;Настроим ES на сегментный адрес блока параметров  
;BX=смещение блока параметров  
;DX=смещение имени дочерней программы

```
int     21h
```

;Собственно активизация дочернего процесса

Получим из завершенной дочерней программы код завершения

```
mov     AH,4Dh
```

```
int     21h
```

;Функция получения кода завершения  
;Обращение к DOS за кодом завершения

Проанализируем полученный код завершения

```
cmp     AL,0
```

```
je       notm
```

```
cmp     AL,1
```

```
je       err_1
```

```
cmp     AL,2
```

```
je       err_2
```

;Вернулось значение 0?  
;Да, нормальное завершение  
;Вернулось значение  
;Да, на обработку ошибки 1  
;Вернулось значение 2?  
;Да, на обработку ошибки 2

Дочерняя программа  
Поля данных  
errcode db 0  
программные строки

;Байт для кода завершения

При выполнении программы обнаружилась ошибка с кодом 1

```
or       errcode,1
```

;Установим бит 0 в коде возврата

При выполнении программы обнаружилась ошибка с кодом 2

```
or       errcode,2
```

;Установим бит 1 в коде возврата

Завершение дочерней программы обычным образом

```
mov     AH,4Ch
```

```
mov     AL,errcode
```

```
int     21h
```

;Функция завершения  
;Код завершения пойдет в DOS  
;Фактический возврат в  
;родительский процесс

Изложенные выше приемы передачи параметров в дочернюю программу через окружение или параметры командой строки и возврата кода завершения удобны тем, что они входят в число



стандартных средств DOS. Однако таким образом можно передать лишь ограниченный объем информации, причем преимущественно символической (имена файлов, ключи). Между тем, при разработке сложных программных комплексов возникает необходимость организовать более тесное взаимодействие родительских и дочерних процессов с передачей адресов массивов и других данных, с тем, чтобы открыть доступ дочерней программе к данным родительской. DOS таких средств не предоставляет, сами же иерархические программы, в сущности, не имеют никакой информации друг о друге.

Для того, чтобы передать дочерней программе адреса своих областей данных, родительская программа может воспользоваться двумя областями памяти, доступными в равной степени всем программам:

- свободными векторами прерываний;
- областью межзадачных связей.

В таблице векторов прерываний есть свободные участки. В частности, векторы 60h...66h отведены для прерываний пользователя. Это значит, что программы DOS или BIOS не обращаются к этим векторам, и прикладные программы могут использоваться по своему усмотрению. Конечно, для того, чтобы родительская и дочерняя программы могли обмениваться информацией через свободные векторы, для них должен быть оговорен, как говорят, межпрограммный интерфейс - обе программы должны знать, в каких векторах и что именно находится. В принципе через векторы можно передавать сами данные, однако значительно эффективнее использовать векторы для передачи адресов групп данных или массивов. Поскольку каждый вектор представляет собой четырехбайтовую ячейку, в нем можно записать полный адрес (сегмент и смещение) любого данного и, тем самым, обеспечить возможность для дочерней программы непосредственно работать с полями данных родительской.

Методика использования вектора пользователя для передачи информации в дочернюю программу проста:

```

;Родительская программа
;Поля данных в сегменте данных
array1 db 1024 dup (0)
;Программные строки
mov AH,25 ;Функция заполнения вектора
;прерываний
mov AL,60 ;Вектор пользователя
mov DX,offset array1 ;Полный адрес массива из DS:DX
int 21h ;засылается в вектор прерываний
;Вызов дочерней программы с помощью функции DOS 4Bh Ehex

```

```

;Дочерняя программа
mov AH,35h ;Функция получения вектора
;прерываний
mov AL,60h ;Номер вектора пользователя
int 21h
;DOS вернула содержимое вектора в регистрах ES:BX
;Дочерняя программа может непосредственно обращаться
;к полям данных родительской
mov AX,ES:[BX] ;Чтение первого слова массива
mov DX,ES:[BX+2] ;array1
;Чтение второго слова массива
;array1

```

Помимо свободных векторов, для передачи информации между программами можно пользоваться областью межзадачных связей, которая располагается в самом конце области данных BIOS по адресам 40h:F0h...40h:FFh (всего 4 двойных слова). Эта область, как и свободные векторы, не используется системой и пользователь может записывать в нее свои данные. Процедура использования области межзадачных связей для передачи информации в дочернюю программу принципиально не отличается от использования для этих целей свободных векторов:

```

;Родительская программа
;Поля данных в сегменте данных
array2 db 10000 dup (0)
;Программные строки
mov AX,40h
mov ES,AX ;Настроим ES на начало
;области данных BIOS
mov word ptr ES:0F0h,offset array2 ;Смещение
;массива array2
mov word ptr ES:0F2h,seg array2 ;Сегмент массива array2
;Вызов дочерней программы с помощью функции DOS Ehex
...

```

```

;Дочерняя программа
mov AX,40h ;Настроим ES на начало
mov ES,AX ;области данных BIOS
les BX,dword ptr ES:0F0h
;Дочерняя программа может непосредственно обращаться
;к полям данных родительской
mov AL,ES:[BX] ;Чтение первого байта массива
;array1
mov AH,ES:[BX+1] ;Чтение второго байта массива
;array1

```

Частным, но практически важным случаем активизации дочернего процесса является вызов из родительской программы второй копии командного процессора COMMAND.COM. Это дает возможность выполнять, не прерывая текущей программы, любые команды DOS, например, копирования файлов, проверки

дискеты и т.д. Такая возможность предусматривается сегодня практически во всех прикладных программах (интегрированных средах программирования, электронных таблицах и базах данных, текстовых процессорах и т.д.).

Для вызова с помощью функции Eхес командного процессора следует указать полную спецификацию его файла, которая обычно извлекается из окружения текущего процесса, хотя может быть задана непосредственно в полях данных программы.

При передаче командной строки, оно должно начинаться как параметр командной строки, оно должно начинаться с ключа /C (как это делается при вызове команды DOS COMMAND, служащей для той же цели). Хвост команды может в этом случае иметь следующий вид:

```
cmdtail db 10, " /C DIR A:", 00h
```

Здесь после загрузки второй копии командного процессора автоматически выполняется команда DOS DIR A:. Таким же образом можно выполнить и любую другую команду DOS или прикладную программу.

Можно выделить два варианта процедуры активизации второй копии командного процессора:

1. Командный процессор вызывается с ключом /C и указанием конкретной программы (команды DOS). Получив управление, командный процессор активизирует затребованную программу. После ее отработки управление немедленно передается родительскому процессу. Таким способом можно запускать не только команды DOS, но и командные файлы, которые, как уже отмечалось, нельзя активизировать с помощью функции Eхес непосредственно.

2. Командный процессор вызывается без ключа C и без имени конкретной команды. В этом случае командный процессор, получив управление, ожидает ввода команд DOS с клавиатуры. Пользователь может неограниченное время работать с DOS, вызывая любые команды DOS или прикладные программы. Для возврата в родительский процесс следует ввести команду EXIT. Такой режим (временный выход в DOS) широко используется в большинстве прикладных программ.

### 8.3. Задачи по управлению процессами

**Задача 8.1.** Динамическое выделение памяти активному процессу. Загрузив программу, освободить всю лишнюю память. После этого затребовать у DOS динамическое выделение блока памяти объемом 64 Кбайт. Заполнить выделенный блок некоторой символической информацией, сбросить ее в файл. После завершения программы проанализировать содержимое файла.

Основные фрагменты программы

```
org 0
mov ax, data
mov ds, ax
;Программа .EXE занимает всю память. Освободим ее
mov ax, 0
mov dx, es
mov ax, dx
sub bx, ax
mov ah, 4ah
int 21h
;Теперь выделим 64 Кбайт
mov ax, 40h
mov bx, 1000h
mov int 21h
mov al, 0
;Заполним выделенный блок памяти каким-либо символом
mov es, ax
mov di, 0
xor cx, 0ffffh
mov al, '0'
cld
stosb
;Создадим файл с помощью функции 3Ch и сохраним
;выделенный системой дескриптор в ячейке handle
;Запишем содержимое заполненного блока в файл. Поскольку за один раз
;можно вывести в файл не более 64К-1 байтов, запишем два раза по 32 Кбайт
;Сначала первую половину массива
mov ah, 40h
mov bx, handle
mov cx, 32768
push ds
mov ds, allocseg
mov dx, dx
xor cx, 0
int 21h
;Затем вторую половину массива
pop ds
mov ah, 40h
mov bx, handle
mov cx, 32768
push ds
mov ds, allocseg
mov dx, 32768
mov int 21h
pop ds
;Освободим выделенный блок памяти. ES уже указывает на него
mov ah, 49h
int 21h
;Файл данных
handle dw 0
```

;Сегментный адрес конца программы  
;Сегментный адрес начала программы  
;Размер программы в параграфах  
;Отправим его в BX  
;Функция изменения размера блока

;Функция выведения памяти  
;1000h параграфов = 64 Кбайт

;Адрес выделенного блока  
;Настроим на него ES  
;ES:DI -> начало выделенного блока  
;65535 - счетчик байтов  
;Заполняющий символ  
;Заполнять вперед  
;Заполнение  
;Заполним последний байт массива  
;для контроля

;Функция записи  
;Дескриптор открытого файла  
;Выведем половину массива  
;Сохраним сегментный адрес  
;Настроим DS на выделенный блок  
;DS:DX -> на выделенный блок

;Восстановим адресность  
;нашего сегмента  
;Настроим DS на выделенный блок  
;Продолжим запись с этого байта

;Дескриптор

alllocseg	dw	0	:Сегментный адрес выделенного
блока			
fname	db	'BIGFILE.DAT', 0	:Имя файла для записи массива
abcd	segment	'endseg'	:Фиктивный сегмент для определения
abcd	ends		:конца программы

**Задача 8.2.** Запуск дочернего процесса. Написать и отладить в автономном режиме (запуск с клавиатуры) программу, предназначенную для использования в качестве дочернего процесса. Предусмотреть в этой программе операцию открытия некоторого файла и возврат (через функцию 4Ch) кода завершения (0 - файл открыт, 1 - файл открыть не удалось). В дальнейшем, запуская эту программу при наличии или отсутствии открываемого файла, легко реализовать оба значения кода завершения.

Написать программу, запускающую дочерний процесс и анализирующую, после его завершения, полученный код возврата. Ввиду сложности отладки такого комплекса, в обе программы следует включить анализ правильности выполнения функций DOS и вывод соответствующих диагностических сообщений.

Отладить программный комплекс в целом. Обратить внимание на то, что и имя вызываемой программы в программе-родителе, и имя открываемого файла в дочерней программе описываются явным и однозначным образом, причем в описании могут входить пути к соответствующим файлам. Подготовить два варианта программного комплекса:

1. Оба имени указаны в текущем каталоге.
2. Спецификации обоих файлов даны полностью, с указанием диска и полного пути к файлам.

Испытать указанные варианты программного комплекса при разном расположении его файлов:

- все три файла находятся в текущем каталоге;
- дочерняя программа и рабочий файл находятся в текущем каталоге, а родительская программа - в другом;
- все три файла находятся в разных каталогах.

Структура дочерней программы CHILD.EXE:

вывод с помощью функции 40h диагностического сообщения  
 "Д\* Дочерний процесс запущен";  
 открытие файла с помощью функции 3Dh;  
 анализ бита CF слова состояния процессора;  
 вывод одного из двух диагностических сообщений, говорящих о результатах открытия файла;

завершение программы (передача управления родительскому процессу) с кодом завершения 0, если CF=0 (файл открыт), и 1, если CF=1 (файл не найден).

Структура родительской программы PARENT.EXE:  
 вывод с помощью функции 40h диагностического сообщения  
 "Р\* Родительский процесс запущен";  
 запуск с помощью функции 4Bh (Exes) дочернего процесса  
 программы CHILD.EXE;  
 анализ кода завершения функции Exes и вывод соответствующего диагностического сообщения, если дочерний процесс не запущен (например, если не освободилась память, необходимая для его загрузки);  
 анализ с помощью функции 4Dh кода завершения дочернего процесса;

в зависимости от значения кода завершения вывод одного из двух диагностических сообщений;

завершение программы с помощью функции 4Ch.

Программа родительского процесса PARENT.EXE для задачи 8.2

:Основные фрагменты программы  
 :Выведем на экран диагностическое сообщение mes1  
 :о запуске родительского процесса

...  
 :Освободим всю лишнюю память (см. задачу 8.1)

...  
 :Запустим дочерний процесс  
 mov AX, DATA ;Настроим ES на  
 mov ES, AX ;сегмент данных  
 mov AH, 4Bh ;Функция Exes  
 mov AL, 0 ;Подфункция запуска программы  
 mov BX, offset parmbk ;Адрес блока параметров  
 mov DX, offset chname ;Адрес имени  
 int 21h ;дочерней программы  
 jc errexec ;Ошибка запуска  
 :Проанализируем код возврата из дочернего процесса  
 mov AH, 4Dh ;Функция получения кода возврата  
 int 21h  
 cmp AL, 1 ;Наш код ошибки?  
 je errchild ;Да

:Выведем на экран сообщение mes2  
 :об успешном завершении дочернего процесса

...  
 :Завершение программы  
 outprog: mov AX, 4C00h ;Функция завершения, код  
 int 21h ;завершения = 0

errchild:  
 :Выведем на экран сообщение mes3 об ошибке при выполнении  
 :дочернего процесса

...  
 jmp outprog  
 errexec:  
 :Выведем на экран сообщение mes4 об ошибке  
 :при запуске дочернего процесса



```

...      outprog
;Имя дочерней программы
;Поля данных
childname db "CHILD.EXE",0
parentbk db 7 dup (0)
mes1 db "Родительский процесс запущен",10,13
mes2 db "Дочерний процесс отработал нормально",10,13
mes3 db "Дочерний процесс завершился с ошибкой",10,13
mes4 db "Дочерний процесс не активизирован",10,13
mes1len equ $-mes1
mes2len equ $-mes2
mes3len equ $-mes3
mes4len equ $-mes4
end      outprog

```

### Программа CHILD.EXE дочернего процесса для задачи 8.2

;Основные фрагменты программы  
 ;Выведем диагностическое сообщение mes1  
 ;об активизации дочернего процесса

```

...      finend
;Сделаем попытку открыть файл
mov     AH,30h
mov     AL,0
mov     DX,offset fname
int     21h
jnc     ok
;Обработка ошибки открытия файла
mov     EAX,1
;Выведем диагностическое сообщение mes2
;Завершим дочерний процесс
;Сначала выведем сообщение mes3 о нормальной работе
ok:

```

```

...      finend
;И завершим программу с передачей родителю кода завершения
finend: mov     AH,4Ch
mov     AL,errcode
int     21h

```

```

;Поля данных
mes1 db "Дочерний процесс запущен",10,13
mes1len equ $-mes1
mes2 db "Файл не открылся",10,13
mes2len equ $-mes2
mes3 db "Файл открылся и child завершается",10,13
mes3len equ $-mes3
fname db "FILE.TXT",0
errcode db 0

```

**Задача 8.3.** Передача параметров с помощью параметров командной строки. Видоизменить программу задачи 8.2 с целью демонстрации передачи параметров от родительского процесса открываемого файла. В качестве параметра использовать имя области PSP, содержащей хвост команды и извлечение оттуда имени открываемого файла. Отладить программу CHILD.EXE в автономном режиме (запуск с клавиатуры). Строка запуска программы CHILD.EXE (запуск с клавиатуры) (если требуется открыть файл FILE.TXT):

В программе PARENT.EXE вместо пустого хвоста команды указать конкретное имя открываемого файла. Отладить программный комплекс в целом.

Программа CHILD.EXE дочернего процесса для задачи 8.3  
 ;Основные фрагменты программы  
 ;Вывод диагностического сообщения mes1  
 ;о запуске дочернего процесса

```

...      ok
;Инициализируем сегментные регистры ES и DS
push    ES
push    DS
pop     ES
pop     DS
;Заберем хвост команды из PSP в fname
mov     SI,80h
mov     CL,[SI]
dec     CL
xor     CH,CH
inc     SI
inc     SI
mov     DI,offset ES:fname
movsb
push    ES
push    DS
;Сделаем попытку открыть файл с анализом флага CF

```

```

...      ok
;Обработка ошибки открытия файла (засылка кода завершения
;в ячейку errcode и вывод сообщения mes2 об ошибке открытия файла)

```

```

...      ok
;Завершим дочерний процесс
;Сначала выведем сообщение mes3 о нормальной работе
ok:

```

```

;И завершим дочерний процесс с передачей родителю кода завершения
outprog:

```

```

mes1 db "Дочерний процесс запущен",10,13

```

```

mes2 db '*А* файл не открылся', 10, 13
mes3 db '*А* файл открылся и child завершается', 10, 13
fname db 80 dup (0) ; Начальное значение кода возврата
errcode db 0

```

Задача 8.4. Извлечение переменной окружения в дочерней программе через адрес окружения в PSP. Видоизменить программу задачи 8.2 с целью демонстрации передачи параметра запускаемой программе через ее окружение. В качестве параметра по-прежнему использовать имя открываемого файла, определяемого теперь не с помощью параметра командной строки, а с помощью окружения, как значение переменной окружения WORKFILE. В этом случае, изменяя в окружении с помощью команды DOS SET значение переменной WORKFILE, можно заставить программу CHILD.EXE работать именно с этим файлом в качестве рабочего. Значение переменной WORKFILE устанавливается следующим образом:

```
>SET WORKFILE=FILE.TXT
```

если файл находится в текущем каталоге текущего диска или

```
>SET WORKFILE=F:\MYDIR\FILE.TXT
```

если файл находится в каталоге MYDIR на диске F:.

После отладки программы CHILD.EXE в автономном режиме испытать работу иерархического комплекса PARENT.EXE - CHILD.EXE, имея в виду, что программа PARENT.EXE в процессе загрузки и запуска ее процессором COMMAND.COM получает от него копию системного окружения и, в свою очередь, передает ее дочернему процессу, в данном случае программе CHILD.EXE.

Проверить следующие ситуации:

- в команде SET указано неверное обозначение переменной окружения;

- в команде SET указано неверное имя файла, либо указанный файл отсутствует.

Программа CHILD.EXE дочернего процесса для задачи 8.4

Основные фрагменты программы  
Выведем диагностическое сообщение mes1  
при запуске дочернего процесса

```

...
; Получим адрес окружения
mov     AX, ES:2Ch      ; Сегментный адрес окружения в PSP
mov     ES, AX          ; ES -> окружение
; Найдем в окружении подстроку 'WORKFILE='
; Сравним переменную в окружении с нашей
mov     BX, 0           ; BX -> начало имени
cld                     ; Поиск вперед
comp    mov     CX, len  ; Длина нашей переменной

```

```

mov     SI, offset envvar ; Её адрес
mov     DI, BX            ; ES:DI -> начало строки
cmpsb   ; Сравнение
je       gotit            ; Нашли нашу переменную
; Эта переменная в окружении не совпадает с нашей. Будем искать
; следующую. Сначала найдем 0, т.е. конец строки вида
; переменная=значение
cstnul: je       gotit    ; Это 0?
        inc      BX       ; Нашли 0
        jmp      tstnul   ; Это не 0, на следующий байт
; Посмотрим, нет ли после этого 0 второго 0 - конца
; всего окружения
cstnul: inc      BX       ; К следующему байту
        cmp      byte ptr ES:[BX], 0 ; Это 0?
        jne      compr   ; Нет, на сравнение следующей
                                ; переменной
; Нашли второй 0 - конец окружения
; Выведем сообщения mes4 об отсутствии в окружении переменной WORKFILE.
; Можно также установить соответствующий код завершения в переменной
errcode ...

```

gotit: Извлечем значение нашей переменной (т.е. имя файла) из окружения.  
DI -> первый байт значения переменной. Сначала определим длину значения переменной (оно заканчивается 0)

```

mov     BX, -1           ; Счетчик длины, его
notend: inc      BX       ; начальное значение = 0
        cmp      byte ptr ES:[DI], 0 ; Еще не 0?
        jne      notend   ; Нет еще, продолжим поиск 0
; Нашли 0. BX = длина значения переменной.
; DI -> по-прежнему первый байт значения переменной.
; ES -> сегмент окружения.
; DS -> наш сегмент данных. Перенесем значение переменной
; (имя файла) в программу

```

```

mov     CX, BX           ; Число байтов в имени файла
push    ES              ; Взаимно поменяем
push    DS              ; содержимое
pop      ES              ; сегментных
pop      DS              ; регистров
mov     SI, DI           ; DS:SI -> имя файла в окружении
mov     DI, offset ES:fname ; ES:DI -> место для имени
                                ; файла в программе
ter     movsb           ; Пересылка подстроки
        push     ES      ; Восстановим адресацию
        pop      DS      ; к нашим данным через DS
; Сделаем попытку открыть файл
...
jnc     ok               ; Переход, если C = 0
; Обработаем ошибку открытия файла как и в предыдущих примерах

```

```

...
jmp     outprog

```

```

;Выведен сообщение mes3 о нормальной работе
ok:
;И завершим дочерний процесс с передачей родительскому
;кода завершения
outproc:

```

```

;Поля данных
mes1 db
mes2 db
mes3 db
mes4 db
fname db
argccode db
envvar db
len equ

;*Д* Дочерний процесс запущен',10,13
;*А* файл не открылся',10,13
;*А* файл открылся и child завершается',10,13
;*А* Переменная WORKFILE
'не найдена в окружении',10,13
80 dup (0)
0
'WORKFILE='
$-envvar

```

Задача 8.5. Активизация второй копии командного процессора. Воспользовавшись фрагментами задач 8.3 и 8.4, написать программу, активизирующую с помощью функции Ehex вторую копию командного процессора.

Структура программы (родительского процесса):

освобождение лишней памяти;  
извлечение из окружения спецификации командного процессора, как значения переменной окружения COMSPEC. Соответствующий фрагмент (для переменной WORKFILE) имеется в дочерней программе задачи 8.4. Адрес поля fname, куда переносится спецификация командного процессора, будет использован при вызове функции Ehex (вместо поля cname);  
запуск командного процессора с помощью функции Ehex;  
вывод сообщения о возврате управления из второй копии командного процессора и завершении родительского процесса.

Отладить и испытать два варианта программы, соответствующие двум вариантам запуска командного процессора (без хвоста команды, а также с ключом /C и именем какой-либо команды DOS). Испытания программы удобно осуществлять под управлением оболочки Norton Commander. В этом случае, пока вызванный командный процессор не завершит свою работу, экран будет оставаться черным (так как продолжается выполнение запущенной программы). Возврат из вторичного командного процессора в программу и ее завершение приведут к появлению на экране обычного информационного кадра Norton Commander.

```

;Основные фрагменты программы
;Освобождение лишней памяти

```

```

;Получение адреса окружения из PSP

```

```

;Поиск в окружении переменной COMSPEC и перенос ее значения
;в поле fname

```

```

;Запуск дочернего процесса, в данном случае вторичного
;командного процессора COMMAND.COM

```

```

;Вывод сообщения о завершении родительского процесса

```

```

;Завершение программы

```

```

;Основные поля данных
orgtblk dw 0

```

```

;Дочернему процессу передается
;родительское окружение
;Адрес хвоста команды

```

```

; 1. Вариант пустого хвоста
cmdtail db 0,13
; 2. Вариант хвоста с именем системной программы
cmdtail db 7,' /c dir',13
cmdtail db 'COMSPEC='
envvar db $-envvar
len equ 80 dup (0)
fname db

```



## 9. Обработка прерываний

### 9.1. Контроллер прерываний и его программирование

Сигналы внешних аппаратных прерываний, возникающие в устройствах, входящих в состав компьютера (таймер, клавиатура, диски и проч.), поступают в процессор не непосредственно, а через контроллер прерываний, в качестве которого используется микросхема Intel 8259A. Обработка аппаратного прерывания обязательно включает в себя процедуры управления контроллером прерываний. При этом методика программирования контроллера прерываний определяется не только его внутренним устройством и возможностями, но и местом в архитектуре компьютера. Начнем изучение этих вопросов с рассмотрения организации системы аппаратных прерываний машин типа IBM PC/XT (рис. 9.1).



Рис. 9.1. Организация аппаратных прерываний в машинах IBM PC/XT.

К восьми входным выводам контроллера прерываний  $IRQ1...IRQ7$  (interrupt request, запрос прерывания) подключаются выходы устройств, на которых возникают сигналы прерываний. Выход INT контроллера подключается к одноименному входу микропроцессора. Таким образом, основное назначение контроллера прерываний - направление сигналов запросов прерываний от восьми устройств на единственный вход прерываний микропроцессора. При этом, кроме сигнала INT, инициирующей процедуру прерывания в микропроцессоре, контроллер

передает в микропроцессор по линиям данных номер вектора, через который должна быть вызвана программа обработки поступившего прерывания. Передаваемый номер вектора образуется в контроллере прерываний путем сложения базового номера, записанного в одном из его регистров, с номером входной линии, по которой поступил запрос прерывания. Номер базового вектора заносится в контроллер автоматически в процессе начальной загрузки компьютера. Контроллер программируется через порты  $20h$  и  $21h$ .

Поскольку базовый вектор всегда равен 8, номера векторов, закрепленных за аппаратными прерываниями, лежат в диапазоне  $8h...Fh$ . Очевидно, что номера векторов аппаратных прерываний однозначно связаны с номерами линий, или уровнями  $IRQ$ , а через них - с конкретными устройствами компьютера. В таблице 9.1 приведен перечень аппаратных векторов и закрепленных за ними устройств.

Таблица 9.1. Соответствие векторов прерываний устройствам компьютера для машин типа IBM PC/XT.

Уровень прерывания	Вектор прерывания	Устройство
$IRQ0$	$08h$	Таймер
$IRQ1$	$09h$	Клавиатура
$IRQ2$	$0Ah$	Резерв для подключения нестандартных устройств
$IRQ3$	$0Bh$	Последовательный порт COM2
$IRQ4$	$0Ch$	Последовательный порт COM1
$IRQ5$	$0Dh$	Жесткий диск
$IRQ6$	$0Eh$	Гибкий диск
$IRQ7$	$0Fh$	Принтер LPT1

Компьютеры IBM PC/AT комплектуются большим количеством устройств, способных возбуждать сигналы прерываний. Однако каждый контроллер прерываний может воспринять только восемь таких сигналов. Для обслуживания большего количества устройств контроллеры можно объединять, образуя из них всеобразную структуру (в пределе можно к каждому из входов объединяющего, ведущего контроллера подсоединить свой ведомый и в итоге получить систему с 64 входами запросов прерываний). В машинах PC/AT устанавливают два контроллера (рис. 9.2), увеличивая тем самым возможное число для входных устройств до 15 (7 у ведущего и 8 у ведомого контроллеров).

Выход INT ведомого контроллера подсоединяется к входу  $IRQ2$  ведущего, а выход INT ведущего - как и раньше, к входу INT микропроцессора. Базовый вектор и порты для

ведущего контроллера остаются прежними; ведомому контроллеру назначаются порты A0h и A1h и базовый вектор 70h. Таким образом, векторы аппаратных прерываний в компьютерах AT лежат в диапазонах 8h...Fh и 70h...77h. В таблице 9.2 приведен перечень аппаратных векторов и закрепленных за ними устройств.



Рис. 9.2. Организация аппаратных прерываний в машинах IBM PC/AT.

Таблица 9.2. Соответствие векторов прерываний устройствам компьютера для машин типа IBM PC/AT.

Уровень прерывания	Вектор прерывания	Устройство
IRQ0	08h	Таймер
IRQ1	09h	Клавиатура
IRQ2	0Ah	Вход от ведомого КМОП-микросхема
IRQ8	70h	Перенаправлено на Int 0Ah
IRQ9	71h	Зарезервировано
IRQ10	72h	Зарезервировано
IRQ11	73h	Мышь (PS/2)
IRQ12	74h	Исключение сопроцессора
IRQ13	75h	Жесткий диск
IRQ14	76h	Зарезервировано
IRQ15	77h	Зарезервировано
IRQ3	0Bh	Последовательный порт COM2
IRQ4	0Ch	Последовательный порт COM1
IRQ5	0Dh	Принтер LPT2
IRQ6	0Eh	Гибкий диск
IRQ7	0Fh	Принтер LPT1

Обратимся теперь к внутренней структуре контроллера. Логически в ней можно выделить четыре основных узла: регистр входных запросов, регистр маски, схему приоритетов и регистр обслуживаемых запросов (рис. 9.3). Все узлы контроллера восьмибитовые, по одному биту на каждый входной сигнал.

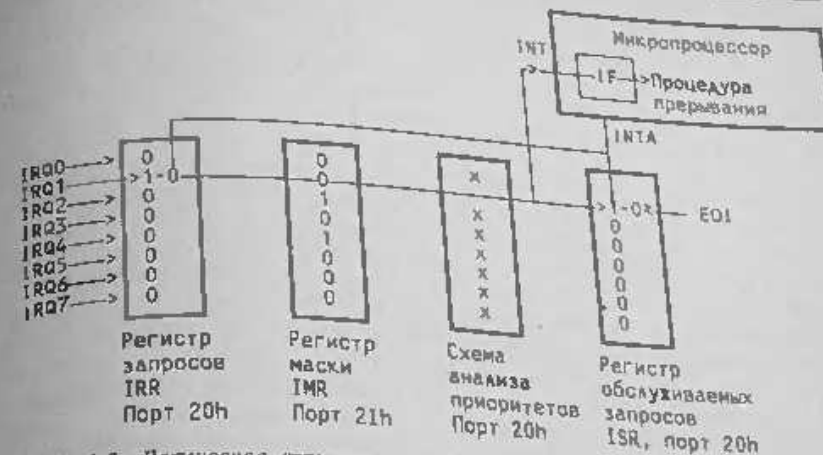


Рис. 9.3. Логическая структура контроллера прерываний.

Сигнал запроса прерывания IRQ от устройства (на рис. 9.3 IRQ1, т.е. сигнал от клавиатуры) поступает на вход регистра запросов и устанавливает в 1 соответствующий бит этого регистра. Далее на пути сигнала стоит регистр маски, программируемое значение сигнала, значение 0 в бите маски разрешает прохождение сигнала, значение 1 - запрещает. Пройдя через маску, сигнал поступает на схему анализа приоритетов. При стандартной настройке схемы анализа приоритетов (она программируется посылкой определенных команд через порт 20h) приоритеты сигналов IRQ снижаются по мере роста номера сигнала, т.е. максимальным приоритетом обладает сигнал IRQ0, минимальным - сигнал IRQ7. В компьютерах AT, где ведомый контроллер подключается к входу IRQ2 ведущего, все приоритеты ведомого контроллера располагаются между приоритетами уровней IRQ1 и IRQ3 ведущего, т.е. образуется такая цепочка приоритетов:

IRQ0 - IRQ1 ppp IRQ8 ... IRQ15 ppp IRQ3 ... IRQ7  
Наивысший приоритет ведомого Низший приоритет

Приоритеты уровней прерываний не имеют никакого значения, пока прерывания поступают редко и не накладываются друг на друга. Вопрос о приоритетах становится важным только в том случае, если очередной сигнал прерывания приходит в тот момент, когда еще не закончено выполнение программы обработки предыдущего прерывания. Алгоритм таких

наложенных прерываний определяет программист путем соответствующего построения обработчика прерывания; этот вопрос будет подробнее рассмотрен ниже.

Пройдя через схему анализа приоритетов, сигнал запроса прерывания поступает на вход регистра обслуживаемых запросов и дает разрешение на установку в 1 его бита (однако не устанавливает его). Одновременно сигнал поступает на вход INT микропроцессора. Микропроцессор регистрирует поступление сигнала INT лишь в том случае, если установлен флаг разрешения прерываний IF в регистре флагов. Таким образом, разрешения прерываний `sti` запрещает все аппаратные прерывания. Микропроцессор, получив сигнал INT, отзывается на него выходным сигналом `INTA`, который поступает в контроллер прерываний и выполняет там два действия. Во-первых, он устанавливает бит регистра обслуживаемых запросов, разрешенный бит регистра запросов. Таким образом, поступивший запрос, для которого началась процедура обслуживания его микропроцессором, переводится в разряд обслуживаемых. Начиная с этого момента на тот же вход контроллера прерываний может придти следующий сигнал прерывания от устройства. Он, правда, какое-то время не будет обслуживаться, но, по крайней мере, не пропадет, а запомнится в регистре запросов и будет ждать своей очереди на обслуживание контроллером и процессором.

Микропроцессор одновременно с посылкой в контроллер прерываний сигнала `INTA` сбрасывает флаг IF в регистре флагов, запрещая все аппаратные прерывания. Прерывания останутся запрещенными до выполнения пользователем команды `sti`, или до установки флага IF каким-либо другим способом.

Установка 1 в бите регистра обслуживаемых запросов воздействует на схему анализа приоритетов. Установленный бит блокирует в схеме анализа приоритетов все уровни прерываний, начиная с текущего и ниже. Таким образом, если не принять специальных мер, даже после завершения обработчика прерывания все прерывания данного и более низких приоритетов останутся заблокированными. Сброс бита регистра обслуживаемых запросов осуществляется засылкой кода `20h` в порт `20h` для ведущего контроллера и в порт `A0h` для ведомого. Этот код получил название команды, или приказа `EOI` (End Of Interrupt, конец прерывания). Приказ конца прерывания должен возбуждаться в любом обработчике прерываний.

Исходя из изложенного, в программе обработки прерывания можно выделить три участка, которые показаны на рис. 9.4 для конкретного случая прерывания уровня 1.

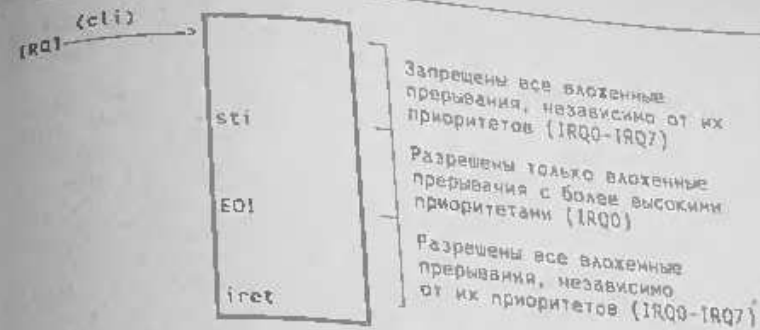


Рис. 9.4. Обобщенная структура программы обработки прерывания.

Поскольку процессор, получив сигнал INT, сбрасывает флаг IF, при входе в наш обработчик все прерывания оказываются запрещенными и программа не может быть прервана внешними сигналами. Команда `sti`, выполненная в программе, устанавливает флаг IF и разрешает прохождение запросов прерываний в процессор. Однако все уровни прерываний, начиная с текущего, остаются заблокированными в контроллере. В результате работа обработчика может быть прервана только при поступлении запроса прерывания более высокого приоритета (в рассматриваемом примере `IRQ0`). Такая ситуация называется вложенным прерыванием.

Выполнение в обработчике команд, реализующих приказ конца прерывания `EOI` снимает блокировку в контроллере, и начиная с этого момента запрос прерывания любого уровня прервет выполнение обработчика. Особенно неприятной может оказаться ситуация, когда обработчик прерывается сигналом запроса прерывания того же уровня. Это приводит к тому, что программа обработчика, не дойдя до конца, опять начинает выполняться с самого начала, т.е. происходит повторный вход в программу. Для того, чтобы такое явление не нарушило работоспособность системы, обработчик прерывания должен быть написан по определенным правилам, обеспечивающим его ресеттерабельность.

Практически структуру программы обработки прерывания выбирают исходя из конкретных условий. Часто в самом начале программы выполняют команду `sti`, чтобы не задерживать обработку прерываний от более приоритетных устройств (в частности, таймера). Приказ конца прерывания `EOI` посылается в контроллер в самом конце программы, перед завершающей командой `iret` с тем, чтобы полностью исключить вложенные прерывания от запросов того же уровня (рис. 9.5,а). Однако



сигнал прерывания того же (или более низкого) уровня может прийти между командой `out 20h AL` и командой `iret`. Поскольку блокировка нижележащих уровней в контроллере уже снята, возникнет вложенное прерывание и повторный вход в ту же программу. Чтобы избежать этого, перед командой `EOI` выполняются команды запрета всех прерываний `cli`. В результате вложенные прерывания запрещаются до выхода из обработчика. Можно подумать, что прерывания останутся запрещенными навсегда, однако это не так. Команда `iret` восстанавливает не только адрес возврата в регистрах `CS` и `IP`, но и содержимое регистра флагов на момент прерывания. Если при этом содержимом регистра флагов возникло прерывание, значит, флаг `IF` был установлен. Получается, что команда `iret` в программе обработки аппаратного прерывания всегда восстанавливает установленное состояние флага `IF`, т.е. разрешает прерывания.

Между прочим, отсюда следует, что в обработчике прерываний вообще может отсутствовать команда `sti`. В этом случае программа обработчика будет выполняться при запрещенных прерываниях, а разрешены прерывания будут после выполнения завершающей команды `iret` (рис. 9.5, 6).

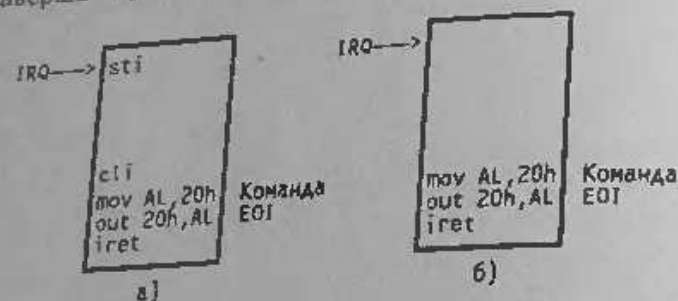


Рис. 9.5. Типичные структуры программы обработки прерываний: при разрешенных вложенных прерываниях (а) и при запрещенных вложенных прерываниях (б).

Запросы на прерывания, поступающие в ведущий контроллер (уровни `IRQ0...IRQ7`) блокируют только ведущий контроллер. Однако запросы на прерывания, поступающие в ведомый контроллер (уровни `IRQ8...IRQ15`) блокируют не только уровни низших приоритетов в ведомом контроллере, но и уровни `IRQ2...IRQ7` в ведущем контроллере. Поэтому в обработчиках прерываний уровней 8...15 следует предусматривать посылку команды конца прерываний в оба контроллера:

```
mov AL, 20h
out 20h, AL
out A0h, AL
```

Обычно программирование контроллера прерываний в прикладных программах сводится к посылке сигнала `EOI` и маскированию, в случае необходимости, отдельных уровней прерываний. Однако в некоторых случаях приходится выполнять более серьезное вмешательство в настройку контроллера. Например, перед переходом в защищенный режим следует изменить базовые векторы обоих контроллеров.

Для смены базового вектора требуется выполнить полностью процедуру инициализации контроллера, которая состоит из ряда так называемых команд инициализации (СКИ), посылаемых в строгой последовательности друг за другом. Формат первого слова инициализации СКИ1, посылаемого для ведущего контроллера в порт `20h`, а для ведомого - в порт `A0h`, представлен на рис. 9.6.



Рис. 9.6. Слово команд инициализации СКИ1.

Второе слово инициализации СКИ2, посылаемое во второй порт контроллера (для ведущего контроллера - `21h`, для ведомого - `A1h`) задает базовый вектор. Как уже отмечалось, при стандартной конфигурации компьютера базовый вектор ведущего контроллера равен 8, ведомого - `70h`.

Третье слово инициализации СКИЗ выглядит по-разному для ведущего и ведомого контроллеров. Для ведущего в слове СКИЗ устанавливаются в 1 те биты, которые соответствуют входам `IRQ`, подключенным к ведомым контроллерам. Биты, соответствующие входам `IRQ`, подключенным к периферийным устройствам, сбрасываются. Для всех компьютеров типа `PC/AT` ведомый контроллер подсоединяется ко входу 2 ведущего, поэтому в СКИЗ должен быть установлен бит 2 и сброшены остальные биты, что соответствует числу 4. СКИЗ посылается (для ведущего контроллера) в порт `21h`.

Для ведомого контроллера в слове инициализации СКИЗ указывается номер входа `IRQ` ведущего контроллера, к которому подключен данный ведомый (которых, как уже отмечалось, в принципе может быть несколько). Для компьютеров типа `PC/AT` ведомый контроллер подсоединяется ко входу 2 ведущего, поэтому слово СКИЗ равно 2.

Формат четвертого слова инициализации СКИ4 представлен на рис. 9.7.



Рис. 9.7. Слово команд инициализации СКИ4.

Обычно (МП 80x86, требуется сигнал EOI) слово СКИ4 равно 1.

Таким образом, для инициализации обоих контроллеров прерываний (для стандартной конфигурации компьютера) следует выполнить такие последовательности команд:

```

;Инициализация ведущего контроллера прерываний
mov     DX,20h           ;Порт контроллера
mov     AL,11h           ;СКИ1: будет СКИ3
out     DX,AL
jmp     $+2               ;Задержка
inc     DX                ;Второй порт контроллера
mov     AL,8              ;СКИ2: базовый вектор
out     DX,AL
jmp     $+2               ;Задержка
mov     AL,4              ;СКИ3: ведомый подключен
out     DX,AL             ;к уровню 2

out     DX,AL             ;Задержка
jmp     $+2
mov     AL,1              ;СКИ4: 80x86, требуется EOI
out     DX,AL

```

```

;Инициализация ведомого контроллера прерываний
mov     DX,A0h           ;Порт контроллера
mov     AL,11h           ;СКИ1: будет СКИ3
out     DX,AL
jmp     $+2               ;Задержка
inc     DX                ;Второй порт контроллера
mov     AL,70h           ;СКИ2: базовый вектор
out     DX,AL
jmp     $+2               ;Задержка
mov     AL,2              ;СКИ3: ведомый подключен
out     DX,AL             ;к уровню 2

out     DX,AL             ;Задержка
jmp     $+2

```

```

mov     AL,1
out     DX,AL

```

;СКИ4: 80x86, требуется EOI

После каждой команды отправки кода в порт контроллера предусмотрена небольшая задержка (команда jmp на следующий байт программы), чтобы аппаратура контроллера успела воспринять посылаемую команду. Конкретная величина задержки зависит от соотношения скоростей работы процессора и контроллера; часто оказывается, что в задержке нет необходимости.

Инициализировав оба контроллера, следует установить в них маски прерываний. Конкретное значение масок зависит от аппаратной конфигурации компьютера. Типичными значениями являются B8h для ведущего контроллера и 9Dh для ведомого.

При отладке взаимодействия с компьютером аппаратуры, работающей в режиме прерываний, приходится анализировать состояние внутренних регистров контроллера прерываний. Для этого существуют специальные команды контроллера.

Если в порт 20h контроллера прерываний послать код 0Ah, то разрешается чтение входного регистра контроллера - регистра запросов IRR. Чтение (в том числе неоднократное) осуществляется через порт 20h. Читая содержимое IRR, можно определить, на какие входы контроллера поступают сигналы аппаратуры. При этом надо иметь в виду, что регистрация процессором сигнала прерывания приводит к сбросу запроса соответствующего уровня в регистре запросов. Поэтому наблюдение запросов в регистре IRR следует выполнять либо при замаскированных, либо при запрещенных прерываниях.

Код 0Bh, посланный в тот же порт 20h, разрешает чтение регистра обслуживаемых запросов ISR. Чтение (в том числе неоднократное) осуществляется через порт 20h. Таким образом можно установить, проходит ли сигнал прерывания в процессор (поскольку установка битов регистра ISR выполняется сигналом INTA, поступающим из процессора в контроллер после регистрации им сигнала прерывания).

## 9.2. Задачи на программирование контроллера прерываний.

**Задача 9.1.** Запретить прерывания от таймера. Выполнить программу, проанализировать результат с помощью команды DOS TIME или часов программы Norton Commander.

```

in      AL,21h           ;Прочитаем текущую маску
or      AL,1             ;Установим добавочно бит 0
out     21h,AL           ;Повлём в регистр маски

```

**Задача 9.2.** Разрешить прерывания от таймера.

```

in      AL,21h           ;Прочитаем текущую маску

```

```
and out AL, 0FEh
        20h, AL
```

```
; сбросим выборочно бит 0
; Пошлем в регистр маски
```

Задача 9.3. Изучить процедуру чтения регистра обслуживаемых запросов контроллера прерываний ISR и влияние на его состояние сигнала EO1. Для этого написать программу обработчика аппаратных прерываний от таймера (методика составления обработчиков прерываний будет подробно описана позже). В обработчике предусмотреть чтение состояния регистра обслуживаемых запросов до и после сигнала EO1. Поскольку программа использует минимум данных и почти не работает со стеком, она выполнена без сегментов данных и стека (но, тем не менее, написана в формате .EXE).

```
text segment 'code'
assume cs:text, ds:text
```

```
main proc
; Настроим сегментный регистр DS на сегмент кода, чтобы можно было
; пользоваться обозначениями полей данных без префикса замены сегмента
```

```
push ES
pop DS
```

```
; Сохраним вектор 08h ; функция чтения вектора прерывания
mov AX, 35h ; Номер читаемого вектора
mov AL, 08h
```

```
int 21h
mov word ptr old_08h, BX ; Сохраним системный вектор в
mov word ptr old_08h+2, ES ; двухсловной ячейке old_08h
```

```
; Заполним вектор 08h ; функция заполнения вектора
mov AH, 25h ; Номер заполняемого вектора
mov AL, 08h ; Номер читаемого вектора
mov DX, offset new_08h ; Смещение нашего обработчика
int 21h
```

```
; Остановим программу до нажатия клавиши ; функция ввода символа
mov AH, 01h ; с клавиатуры
```

```
int 21h
; Перед завершением программы восстановим содержимое вектора 08h
lds DX, old_08h ; DS:DX=системный вектор 08h
mov AX, 25h ; функция заполнения вектора
mov AL, 08h ; Номер заполняемого вектора
int 21h
```

```
; Завершим программу обычным образом
```

```
old_08h dd 0 ; двухсловная ячейка для системного
; вектора
```

```
main endp
```

```
; Обработчик прерываний 08h
```

```
new_08h proc
push AX ; Сохраним используемые в
push ES ; обработчике регистры
mov AX, 0B800h ; Настроим ES
mov ES, AX ; на адрес видеобуфера
```

```
mov out AL, 08h
        20h, AL
```

```
jmp $+2
in AL, 20h
add AL, '0'
mov AH, 1Eh
mov ES, 1680, AX
mov AL, 20h
out 20h, AL
jmp $+2
jmp $+2
in AL, 20h
```

```
add AL, '0'
mov AH, 4Eh
mov ES, 1690, AX
pop ES
pop AX
iret
endp
ends
main
```

```
new_08h
text
end
```

```
; Пошлем в контроллер прерывания
; команду 08h - разрешение чтения
; регистра ISR
; Небольшая задержка
; Прочитаем регистр ISR
; Преобразуем в символьную форму
; Произвольный атрибут символа
; Выведем на экран
; Пошлем в ведущий контроллер
; команду EO1
; Небольшая
; задержка
; Снова прочитаем регистр
; обслуживаемых запросов
; Преобразуем в символьную форму
; Атрибут для наглядности другой
; Выведем на экран в другом месте
; Восстановим сохраненные
; регистры
; Возврат в прерывную программу
```

Задача 9.4. Изучить работу регистра запросов контроллера прерываний IRR. Для этого модифицировать программу предыдущего примера следующим образом. В основной программе замаскировать уровень 1 запросов от клавиатуры, чтобы установка бита запроса от клавиатуры в регистре запросов не инициировала процедуру обслуживания прерывания в процессоре и, соответственно, почти немедленного сброса бита запроса. После этого поставить два последовательных запроса к DOS на ввод с клавиатуры. Это даст нам время для наблюдения состояния регистра запросов контроллера прерываний и после первого нажатия клавиши, и после возвращения уровня 1 в размаскированное состояние (если второго запроса нет, программа после первого нажатия сразу же завершится, и мы не сможем наблюдать сброс бита запроса после регистрации этого прерывания контроллером).

В обработчике прерываний подать в контроллер команду разрешения чтения регистра запросов (код 0Ah в порт 20h), после чего прочитать и вывести на экран состояние этого регистра. Эти действия выполнять при каждом прерывании в течение 5...10 с (путем отсчета 90...180 прерываний). Если за это время нажать на какую-либо клавишу, мы увидим на экране установку бита 1 регистра запросов.

По истечении указанного времени переключить программу обработчика прерываний на другую ветвь: по-прежнему выводить на экран состояние регистра запросов, но перед выходом



из прерывания размаскировать уровень 1. Поскольку основная программа ждет второго нажатия клавиши, обработчик прерываний от таймера вместе со всей программой по-прежнему загружен, и мы будем иметь время наблюдать сброс бита запроса, как только после размаскирования этого уровня запрос на прерывание пройдет в процессор и тот пошлет в контроллер сигнал INTA.

```

text    segment    'code'
        assume     cs:text,ds:text
main    proc
;Настроим сегментный регистр DS на сегмент кода, чтобы можно было
;пользоваться обозначениями полей данных без префикса замены сегмента
...
;Сохраним вектор 08h
...
;Заполним вектор 08h
...
;Замаскируем уровень 1 запросов прерываний от клавиатуры,
;чтобы можно было наблюдать установку бита регистра запросов
;при нажатии любой клавиши
        in         AL,21h
        or         AL,2
        out        21h,AL
;Остановим программу
        mov        AH,01h
        int        21h
;Еще раз остановим программу
        mov        AH,01h
        int        21h
;Перед завершением программы восстановим содержимое вектора 08h
...
;Завершим программу обычным образом
old_08h dd        0
main    endp
;Обработчик прерываний 08h
new_08h proc
        push      AX
        push      ES
        mov       AX,0800h
        mov       ES,AX
        mov       AL,0Ah
        out       20h,AL

        jmp       $+2
        in        AL,20h
        add       AL,'0'
        mov       AH,1Eh
        mov       ES:1680,AX
        mov       AL,20h
        out       20h,AL
        dec       CS:count
;Сохраним используемые в
;обработчике регистры
;Настроим ES
;на адрес видеобuffers
;Пошлем в контроллер прерываний
;команду 0Ah - разрешение чтения
;регистра IRR
;Небольшая задержка
;Прочитаем регистр запросов
;Преобразуем в символьную форму
;Произвольный атрибут символа
;Выведем на экран
;Пошлем в ведущий контроллер
;команду EOI
;Отсчитаем count штук прерываний

```

```

cmp     CS:count,0
jg      go_out
in      AL,21h
and     AL,00h
out     21h,AL

go_out: pop     ES
        pop     AX
        iret
new_08h endp
count   dw      160
text    ends
end      main

```

;Пока время не истекло, будем  
;переходить на go\_out  
;Время истекло, будем сбрасывать  
;маску (в действительности ее  
;достаточно сбросить лишь  
;один раз)  
;Восстановим сохраненные  
;регистры  
;Возврат в прерванную программу  
;Счетчик числа прерываний

### 9.3. Взаимодействие прикладных и системных обработчиков прерываний

Программы обработки прерываний, или обработчики прерываний, как их обычно называют, являются важнейшей составной частью большинства программных продуктов. Структура обработчика прерываний и его взаимодействие с остальными компонентами программного комплекса определяются рядом факторов:

- прерывания, инициализирующие обработчик, могут быть аппаратными (от периферийных устройств) или программными (команда INT);
- программа обработчика может быть резидентной или транзитной;
- вектор обрабатываемого прерывания может быть свободным или использоваться системой;
- если вектор уже используется системой, т.е. в составе DOS имеется системный обработчик прерываний данного типа, то новый обработчик может полностью заменять системный или "сцепляться" с ним;
- в случае сцепления с системным обработчиком новый обработчик может выполнять свои функции до системного или после него.

Для того, чтобы прикладной обработчик получал управление в результате прерывания, его адрес следует поместить в соответствующий вектор прерывания. Хотя содержимое вектора прерываний можно изменить простой командой MOV, однако предпочтительнее использовать специально предусмотренную функцию DOS 25h прерывания DOS INT 21h). В регистр AL помещается номер модифицируемого вектора, а в регистры DS:DX - адрес нового обработчика. Старое содержимое вектора следует сохранить в выделенных для этого ячейках и восстановить перед завершением программы (с помощью той же

функции 25h). Для получения исходного содержимого вектора предусмотрена функция DOS 35h. Для ее вызова следует загрузить в АН номер функции - 35h, а в AL - тип вектора; двухсловное содержимое вектора возвращается в регистрах ES:BX.

Структура программы с обработкой прерываний, не используемых системой, может выглядеть следующим образом:

```

text
intr      segment 'code'
          proc
          ...
          iret
          endp
main      proc
          push    CS           ; Установим адресруемость
          pop     DS           ; к программному сегменту
          mov     DX, offset intr ; DS:DX -> intr
          mov     AX, 25h      ; Функция заполнения вектора
          mov     AL, 0Ah      ; Тип вектора
          int     21h
          ; Вектор прерывания инициализирован.
          ; Далее текст основной программы.
          ...
main      endp

```

Если пользователя не устраивают системные алгоритмы обработки какого-то прерывания, он может заменить содержимое соответствующего вектора на адрес собственного обработчика и выполнять всю обработку прерываний своими силами. В этом случае следует в начале основной программы сохранить (например, в двухсловной ячейке old\_09) системное содержимое вектора прерывания, а в конце восстановить его:

```

main      proc
; Сохранение системного вектора в начале программы
mov       AH, 35h           ; Функция получения вектора
mov       AL, 09h           ; Номер вектора
int       21h
mov       word ptr old_09, BX ; Относительный адрес системного
                             ; обработчика
mov       word ptr old_09+2, ES ; Сегмент системного обработчика
          ...
; Восстановление системного вектора в конце программы
push      DS                ; Сохраним наш DS
lds       DX, old_09         ; Заполним DS:DX из old_09
mov       AH, 25h           ; Функция заполнения вектора
mov       AL, 09h           ; Номер вектора
int       21h
pop       DS                ; Восстановим адресруемость
          ...
main      endp

```

Поля данных  
old\_09 dd 0

Наконец, часто требуется лишь незначительно изменить или дополнить системный алгоритм. В этих случаях используется другой подход: программа пользователя "сцепляется" с системой обработки прерываний, выполняя свою часть прерывания либо до, либо после системной. Такая методика сцепления годится и для аппаратных, и для программных прерываний и используется чрезвычайно широко.

При инициализации прикладного обработчика, сцепляемого с системным, следует сохранить (например, в двухсловной ячейке old\_int) адрес системного обработчика, чтобы обеспечить возможность перехода в него, и поместить в вектор прерывания адрес (например, new\_int) прикладного обработчика. Если прикладная обработка должна выполняться после системной, как-то дополняя или корректируя ее, структура прикладного обработчика выглядит следующим образом:

```

new_int   proc
          pushf
          call   CS:old_int
          ; В системный обработчик
          ; с возвратом
          ...
          iret
          endp
new_int   endp

```

После того, как процессор выполнит процедуру прерывания, в стеке прерванного процесса оказываются три слова: слово флагов, а также двухсловный адрес возврата в прерванную программу (рис. 9.8).

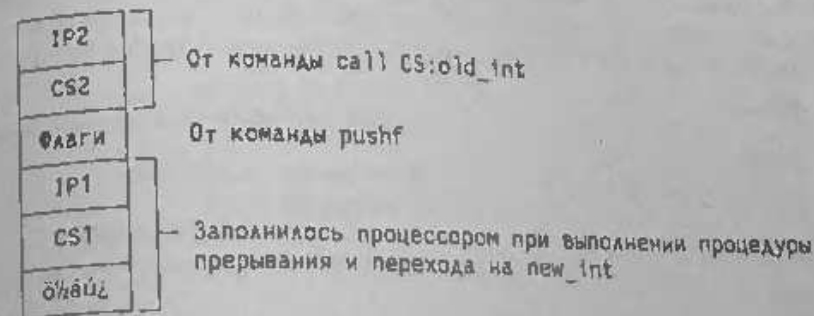


Рис. 9.8. Стек прерванной программы в процессе выполнения прикладного обработчика прерываний.

CS1 - сегментный адрес прерванного процесса;

IP1 - относительный адрес точки возврата в прерванный процесс;

CS2 - сегментный адрес прикладного обработчика;

IP2 - относительный адрес точки возврата в прикладной обработчик.

Именно такая структура данных должна быть на вершине стека, чтобы команда IRET, которой завершается любая программа обработки прерываний, могла вернуть управление в прерванный процесс.

Первая команда нашего обработчика PUSHF засылает в стек еще раз слово флагов, а команда дальнего вызова процедуры CALL CS:old\_int (где old\_int объявлено с помощью оператора DD двойным словом) в процессе передачи управления системному обработчику помещает в стек двухсловный адрес возврата на следующую команду прикладного обработчика. В результате в стеке формируется трехсловная структура, необходимая для команды IRET (рис. 9.8).

Системный обработчик, закончив обработку данного прерывания, завершается командой IRET. Эта команда забирает из стека три верхних слова и осуществляет переход по адресу CS:IP2, т.е. на продолжение прикладного обработчика.

Завершающая команда нашего обработчика IRET снимает со стека три верхних слова и передает управление по адресу CS:IP1.

Если прикладная обработка должна выполняться до системной, структура прикладного обработчика будет иной:

```
new_int:
;Прикладная обработка
...
jmp CS:old_int
```

Завершающая команда перехода передает управление (не затрагивая стека) в системный обработчик, который далее работает обычным образом.

Иногда прикладной обработчик должен выполнить некоторые действия до передачи управления в системный, а некоторые — после. Тогда используется следующая структура обработчика:

```
new_int proc
;Прикладная обработка до системной
...
pushf CS:old_int ;8 системный обработчик
call CS:old_int ;с возвратом
;Прикладная обработка после
;системной
...
iret
new_int endp
```

Наконец, бывают случаи, когда прикладной обработчик, получив управление в результате прерывания, выполняет анализ ситуации и, в зависимости от результатов этого анализа, либо передает управление системному обработчику, либо выполняет обработку сам, полностью исключив системный обработчик из

обслуживания этого конкретного акта прерывания. Тогда структура прикладного обработчика является комбинацией приведенных выше схем:

```
new_int proc
...
;Анализ ситуации. Если системная
;обработка требуется:
jmp sys

...
;Если системная обработка не требуется:
;Прикладная обработка
iret
sys: jmp CS:old_int
new_int endp
```

#### 9.4. Обработка прерываний от таймера и буллита

В составе машин типа IBM PC имеется таймер, служащий для отсчета текущего времени и работающий с частотой 1,193 МГц. Сигналы таймера после пересчета 65535:1 поступают в контроллер прерываний и инициируют прерывания через вектор 8 с частотой 18,2 1/с. Отсчет текущего времени ведется системным обработчиком BIOS (рис. 9.9).



Рис. 9.9. Прикладная обработка прерываний от таймера.

Для того, чтобы прикладные программы могли использовать сигналы таймера, не разрушая при этом работу системных часов, в обработчик BIOS включен вызов INT 1Ch, передающий управление на программу-заглушку BIOS, содержащую единственную строку IRET. Пользователь может записать в вектор 1Ch адрес прикладного обработчика сигналов таймера и



использовать в своей программе средства реального времени. Естественно, перед завершением программы следует восстановить старое значение вектора ICh (0F000h:0FF53h).

Компьютеры типа IBM PC/AT оснащаются КМОП-микросхемой с батарейным питанием, которая служит, с одной стороны, для хранения информации о конфигурации компьютера (количество и типы дисков, объем памяти и проч.), а с другой - для отсчета реального календарного времени. Поскольку эта микросхема питается от встроенной батарейки, часы реального времени продолжают работать, даже если компьютер выключен. В процессе начальной загрузки программы DOS считывают показания часов реального времени, сохраняют их в ячейках системной области и модифицируют в дальнейшем эти ячейки в соответствии с ходом работы системного таймера.

Для чтения и изменения текущего времени и даты, хранящихся в оперативной памяти, используются функции прерывания 21h DOS: 2Ah (получить дату), 2Bh (установить дату), 2Ch (получить время) и 2Dh (установить время). Для чтения же или изменения показаний часов реального времени в КМОП-микросхеме используется прерывание BIOS 1Ah. Оно также имеет несколько функций (получить или установить дату, получить или установить время и др.).

Время в КМОП-микросхеме хранится в упакованном двоично-десятичном формате, причем и при получении, и при установке времени число часов находится в регистре CH, число минут - в CL и число секунд - в DH.

Получив с помощью функций BIOS три составляющих времени из КМОП-микросхемы, преобразовав эти числа в символическую форму и выведя полученную строку на экран, мы получим программу-часы, которая при ее запуске будет выводить на экран текущее реальное время.

Прерывание BIOS 1Ah позволяет не только прочитать или установить часы реального времени, но также и "завести будильник", т.е. указать момент календарного времени, когда микросхема часов реального времени должна выдать прерывание с вектором 4Ah. Для установки будильника служит функция 06h прерывания 1Ah. Если пользователь, установив будильник на какое-то время суток, заполнит вектор 4Ah адресом собственного обработчика, он будет активизирован в заданный момент времени, независимо от того, чем занят в этот момент компьютер. После установки будильника КМОП-микросхема будет выдавать прерывание 4Ah каждые сутки в заданное время до тех пор, пока с помощью функции 07h того же прерывания BIOS мы не выключим будильник.

Скелетная схема программы установки будильника на момент времени, отстоящий от текущего на заданный интервал, выглядит следующим образом:

```

;Сохраним вектор 4Ah в двухсловной ячейке old_4ah
mov     AX,354Ah
int     21h

;Установим собственный обработчик прерывания 4Ah
mov     AX,254Ah
mov     DX,offset new_4ah
mov     DS,seg new_4ah
int     21h

;Прочитаем из КМОП-микросхемы текущее время
mov     AH,02h

int     1Ah
;Функция получения текущего
;времени
;Прерывание BIOS

;Время возвращается в двоично-десятичном упакованном формате
;в следующих регистрах:
;CH=часы
;CL=минуты
;DH=секунды
;Прибавим к полученному времени заданный временной интервал
;Результат должен содержаться в тех же регистрах CH:CL:DH
;в двоично-десятичном упакованном формате
...

;Установим будильник
mov     AH,06h
int     1Ah
;Функция установки будильника
;Прерывание BIOS

;Продолжение программы
...
```

Для того, чтобы создать программный будильник, мы должны написать прикладную программу обработчика прерывания будильника и загрузить ее адрес в вектор 4Ah. Поскольку этот обработчик обрабатывает, в сущности, аппаратное прерывание от часов, он может активизироваться в любой момент времени, в частности, когда текущая программа выполняет вызванные ею функции DOS или BIOS. Поэтому в обработчике недопустимо использование каких-либо системных средств, так как нельзя прервать выполнение программы DOS, вызвать "изнутри DOS" эту же или даже другую программу DOS. То же, хотя и в меньшей степени, относится и к программам BIOS. Таким образом, наш обработчик прерывания будильника не может использовать функции DOS или BIOS. Однако в нем можно вывести информацию на экран, если выполнить эту операцию путем непосредственного обращения в видеобuffer.

Из сказанного следует, что возможности обработчика аппаратных прерываний весьма ограничены. В настоящее время разработаны методы преодоления этого недостатка. В частности, для того, чтобы из обработчика аппаратного прерывания

обратиться к функциям DOS, надо предварительно выяснить, не выполняется ли уже какая-либо функция DOS, и продолжать выполнение программы обработчика, только если DOS "свободна". Методика написания обработчиков аппаратных прерываний, в которых можно обращаться к функциям DOS и BIOS, будет подробно рассмотрена в следующей главе, посвященной резидентным программам, в состав которых всегда входят обработчики аппаратных прерываний, и для которых проблема взаимодействия с DOS стоит весьма остро.

### 9.5. Системные стеки и обработчики прерываний

В предыдущем разделе рассматривались обработчики прерываний, использующие прерывания 1Ch и 4Ah, специально предназначенные для подключения прикладных обработчиков. На рис. 9.10 схематически изображено взаимодействие всех вычислительных объектов, принимающих участие в обработке прерывания от КМОП-микросхемы (вектор аппаратного прерывания 70h, вектор подключения прикладного обработчика 4Ah).

В случае отсутствия показанных на рис. 9.10 объектов осуществляется следующим образом. При достижении момента времени, на которое поставлен будильник и которое было записано нами в КМОП-микросхеме с помощью функции 06h прерывания 1Ah, возникает аппаратное прерывание, за которым закреплен вектор 70h. В этом векторе, как и в большинстве других векторов аппаратных прерываний, записан один из входных адресов программы DOS, которая в начале своего выполнения осуществляет переключение текущего стека программы на стек DOS, а в конце - обратное переключение на стек программы. Процессор, получив сигнал прерывания с вектором 70h, запоминает в стеке прерываемой прикладной программы флаги и адрес возврата в нее (называемые иногда вектором возврата), после чего управление передается программе DOS переключения стека. Эта программа, установив новый стек (из числа тех стеков DOS, которые объявляются с помощью директивы STACKS файла CONFIG.SYS) передает управление программе BIOS обслуживания прерывания будильника.

Программа BIOS, работая с КМОП-микросхемой на уровне ее портов, проверяет, возникло ли данное прерывание в результате достижения времени установки будильника (КМОП-микросхема может инициировать прерывания и по другим причинам), и заодно тестирует батарейное питание микросхемы. Если результаты проверки оказываются положительными, выполняется команда `int 4Ah`. Поскольку программа BIOS

работала на стеке аппаратных прерываний DOS, именно на нем при выполнении команды `int 4Ah` процессор сохраняет вектор возврата.

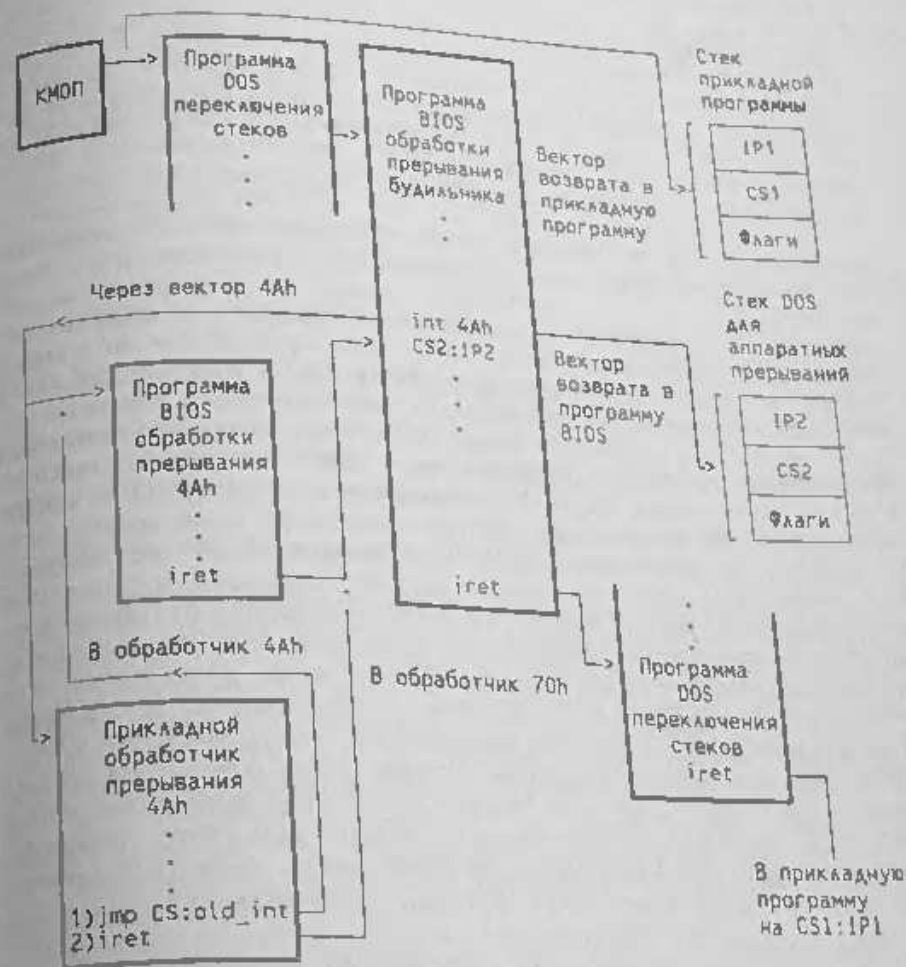


Рис. 9.10. Взаимодействие системного и прикладного обработчиков прерывания будильника.

В векторе 4Ah находится адрес универсальной программы BIOS, которая используется для обслуживания целого ряда зарезервированных векторов. Ее назначение - прочитать состояние контроллеров прерываний компьютера и при наличии запросов на прерывания замаскировать соответствующие входы



контроллеров, после чего инициировать для обоих контроллеров сигналы конца прерывания. В результате аппаратные прерывания, для которых пользователь не предусмотрел программ обработки, не будут нарушать работоспособность всей системы. Программа обработки прерывания 4Ah заканчивается командой `iret`, которая возвращает управление в программу BIOS обработки прерывания будильника. Эта программа на заключительном участке иницирует для обоих контроллеров сигналы конца прерывания и выполняет завершающую команду `iret`, по которой выполняется возврат в программу переключения стеков DOS. Та осуществляет переход на стек прикладной стеков и своей завершающей командой `iret` передает управление программе, вектор возврата которой был сохранен процессором (при обслуживании аппаратного прерывания) на стеке.

Наличие в программе пользователя прикладного обработчика прерывания 4Ah приводит к тому, что команда `int 4Ah` активизирует не универсальную программу BIOS, а прикладной обработчик. Дальнейшие процессы в системе будут зависеть от его завершающих команд. Если обработчик завершит работу по адресу, сохраненному при заполнении командой `4Ah`, то управление будет возвращено в программу BIOS обслуживания этого прерывания, которая завершит свою работу, как и при отсутствии прикладного обработчика. Если же обработчик завершит работу в программе BIOS обслуживания будильника, и системная программа обработки прерывания 4Ah выполнена не будет. В данном случае эти программы и не нужны, так как при наличии прикладного обработчика прерываний от КМОП-микросхемы маскировать эти прерывания, разумеется, не следует, а сигналы конца прерывания для обоих контроллеров все равно будут инициированы при завершении программы BIOS обработки прерывания будильника. Однако для того, уверенно решить вопрос о возможностях встраивания в систему прикладного обработчика, надо знать функции принимающих участие в процессе системных программ.

Между прочим, завершение прикладного обработчика командой `jmp CS:old-int` в данном случае также возможно. Правда, после его отработки будут замаскированы прерывания от будильника, который, таким образом, будет выключен, однако при очередной установке будильника функцией `06h` маска в контроллере прерываний будет снята.

Важным для практического программирования обстоятельством является то, что программа BIOS обработки прерывания будильника, как и любая системная программа обработки аппаратного прерывания, работает на одном из специальных стеков

DOS для аппаратных прерываний. Поскольку при выполнении команды `int 4Ah` не происходит переключения стека, прикладной обработчик также работает на этом стеке. Однако стеки DOS по умолчанию имеют незначительный размер, всего 128 байтов, и если в программе прикладного обработчика стек используется активно, может произойти переполнение стека и разрушение системы. В тех случаях, когда в прикладном обработчике требуется стек большого размера, следует либо увеличить размер системных стеков с помощью директивы `STACKS` файла `CONFIG.SYS`, либо предусмотреть собственный стек в обработчике. Процедура переключения стека будет описана в дальнейшем.

Приведенная на рис. 9.10 схема носит общий характер. Она полностью приложима к процедуре обработки прерывания от таймера (вектор аппаратного прерывания 8h, вектор подключения прикладного обработчика `1Ch`), а также и к другим аппаратным прерываниям за тем исключением, что программы BIOS для других аппаратных прерываний не предусматривают возможности включения внутрь этих программ прикладных обработчиков.

Примеры обработчиков прерываний разного рода будут приведены в последующих разделах.

## 9.6. Обработка прерываний по `<Ctrl>/C` и `<Ctrl>/Break`

Во многих вычислительных системах сочетание клавиш `<Ctrl>/C` зарезервировано для принудительного завершения активной программы и передачи управления системе. Однако для этого нужно, чтобы DOS, обрабатывая прерывания от клавиатуры, постоянно анализировала бы поступающие коды и "вылавливала" код нажатия `<Ctrl>/C` (код ASCII `03h`). MS-DOS проверяет наличие `<Ctrl>/C` во входном потоке не в программе обработки прерываний от клавиатуры, а на более высоком уровне, при выполнении программных запросов. При этом различные функции DOS по-разному реагируют на ввод с клавиатуры `<Ctrl>/C`.

Все функции DOS делятся на две группы - функции ввода-вывода с номерами `01h...0Ch` и все остальные функции, т.е. функции с номерами `00h, 0Dh...6Ch`. Функции с номерами, превышающими `6Ch`, используют расширители DOS, сетевые программы, инструментальные пакеты и другие "околосистемные" программы. Различие двух указанных групп заключается в том, что при вызове функций ввода-вывода DOS переходит на внутренний стек ввода-вывода, а при вызове всех остальных функций - на другой внутренний стек, который называется дисковым. Наличие в DOS двух внутренних стеков



обеспечивает ее частичную ресентерабельность, т.е. возможность при обнаружении ошибки в процессе выполнения какой-либо функции DOS (принадлежащей к "дисковой" группе) вызвать функции ввода-вывода для вывода на экран аварийного сообщения и ввода с клавиатуры указаний пользователя. Вопросы нересентерабельности DOS и методы ее преодоления будут рассмотрены в следующей главе.

Большая часть функций ввода-вывода из диапазона 01h...0Ch проверяет перед своим выполнением наличие в кольцевом буфере клавиатуры кода 03h (<Ctrl>/C) и при обнаружении этого кода выполняют команду int 23h. В этом векторе обычно находится адрес программы DOS, завершающей текущий процесс. Исключение составляют функции 06h и 07h, нечувствительные к <Ctrl>/C, а также функции 02h и 09h, которые (во всяком случае, в MS-DOS версий 5.0, 6.0 и 6.2) анализируют кольцевой буфер на предмет наличия там <Ctrl>/C один раз на каждые 64 вызова. Проверка на <Ctrl>/C осуществляется независимо от перенаправления ввода-вывода, а также независимо от состояния системного флага BREAK.

"Дисковые" функции выполняют проверку на <Ctrl>/C лишь в том случае, если установлен флаг BREAK, т.е. была выполнена команда DOS

BREAK ON

Таким образом, изменяя состояние BREAK, можно включать или выключать механизм реакции на <Ctrl>/C большинства функций DOS. Заметим, что речь идет практически обо всех функциях DOS: файловых, получения или установки даты и времени, выделения и освобождения памяти, запуска и завершения задач и др.

Поскольку проверка на <Ctrl>/C осуществляется только при выполнении функций DOS, нажатием <Ctrl>/C в системе MS-DOS нельзя завершить чисто процессорную (вычислительную) задачу, а только такую, в которой имеются вызовы системных функций.

Однако и такая возможность часто оказывается чрезмерной. При завершении задачи по <Ctrl>/C могут остаться невосстановленными модифицированные векторы прерываний или неправильно закрыты открытые файлы. Поэтому большинство прикладных программ не использует системный обработчик <Ctrl>/C, заменяя его собственным. При выполнении такой операции необходимо иметь в виду, что вектор 23h, как, впрочем, и любой другой, принадлежит не конкретной задаче, а всей вычислительной системе. После завершения задачи необходимо восстановить исходное содержимое вектора, так как в

противном случае ввод <Ctrl>/C при выполнении последующих задач неминуемо приведет к нарушению работы системы. Однако DOS при загрузке задачи в память копирует в определенные ячейки префикса программного сегмента содержимое векторов 22h (завершение задачи), 23h (обработка критической ошибки) и 24h (обработка критической ошибки). Стандартная системная процедура завершения задачи включает в себя восстановление исходного содержимого указанных трех векторов, которое берется из префикса программного сегмента. Таким образом, даже если прикладная программа, модифицировав вектор 23h, не восстановила его, это сделает DOS в процессе завершения задачи.

В момент передачи управления через вектор 23h система находится в обычном стабильном состоянии, что позволяет использовать в обработчике любые функции DOS (например, функции вывода). При завершении обработчика командой IRET управление вернется в программу в той же точке, где она была прервана. Однако в программе обработчика можно предусмотреть переход в любое место программы без выполнения команды IRET, что позволяет организовывать переключения по <Ctrl>/C хода выполнения программы и, в частности, ее корректное завершение. Следует только иметь в виду, что нажатие <Ctrl>/C будет отрабатываться не немедленно, а лишь когда программа дойдет до выполнения какой-либо функции DOS. В программах с большим объемом вычислительной работы эта задержка может быть значительной.

На машинах типа IBM PC имеется вторая возможность вмешательства в ход выполнения программы - нажатие клавиш <Ctrl>/<Break>.

Системный обработчик прерываний от клавиатуры, входящий в состав BIOS, при обнаружении комбинации клавиш <Ctrl>/<Break> передает управление программе, адрес которой содержится в векторе 1Bh. Эта программа, также входящая в состав BIOS, состоит из единственной команды IRET и не выполняет, таким образом, никаких функций. Однако в процессе начальной загрузки операционной системы DOS изменяет содержимое вектора 1Bh, записывая в него адрес своего обработчика. Этот обработчик, получив управление, выполняет следующие действия:

включает 0000h в кольцевой буфер клавиатуры на место головного символа;

модифицирует указатели кольцевого буфера так, что буфер представляется системе очищенным;

записывает флаг <Ctrl>/<Break> в ячейку области данных BIOS по адресу 40h:71h;

записывает код `<Ctrl>/C` (03h) в буфер драйвера консоли CON, так что драйвер "думает", что была нажата комбинация `<Ctrl>/C`.

В результате, если после нажатия `<Ctrl>/<Break>` программа вызовет какую-либо функцию DOS, выполняющую проверку на `<Ctrl>/C`, то DOS обнаружит наличие `<Ctrl>/<Break>` и передаст управление на вектор 23h точно также, как и при обнаружении `<Ctrl>/C`. Таким образом, системная обработка `<Ctrl>/<Break>` и `<Ctrl>/C` в итоге выполняется почти одинаково. Особенность обработки `<Ctrl>/C` заключается в том, что если перед вводом `<Ctrl>/C` были нажаты какие-то клавиши и их коды остались в кольцевом буфере клавиатуры, они будут "маскировать" код `<Ctrl>/C`, так как драйвер консоли всегда анализирует только самый старый из символов, находящихся в кольцевом буфере. С вводом `<Ctrl>/<Break>` ситуация иная. Программа DOS, обрабатывающая прерывание 1Bh, отправляет код 03h не в кольцевой буфер клавиатуры, а непосредственно в драйвер CON. Поэтому комбинацию `<Ctrl>/<Break>` нельзя замаскировать вводом символов с клавиатуры (к тому же, программа обработки `<Ctrl>/<Break>` очищает кольцевой буфер).

Прикладная программа может заменить содержимое вектора 1Bh адресом собственного обработчика. В этом случае при нажатии `<Ctrl>/<Break>` произойдет немедленный (через программу INT 09h и вектор 1Bh) переход на программу обработчика, который, таким образом, может взять на себя управление практически в любой точке программы, в том числе и при заиклиивании или других чисто процессорных операциях. Однако такой обработчик работает на уровне прерываний, что ограничивает его возможности. В момент прерывания могли выполняться какие-то программы DOS, поэтому завершение обработчика иначе, чем командой IRET, может привести к аварии системы. Кроме того, из обработчика нельзя обращаться к функциям DOS. Однако при всех этих ограничениях возможность в любой момент вмешаться в ход выполнения программы оказывается для некоторых приложений весьма полезной.

Исходное содержимое вектора 1Bh (в отличие от вектора 23h) не восстанавливается системой автоматически при завершении программы. Поэтому в прикладной программе, перехватывающей прерывание по `<Ctrl>/<Break>`, необходимо предусмотреть перед ее завершением восстановление исходного содержимого этого вектора. Однако пользователь может завершить программу и аварийно, нажав `<Ctrl>/C` и обойдя, тем самым, строки нормального завершения. Поэтому в программе, перехватывающей вектор 1Bh, следует предусмотреть собственный обработчик прерывания по `<Ctrl>/C`, в котором перед

завершением программы восстанавливается вектор 1Bh. Пример программы такого рода дан в задаче 9.8.

## 9.7. Задачи на обработчики прерываний

**Задача 9.5. Работа с таймером с помощью прерывания 1Ch.** Написать программу, перехватывающую прерывания от системного таймера, поступающие каждые 18,2с, пересчитывающую их для уменьшения частоты и периодически выводящую на экран какую-либо информацию. В данном примере программа обработки прерываний от таймера включена в состав основной транзитной программы. При необходимости ее можно сделать резидентной.

Основные фрагменты программы

Сделаем в программе только два сегмента - программный и стек

тургос

Заполним вектор 1Ch адресом нашего обработчика прерываний от таймера. Сохранять адрес "программы - заглушки" BIOS нет необходимости, так как он известен из документации

mov	AX, 25h	Функция загрузки вектора
mov	AL, 1Ch	Тип вектора
lea	DX, alarm	Смещение нашего обработчика
push	CS	Отправим CS в DS
pop	DS	DS:DX->alarm
int	21h	

Остановим программу, чтобы наблюдать работу таймера

Завершим программу. Сначала восстановим содержимое вектора 1Ch. Адрес "программы-заглушки" в ПЗУ BIOS известен

push	DS	Сохраним DS
mov	DX, 0FF53h	Смещение обработчика BIOS
mov	AX, 0F000h	Сегмент обработчика BIOS
mov	DS, AX	Настроим DS
mov	AX, 251Ch	Функция заполнения вектора
int	21h	
pop	DS	Восстановим DS

Завершим программу обычным образом

Наш обработчик прерываний от таймера

alarm:	push	AX	Сохраним все
	push	BX	используемые
	push	ES	регистры

Регистр DS пришел к нам из программы BIOS.

Поэтому адресуемся через CS

dec	CS:count	Декремент счетчика времени	
cmp	CS:count, 0	Отсчитали 18 прерываний?	
je	output	Да, на вывод символа	
jmp short	retret	Нет, на выход	
output:	dec	CS:nmb	Декремент счетчика символов
	cmp	CS:nmb, 0	Все символы выведены?
	je	retret	Да, на выход

```

mov     AX,CS:count1      ;Заполним счетчик
mov     CS:count,AX       ;времени константой
mov     AX,0B800h         ;Получим адресуемость
mov     ES,AX             ;к видеобуферу
mov     BX,CS:pos         ;Позиция символа
mov     AL,CS:sym         ;Символ
mov     byte ptr ES:[BX].AL ;На экран его
mov     BX,CS:sym         ;Инкремент позиции
inc     BX                ;Атрибут
mov     byte ptr ES:[BX].1Fh ;Инкремент символа (0,1,...)
inc     BX                ;Еще раз инкремент позиции
inc     BX                ;Сохраним новую позицию
mov     CS:pos,BX         ;Восстановим
retret: pop     ES         ;все
        pop     BX         ;регистры
        pop     AX         ;Выход из обработчика
        ret         ;Выводимый символ
sym     db      '0'       ;Число выводимых символов+1
rmb     db      10+1      ;Счетчик времени (1с)
count   dw      18        ;Константа счетчика времени
count1  dw      18        ;Позиция вывода на экран
pos     dw      80*2+60*2

```

Задача 9.6. Установка будильника реального времени. Прочитать из КМОП-микросхемы текущее время, прибавить к нему заданный интервал (например, 5 с) и установить будильник на полученное время. В программе обработке прерывания будильника вывести на экран символ.

```

;Основные фрагменты программы
new_4ah proc      ;Нам обработчик прерывания будильника
push     ES       ;Сохраним
push     AX       ;используемые
push     CX       ;регистры
mov     AX,0B800h ;Настроим ES
mov     ES,AX     ;на видеобуфер
mov     ES:2000,0F40Fh ;Выведен в центр экрана символ
pop     CX        ;Восстановим
pop     AX        ;используемые
pop     ES        ;регистры
ret       ;Выход из обработчика прерывания
new_4ah endp

```

;Главная процедура

;Сохраним вектор 4Ah

```

mov     AX,354Ah
int     21h
mov     word ptr old_4ah,BX
mov     word ptr old_4ah+2,ES

```

;Установим собственный обработчик прерывания 4Ah

```

mov     AX,254Ah
mov     DX,offset new_4ah
push    DS      ;Сохраним DS
push    CS      ;Настроим DS

```

```

pop     DS      ;на сегмент команда
int     21h     ;Восстановим DS
pop     DS      ;Функция получения текущего
mov     AH,02h  ;времени
int     1Ah     ;Прибавим
mov     SI,offset hour ;время из поля time
call    add_time ;mov
mov     AH,06h  ;Установим будильник
mov     CH,CL   ;Регистры CH, CL и DH уже настроены
int     1Ah     ;Остановим программу вызовом функции DOS ввода символа с клавиатуры
...           ;Функция установки будильника
mov     AH,07h  ;Сбросим будильник
int     1Ah     ;Восстановим системный обработчик 4Ah
mov     AX,254Ah
push    DS
lds     DX,old_4ah
int     21h
pop     DS
;Завершим программу
...

```

;Поля данных программы

```

hour    db      0
min      db      0
sec      db      5h
old_4ah dd      0

```

;Задаваемый интервал

;времени перед срабатыванием

;будильника

;Ячейка для хранения вектора 4Ah

;Подпрограмма сложения трех разрядов времени (часов, минут и секунд)

;в двоично-десятичном формате

add\_time proc

;При вызове:

;Исходное время: CH=часы, CL=минуты, DH=секунды (в BCD)

;Прибавляемое время в массиве из трех байтов:

;часы, минуты, секунды (в BCD), адрес массива в DS:SI

;При возврате в тех же регистрах результат сложения в том же формате

```

push    AX      ;Сохраним
push    BX      ;используемые
push    DI      ;регистры
mov     DI,1     ;Подготовим 1 для переноса

```

;Сложим секунды

```

mov     AL,DH
mov     BL,[SI+2]
mov     BH,0
push    BX
call    add_unit
mov     DH,AL
jnc     mmh

```

;Исходные секунды отправим в AL

;Получим прибавляемые секунды

;и в виде слова

;отправим в стек

;Вызов подпрограммы сложения

;Результат назад в DH

;Переноса нет, на сложение минут



```

;Перенос, прибавим 1 к минутам
mov     AL,CL
push    DI
call    add_unit
mov     CL,AL
jnc     .

;Перенос, прибавим 1 к часам
mov     AL,CH
push    DI
call    add_unit
mov     CH,AL

;Сложим минуты
mov     AL,CL
mov     BL,[SI+1]
mov     BH,0
push    BX
call    add_unit
mov     CL,AL
mov     hhh
jnc     .

;Перенос, прибавим 1 к часам
mov     AL,CH
push    DI
call    add_unit
mov     CH,AL

;Сложим часы
mov     AL,CH
mov     BL,[SI]
mov     BH,0
push    BX
call    add_unit
mov     CH,AL

;Восстановим используемые регистры
pop     DI
pop     BX
pop     AX
add_time endp

;Подпрограмма сложения одного разряда времени
add_unit proc
;При вызове:
;AL=время в BCD (секунды, минуты или часы)
;Младший байт верхнего слова стека - прибавляемая величина
;(секунды, минуты или часы)
;При возврате:
;AL=результат сложения в BCD, AH разрушен
push    BP
push    BX
mov     AH,0
mov     BP,SP
mov     BX,[BP+6]
add     AL,BL
das
;Сохраним используемые
;регистры
;Подготовим AH
;Настроим базовый регистр
;Получим параметр
;Сложим слагаемые
;как BCD

```

```

;Исходные минуты отправим в AL
;Прибавляем 1 в стек
;Вызов подпрограммы сложения
;Результат назад в CL
;Переноса нет, на сложение минут

```

```

;Исходные часы отправим в AL
;Прибавляем 1 в стек
;Вызов подпрограммы сложения
;Результат назад в CH

```

```

;Исходные минуты отправим в AL
;Получим прибавляемые минуты
;и в виде слова
;отправим в стек
;Вызов подпрограммы сложения
;Результат назад в CL
;Переноса нет, на сложение часов

```

```

;Исходные часы отправим в AL
;Прибавляем 1 в стек
;Вызов подпрограммы сложения
;Результат назад в CH

```

```

;Исходные часы отправим в AL
;Получим прибавляемые часы
;и в виде слова
;отправим в стек
;Вызов подпрограммы сложения
;Результат назад в CH

```

```

jnc     less100
mov     AH,1

less100: cmp     AX,60h
        jb     done
        sub     AX,60h
        das
        stc

        jmp     done1

done:    ctc
done1:  pop     BP
        pop     BX
        ret     2
add_unit endp

```

```

;Сумма меньше 100
;Сумма больше 100,
;запишем 01h в AH
;Нужно ли корректировать
;следующий разряд времени?
;Нет, сумма < 60
;Ав, сумма > 60, вычтем 60
;как BCD
;Установим CF, как признак
;переноса
;и на выход
;Сбросим CF (переноса нет)
;Восстановим используемые
;регистры
;Скорректируем указатель стека

```

**Задача 9.7. Транзитный обработчик прерываний по <Ctrl>/C.** Написать тестовую программу, периодически выводящую на экран строку символов (через дескриптор стандартного вывода). Период желательно выбрать не менее 2-3 секунд. Предусмотреть завершение программы после вывода 5-6 строк. Отладив программу, проверить действие на нее нажатия комбинации клавиш <Ctrl>/C. Перед запуском программы выполнить команду DOS BREAK ON.

Включить в текст программы собственный обработчик прерываний по <Ctrl>/C. В качестве функций обработчика можно принять, например, следующие действия:  
вывод на экран сообщения о перехвате <Ctrl>/C;  
изменение состояния счетчика цикла (как имитацию перенастройки программы);  
возврат в программу и ее завершение после вывода установленного числа строк.

Основные фрагменты программы  
Установим наш обработчик

```

push    DS
mov     AX,seg handler
mov     DS,AX
mov     AH,25h
mov     AL,23h
mov     DX,offset handler
int     21h
pop     DS

;Сохраним наш сегмент данных
;Сегментный адрес обработчика
;поместим в DS
;Функция установки вектора
;Номер вектора
;DS:DX->handler

;Восстановим сегмент данных
;Организуем с помощью Int 21h с функцией 40h цикла вывода
;тестовой строки tst
go:
...
;Введем задержку
...
;Счет числа выводимых строк и повторение вывода

```

```

;если в счетчике не 0
dec count
test count,0FFFFh
jne go
;Завершим программу
...
;Обработчик прерываний по <Ctrl>/C
handler: push AX
          push BX
          push CX
          push DX
;Выведем на экран сообщение по об активности обработчика
;{в таком обработчике допустимы все функции DOS}
...
pop DX
pop CX
pop BX
pop AX
;Изменим ход выполнения программы, поместив в
;счетчик цикла другое значение
mov count,8
iret
;Возврат из обработчика

;Поля данных
tst db
mes db
count dw

```

Тестовая строка  
Обработчик <Ctrl>/C получил управление, if, cr  
Счетчик числа выводимых строк

Задача 9.8. Транзитный обработчик прерываний по <Ctrl>/<Break>. Программа иллюстрирует технику перехвата <Ctrl>/<Break>, а также мгновенность реакции системы на ввод этой команды.

Программа устанавливает в векторах 1Bh и 23h адреса собственных обработчиков, после чего входит в бесконечный цикл вывода на экран символа из ячейки sum (первоначально буква а).

Обработчик прерывания по <Ctrl>/<Break> выводит на экран в заданную позицию аварийное сообщение, после чего модифицирует эту позицию, а также модифицирует байт программы с кодом выводимого символа.

Обработчик прерывания по <Ctrl>/C, активизируемый (в случае нажатия <Ctrl>/C) функцией 40h прерывания 21h, восстанавливает системное содержимое вектора 1Bh и завершает программу вызовом функции 4Ch, не заботясь о возврате из прерывания по <Ctrl>/C. Перед запуском программы следует установить флаг BREAK командой DOS

BREAK=ON

Основные фрагменты программы  
Собственный обработчик прерываний по <Ctrl>/<Break> (1Bh)  
break: push AX

```

push ES
push CX
push SI
push DI
push DS
mov AX,seg sym
mov DS,AX
mov AX,0BB00h
mov ES,AX
mov DI,offs
lea SI,breakmes
mov CX,10
movsb
;Адрес видеобуфера
;Смещение по экрану
;Адрес сообщения
;Число выводимых байтов
;Вывод на экран
;Сдвиг по экрану
;Модификация константы в программе

```

Собственный обработчик прерываний по <Ctrl>/C (23h)  
ctrlc:

Восстановим системное содержимое вектора 1Bh

```

lds DX,old_1bh
mov AX,251Bh
int 21h
;Завершим программу из обработчика <Ctrl>/C
mov AX,4C00h
int 21h

```

Основная программа

Сохраним системный вектор <Ctrl>/<Break>

```

mov AX,351Bh
int 21h
mov word ptr old_1bh,BX
mov word ptr old_1bh+2,ES
;Инициализируем вектор <Ctrl>/<Break> (1Bh)
mov AX,251Bh
push DS
push CS
pop DS

```

Настроим DS на программный  
сегмент

Инициализируем вектор <Ctrl>/C (23h), чтобы иметь возможность  
аккуратного завершения программы по <Ctrl>/C.

```

mov AX,2523h
lea DX,ctrlc
;DS настроен на программный
;сегмент
int 21h
pop DS

```

Восстановим адресуемость данных

; Будем выполнять в программе периодический вывод символа на экран  
; функцией, чувствительной к <Ctrl>/C

```
ring: mov     AH, 40h
      mov     BX, 1
      mov     CX, 1
      mov     DX, offset sym
      int     21h
```

; Введем достаточно большую задержку (2 - 5с)

```
jmp     ring
```

; Заключим программу

```
; Имя данных
sym     db     'a', 'b', '4', 'R', '4', 'E', '4', 'A', '4', 'K', '4'
breakmes db     0
old_16h dd     800
offs    dw
```

; Начальная позиция на экране

**Задача 9.9.** Установить транзитный обработчик прерываний от клавиатуры, включив его в текст прикладной программы. Обработчик должен перехватывать код "горячей клавиши" (например, клавиши <F10>) с целью выполнения некоторых аварийных действий. Коды всех остальных клавиш должны передаваться системному обработчику прерываний от клавиатуры и обрабатываться обычным образом.

В данном примере рассматривается аварийное завершение программы, выполняющей какие-то вычислительные операции. Для этого в обработчике прерываний изменяется адрес возврата в стеке. На место адреса возврата в точку прерывания загружается адрес точки аварийного выхода (в основной программе). В результате после выхода из прерывания основная программа не продолжает прерывание действия, а аварийно завершается.

Такую методику нельзя использовать, если основная программа обращается к функциям DOS или BIOS, так как при выполнении программ DOS состояние стека нам неизвестно, и мы заменим не адрес возврата, а что-то другое, разрушив в итоге систему.

; Основные фрагменты программы

F10=44h ; Скан-код горячей клавиши <F10>

; Секция инициализации

```
mov     AH, 35h ; Функция получения вектора
mov     AL, 09h ; Номер вектора
int     21h ; Переход в MS-DOS
mov     word ptr old_09, BX ; Сохраним старое IP
; системной программы 09h
mov     AX, ES ; Сохраним старое CS
mov     word ptr old_09+2, AX ; системной программы 09h
push    DS ; Сохраним на время DS
mov     AX, seg new_09 ; Сегментный адрес обработчика
mov     DS, AX ; Он нам нужен в DS
mov     DX, offset new_09 ; Теперь DS:DX - наш обработчик
mov     AH, 25h ; Функция заполнения вектора
```

```
mov     AL, 09h
int     21h ; Номер вектора
pop     DS ; Переход в MS-DOS
; Восстановим наш DS
; Введем сообщение mes о загрузке обработчика
; любой подходящей функцией DOS
```

; В качестве имитации вычислительной работы программы  
предусмотрим в ней бесконечный цикл. Такой цикл  
нельзя прервать системными средствами (<Ctrl>/C),  
что и оправдывает рассматриваемую методику

```
loop1: jmp     loop1
norm_out: ; Нормальное завершение программы
; Восстановим исходное содержимое вектора 09 перед
; завершением программы. Это делать обязательно, так как
; вектор 09 автоматически системой не восстанавливается
```

```
push    DS ; Сохраним на время DS
mov     DX, word ptr old_09 ; Двухсловный адрес
mov     AX, word ptr old_09+2 ; системного обработчика
mov     DS, AX ; поместим в DS:DX
mov     AH, 25h ; Функция заполнения вектора
mov     AL, 09 ; Номер вектора
int     21h ; Переход в MS-DOS
pop     DS ; Восстановим наш DS
; Завершим программу
```

err\_out: ; Аварийное завершение программы по нажатию горячей клавиши  
; Введем аварийное сообщение прямым обращением к видеопамати

```
mov     AX, 0B800h ; Настроим ES
mov     ES, AX ; на начало видеобuffers
mov     DI, 80*2*12+35*2 ; Сместимся в середину экрана
lea     SI, err_mes ; Адрес сообщения
mov     CX, err_len ; Его длина
rep     movsb ; Пересылка в видеобuffer
jmp     norm_out ; И на нормальное завершение
```

new\_09: ; Программа обработчика прерываний от клавиатуры

```
proc     far
push    AX ; Сохраним AX прерываемой программы
in      AL, 60h ; Введем скан-код нажатой клавиши
cmp     AL, F10 ; Нажата "горячая клавиша"?
je       hotkey ; Да, перейдем на ее обработку
pop     AX ; Нет, восстановим AX
jmp     CS:old_09 ; И перейдем в системный обработчик
```

hotkey:

; Наша собственная обработка прерывания от клавиатуры

; Прейдем всего следует выполнить определенные действия

; в порте B (61h) программируемого периферийного интерфейса

```
in      AL, 61h ; Получим содержимое порта B
or      AL, 80h ; Запретим работу клавиатуры,
; добавив бит 80h к содержимому
out     61h, AL ; порта 61h
```



```

and     AL, 7Fh
out     61h, AL
; Действия по нажатию "горячей клавиши"
push    BP
mov     BP, SP
; Загрузим адрес аварийного завершения в стек на место "нормального"
; адреса возврата. Стек сейчас смещен относительно адреса возврата
; на 2 слова (командами push AX и push BP)
mov     word ptr 4[BP], offset err_out
; Восстановим BP
pop     BP
; Теперь оповестим о завершении обработки прерывания контроллер
; прерываний, посыл в порт 20h сигнал завершения прерывания EO1
mov     AL, 20h
out     20h, AL
; Сигнал EO1
; В порт 20h
; Восстановим AX
pop     AX
; Возврат в прерываемую программу
iret

new_09  endp
; Поля данных
old_09  dd      0
; Два слова для адреса системного
; обработчика прерывания 09
mes      db 10, 13, 'Обработчик прерывания от клавиатуры'
db      'установлен', 10, 10, 13
; Длина сообщения
meslen = $ - mes
err_mes  db      'A', 8Ch, 'B', 8Ch, 'C', 8Ch, 'P', 8Ch
db      'M', 8Ch, 'Я', 8Ch, '!', 8Ch
errlen = $ - err_mes

```

Задача 9.10. Изучить реакцию функции 02h (вывод символа) на ввод с клавиатуры <Ctrl>/C. Для этого предусмотреть в программе собственный обработчик прерывания int 23h, который должен просто возвращать управление в программу. Функцию 02h включить в цикл. Если, запустив программу, вводить с клавиатуры <Ctrl>/C, можно наглядно наблюдать моменты отработки <Ctrl>/C, поскольку система, обнаружив код 03 в кольцевом буфере, перед выполнением команды int 23h выводит на экран символы 'C'.

Основные фрагменты программы

Предусмотрим в программе собственный обработчик <Ctrl>/C

```

mov     AX, 2523h
push    DS
push    CS
pop     DS
mov     DX, offset new_23h
; Функция загрузки вектора
; Сохраним DS
; Настроим DS на сегмент
; команда
; Смещение нашего обработчика
; int 23h

int     21h
pop     DS
mov     CX, 40
push    CX
mov     CX, 10
; Восстановим DS
; Будем повторять 40 раз
; Сохраним внешний счетчик
; Будем выводить строку из 10
; символов
; Начнем с начала строки
; Исследуемая функция

```

```

mov     DI, sum[BX]
int     21h
; Текущий символ в DI
; Введем в программу задержку на 1-2 секунды
...
inc     BX
loop    abcd
pop     CX
loop    fgeh
; Инкремент индекса в строке
; Повторить 10 раз
; Счетчик внешнего цикла
; Повторить 40 раз
; Завершим программу
...
; Наш обработчик прерывания по <Ctrl>/C. Нам надо, чтобы он не завершал
; программу и вообще ничего не делал. Команда iret в обработчике int 23h
; возвращает управление на выполнение прерывной функции
new_23h proc
iret
new_23h endp
; Поля данных
sum      db      '-123456789'

```

## 10. Резидентные программы

### 10.1. Основы организации резидентных программ

Большой класс программ, обеспечивающих функционирование вычислительной системы (драйверы устройств, оболочки DOS, программы шифрации и защиты данных, русификаторы, обслуживающие программы типа электронных блокнотов или калькуляторов и др.), должны постоянно находиться в памяти и мгновенно реагировать на запросы пользователя, или на какие-то события, происходящие в вычислительной системе. Такие программы носят названия программ, резидентных в памяти (Terminate and Stay Resident, TSR), или просто резидентных программ. Сделать резидентной можно как программу типа .COM, так и программу типа .EXE, однако поскольку резидентная программа должна быть максимально компактной, чаще всего в качестве резидентных используют программы типа .COM.

Программы, предназначенные для загрузки и оставления в памяти, обычно состоят из двух частей (секций) - инициализирующей и рабочей (резидентной). В тексте программы резидентная секция размещается в начале, инициализирующая - за ней.

При первом вызове программы она загружается в память целиком и управление передается секции инициализации, которая заполняет или модифицирует векторы прерываний, настраивает программу на конкретные условия работы (возможно, исходя из параметров, переданных программе при ее вызове с помощью параметров командной строки) и с помощью прерывания DOS Int 21h с функцией 31h завершает программу, оставляя в памяти ее резидентную часть. Размер резидентной части программы (в параграфах) передается DOS в регистре DX. Указывать при этом сегментный адрес программы нет необходимости, так как он известен DOS. Для определения размера резидентной секции ее можно завершить предложением вида

ressize=\$-myproc

где myproc - смещение начала программы, а при вызове функции 31h в регистр DX заставить результат вычисления выражения  $(ressize+10Fh)/16$ .

Функция 31h, закрепив за резидентной программой необходимую для ее функционирования память, передает управление командному процессору COMMAND.COM, и вычислительная система переходит, таким образом, в исходное состояние. Наличие программы, резидентной в памяти, никак не отражается на ходе вычислительного процесса за исключением того, что уменьшается объем свободной памяти. Одновременно может быть загружено несколько резидентных программ.

Для того, чтобы активизировать резидентную программу, ей надо как-то передать управление и, возможно, параметры. Запустить резидентную программу можно тремя способами:

- вызвать ее оператором CALL как подпрограмму;
- использовать механизм асинхронных (аппаратных) прерываний;
- с помощью синхронного (программного) прерывания.

Кроме того, специально для взаимодействия с резидентными программами в DOS предусмотрено мультиплексное прерывание 2Fh.

Первый способ требует наличия в памяти текущей активной программы, которая, очевидно, должна образовывать с родительской программой единый программный комплекс с определенными заранее соглашениями взаимодействия (межпрограммным интерфейсом). Оформив в виде резидентной программы процедуры выполнения функций, общих для группы транзитных программ, можно упростить структуру и объем транзитных программ такого многопрограммного комплекса.

Второй способ, асинхронная активизация резидентной программы внешним прерыванием (от таймера, клавиатуры, последовательного порта или другого периферийного оборудования) широко используется системными и прикладными резидентными программами: спулерами принтеров, программами календарей-часов, русификаторами, калькуляторами, электронными блокнотами, резидентными электронными справочниками и базами данных и т.д.

Наконец, синхронная передача управления резидентной программе с помощью команды INT является основой взаимодействия с функциями DOS и BIOS и иногда используется в прикладных резидентных программах, в частности, при их отладке с помощью прерывания INT 3.

Рассмотрим типичную структуру резидентной программы и системные средства оставления ее в памяти. Как уже отмечалось, резидентные программы чаще всего пишутся в формате .COM:

```

text      segment "code"
          assume CS:text, DS:text
          org      100h
myproc    proc      far           ;Переход на секцию инициализации
          jmp      init          ;Данная резидентная секция программы
          ...
entry:    ...                    ;Текст резидентной секции
          ...
програма  iret                  ;Размер (в байтах) резидентной
myproc    endp                  ;программы
          resize=equ-$-myproc    ;Секция инициализации
          section
init      proc
          ...
          mov     DX, (resize+10Fh)/16 ;Размер в параграфах
          mov     AX, 3100h           ;Функция "завершить и оставить в
          int     21h                 ;памяти"
init      endp
text      ends
          end      myproc

```

При первом запуске программы с клавиатуры управление передается на начало процедуры `myproc` (первый байт после префикса программного сегмента). Командой `JMP` осуществляется переход на секцию инициализации, в которой, в частности, подготавливаются условия для дальнейшей активизации программы уже в резидентном состоянии. Последними строками секции инициализации вызывается функция `31h`, которая выполняет завершение программы с оставлением в памяти указанной ее части. С целью экономии памяти секция инициализации располагается в конце программы и отбрасывается при ее завершении.

Содержательная часть резидентной программы, начинающаяся с метки `entry`, активизируется, как уже отмечалось выше, с помощью аппаратного или программного прерывания или командой `CALL`. В последнем случае программа резидентной секции заканчивается командой `RET` (вместо `IRET`). На рис. 10.1 приведена типичная структура резидентной программы.

Как видно из рис. 10.1, резидентная программа имеет по крайней мере две точки входа. После загрузки программы в память командой оператора, вводимой на командной строке, управление передается в точку, указанную в поле завершающего текст программы оператора `end` (на рисунке - начало процедуры `myproc`). Для программ типа `.COM` эта точка входа должна соответствовать самой первой строке программы, идущей

вместо за префиксом программного сегмента. Поскольку при загрузке программы должна выполняться ее установка в память, первой командой программы всегда является команда перехода на секцию инициализации и установки (`jmp init` на рисунке).

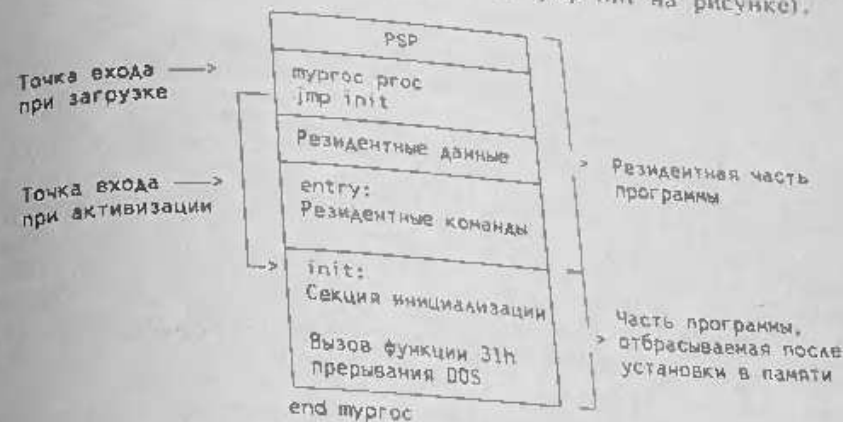


рис. 10.1. Структура резидентной программы.

После установки в памяти резидентная программа остается пассивной и никак не проявляет своего существования, пока не будет активизирована предусмотренным в ней для этого способом (аппаратным или программным прерыванием или командой дальнего вызова). Эта, вторая точка вызова обозначена на рисунке меткой `entry`.

## 10.2. Связь с резидентной программой

Для обращения к резидентной программе из транзитной можно использовать область межзадачных связей, являющуюся частью области данных BIOS и расположенную по адресам `40h:F0h...40h:FFh`. Эта область не используется системой и предназначена специально для коммуникации программ пользователя. С помощью области межзадачных связей можно осуществлять как вызов резидентной программы, так и передачу параметров. Удобнее передавать не сами параметры, а их адреса; поскольку полный адрес занимает два слова, в области межзадачной связи помещается 4 адреса.

По ходу выполнения секции инициализации будущая резидентная программа помещает в обусловленные ячейки области межзадачной связи адрес своей точки входа, например, в слово `40h:F0h` относительный адрес команды с меткой `entry`, а в



слово 40h:F2h - содержимое сегментного регистра CS. Транзитная программа, желая передать управление резидентной, настраивает регистр ES на начало области данных BIOS и выполняет команду дальнего вызова

```
call dword ptr ES:0F0h
```

В этом случае резидентная программа должна быть объявлена, как дальняя процедура (turproc proc far) и завершаться командой RET дальнего возврата.

Если резидентной программе требуется передавать параметры, их начальный адрес (или, при необходимости, адрес целого списка адресов) заносится транзитной программой в другую обусловленную ячейку области межзадачной связи, например, двухсловную ячейку 40h:F4h. В этом случае резидентная программа переносит, например, в регистр SI относительный адрес параметра из ячейки 40h:F4h, а в регистр DS - сегментный адрес параметра из ячейки 40h:F6h и после этого забирает сами параметры командами типа

```
mov ax, ds:[si]
mov bx, ds:[si+2] и т.д.
```

Естественно, перед началом выполнения резидентная программа должна сохранить все используемые ею регистры (а перед завершением - восстановить их).

Более изящный метод передачи резидентной программе управления и параметров заключается в использовании свободных векторов, например, 60h...66h. В процессе инициализации резидентная программа помещает свой адрес в свободный вектор:

```
mov ax, 0
mov es, ax
mov es:180h, offset entry ;Адрес вектора 60h
mov es:182h, CS
```

Для активизации резидентной программы в транзитной программе достаточно выполнить команду

```
int 60h
```

В этом случае резидентная программа должна заканчиваться командой IRET. Адреса параметров можно передавать через другие свободные векторы, например, 61h или 62h. Естественно, по-прежнему можно воспользоваться областью межзадачных связей.

Если резидентная программа должна запускаться непосредственно с клавиатуры, адрес ее точки входа entry должен быть помещен в вектор прерываний от клавиатуры 09h. При этом, естественно, возникает проблема взаимодействия с системным обработчиком прерываний от клавиатуры (а также

русификатором и остальными программами, перехватывающими прерывание 09h). Вопросы взаимодействия прикладных и системных обработчиков прерываний были рассмотрены в предыдущей главе.

Следует заметить, что реальные резидентные программы, как правило, перехватывают целый ряд аппаратных и программных прерываний и имеют, соответственно, не одну точку входа активизации, а несколько. Необходимость такого построения резидентных программ будет обоснована ниже.

Для взаимодействия с параллельными процессами и, в частности, с резидентными программами в системе MS-DOS предусмотрен стандартный интерфейс - прерывание 2Fh. Перед вызовом этого прерывания следует поместить в регистр AH номер функции, а в регистр AL - номер подфункции. Функции 00h...BFh зарезервированы за DOS (например, функция 01h позволяет управлять процессом вывода файлов на печать с помощью системной программы PRINT.COM, а функция 02h дает возможность определить, загружена ли резидентная часть утилиты - расширения DOS ASSIGN); функции C0h...FFh зарезервированы для прикладных программ. Если с помощью прерывания 2Fh предполагается передача в резидентную программу или получение из нее большого количества параметров, можно использовать и другие регистры, например, DS:DX или DS:SI и передавать через них адреса списков параметров.

Согласно системным соглашениям прерывание 2Fh возвращает в регистре AL состояние резидентной программы:

AL=0 - программа не установлена (т.е. в памяти еще нет этой резидентной программы) и ее можно устанавливать;  
AL=1 - программа не установлена и ее установить нельзя;  
AL=FFh - программа уже установлена и, следовательно, ее повторная установка не требуется.

В случае ошибки следует установить флаг CF, а в регистре AX вернуть код ошибки.

Прерывание 2Fh используется прежде всего с целью исключить повторную установку уже загруженной в память программы; иногда с помощью этого прерывания уже загруженной резидентной программе передается необходимая информация (например, приказ на выгрузку из памяти).

Для того, чтобы резидентная прикладная программа отзывалась на прерывание 2Fh, в нее следует включить прикладной обработчик одной или нескольких функций этого прерывания. Тогда вызов соответствующей функции этого прерывания в любой транзитной (или другой резидентной) программе позволит организовать взаимодействие с загруженной резидентной программой.

Рассмотрим структуру резидентной программы с обработчиком прерывания 2Fh. Закрепим за нашей программой функцию 00h этого прерывания.

```

;Секция инициализации
;Сохраним адрес системного обработчика 2Fh, чтобы не лишать себя
;возможности использовать системные функции этого прерывания
init: mov     AH, 35h           ;Функция получения вектора
      mov     AL, 2Fh          ;Номер вектора
      int     21h
      word ptr old_2fh, BX    ;Сохраним вектор 2Fh
      mov     word ptr old_2fh+2, ES ;в ячейке old_2fh
      mov     word ptr 2Fh, BX ;Функция установки вектора
;Загрузим в вектор 2Fh адрес нашего обработчика
      mov     AH, 25h          ;Номер вектора
      mov     AL, 2Fh
      mov     DX, offset new_2fh
      int     21h
;Другие действия по инициализации
...
;Завершим программу, оставив ее резидентной в памяти с помощью
;функции 31h прерывания DOS 21h
...
;Резидентные поля данных
old_2fh dd 0
;Ячейка для сохранения исходного
;содержимого вектора 2Fh

;Рабочая секция
entry:
...
iret
;Наш обработчик прерывания 2Fh (функция 00h)
new_2fh: cmp     AH, 00h       ;Наша функция?
         je      c0h           ;Наша!
         jmp     CS:old_2fh     ;Не наша, в системный обработчик
c0h:
;Возможно, анализ AL и переходы на программы реализации подфункций:
;передача или получение параметров, настройка резидентной программы,
;выгрузка ее из памяти и т.д.
...
iret
;Возврат в вызвавшую программу

```

### 10.3. Проверка на повторную установку

После того, как программа, которой надлежит стать резидентной, загружена и оставлена в памяти с помощью функции DOS 31h, управление передается COMMAND.COM, который ожидает последующих команд оператора. Система не выполняет какой-либо проверки имен запускаемых программ, поэтому оператор может по ошибке повторно или даже многократно запустить ту же программу, в результате чего в памяти останутся резидентными два или несколько экземпляров одной и той же программы. В зависимости от характера настройки,

выполняемой на этапе инициализации, новые экземпляры программы могут логически "деактивировать" предыдущие, но могут и "сцепляться" с ними, так что при передаче управления резидентной программой будут последовательно выполняться все ее копии в памяти. Даже если такое многократное выполнение не приведет к сбою системы, повторной загрузки следует избегать, чтобы не расходовать напрасну память.

Для защиты резидентной программы от повторной загрузки можно включить в загрузочный модуль программы некоторый произвольный код (сигнатуру) и проверять его наличие в памяти в процессе инициализации программы. Такой метод удобен для резидентных программ, запускаемых через аппаратный вектор прерывания, например, 09h. Если наша программа еще не установлена, в векторе 09h хранится адрес системного обработчика, в котором, естественно, нет сигнатуры. Если же наша программа уже установлена, сигнатура в памяти имеется; процесс повторной инициализации обнаружит ее и аварийно завершится.

Структура программы, защищающей себя от повторной загрузки, может выглядеть следующим образом.

```

org 100h
myproc proc
      jmp     init
      dw      1234h
mark:
entry:
;Рабочая секция
...
iret

init:
;Секция инициализации
      mov     AX, 3509h
      int     21h
      sub     BX, 2
      cmp     word ptr ES:[BX], 1234h
      je      installed
;Сигнатура не обнаружена. Продолжение инициализации.
;Установка программы в памяти
...
installed:
;Вывод сообщения "Программа уже установлена!"
...
      mov     AX, 4C01h
      int     21h
;Завершение с кодом ошибки и без
;оставления в памяти

```

Следует отметить, что если после инициализации и установки описанной выше программы будет загружена другая

резидентная программа обработки того же прерывания, то рассмотренная методика не спасет от повторной загрузки, поскольку в векторе прерывания будет находиться адрес другой программы, в которой данная сигнатура отсутствует.

Общепринятым методом защиты резидентных программ от повторной загрузки является использование мультитеплексного прерывания 2Fh. Структура резидентной программы с такой защитой может выглядеть следующим образом (в реальных программах секция инициализации располагается всегда в конце текста программы).

;Секция инициализации

init:

```

;Прежде всего проверим, не загружена ли уже эта программа
mov     AX,0C000h      ;Функция C0h, подфункция 00h
int     2Fh            ;Наш обработчик должен вернуть
                        ;AL=FFh
                        ;Установлена?
                        ;Да, уже! - на аварийный выход
                        ;инициализации
                        ;Программа не установлена. Приступим к
;Сохраним адрес системного обработчика 2Fh
mov     AH,35h         ;Функция получения вектора
mov     AL,2Fh         ;Номер вектора
int     21h
mov     word ptr old_2fh,BX ;Сохраним вектор 2Fh
mov     word ptr old_2fh+2,ES ;в ячейке old_2fh
;Загрузим в вектор 2Fh адрес нашего обработчика
mov     AH,25h         ;Функция установки вектора
mov     AL,2Fh         ;Номер вектора
mov     BX,offset new_2fh
int     21h
;Продолжение инициализации: загрузка других векторов, вывод
;сообщения об установке программы и правилах работы с ней

```

... ;Оставим программу в памяти

```

mov     AX,3100h
mov     DX,(ressize+10Fh)/16
int     21h

```

new\_2fh proc

;Обработчик 2Fh, входящий в состав резидентной секции программы

```

cld     AX,0C000h      ;Наша функция и подфункция?
jne     notour         ;Нет, вызовем системный обработчик
                        ;прерывания 2Fh
mov     AL,0FFh        ;Программа уже загружена (иначе
                        ;откуда взяться этому
                        ;обработчику?), вернем код FFh
iret                     ;Выход из обработчика
notour: jmp     CS:old_2fh ;Переход в системный обработчик
                        ;прерывания 2Fh

```

;Резидентные поля данных

```

old_2fh dd     0        ;Ячейка для сохранения исходного
                        ;содержимого вектора 2Fh

```

#### 10.4. Задачи на резидентные программы

**Задача 10.1.** Передача параметров через область межадачной связи BIOS, расположенной по адресам 40h:F0h - 40h:FFh. В рассматриваемой задаче резидентная программа в процессе своей инициализации помещает по адресу 40h:F0h относительный адрес своей точки входа, а по адресу 40h:F2h - свой сегментный адрес. Транзитная программа помещает в ячейку 40h:F4h произвольное двоичное число и через вектор 40h:F0h передает управление резидентной программе с помощью команды дальнего (межсегментного) вызова подпрограммы. Резидентная программа извлекает переданное ей данное, преобразует его в десятичную форму в кодах ASCII и помещает результат связи BIOS, начиная с адреса 40h:F6h, после чего завершает свою работу командой дальнего возврата RET. Транзитная программа, возобновив выполнение, забирает преобразованное число из области связи и выводит его на экран подходящей функцией DOS.

Схема использования области межадачной связи для этой задачи приведена на рис.10.2.



Рис. 10.2. Использование области межадачных связей и конкретной задачи.

#### Основные фрагменты резидентной программы комплекса

myproc	proc	far	;Дальняя процедура
	jmp	init	;Переход на секцию инициализации
entry:	push	AX	;Сохраним все используемые
	push	BX	;в резидентной программе
	push	CX	;регистры
	push	DX	
	push	SI	
	push	DS	;Сохраним DS основной программы



; Настроим DS на начало области данных BIOS

```

mov     AX, DS:0F4h
mov     BX, 0F6h
; Программа преобразования двоичного числа
; в ASCII - представление (А.Скэнлон)
push    CX, 6
mov     byte ptr [BX], ' '
fill_buff: mov     BX
inc     fill_buff
loop    SI, 10
mov     AX, AX
or      clr_dvd
jns     AX
neg     DX, DX
xor     SI
div     DX, '0'
add     BX
dec     [BX], DL
mov     CX
inc     AX, AX
or      clr_dvd
jnz     AX
pop     AX, AX
or      no_more
jns     BX
dec     byte ptr [BX], '-'
mov     CX
inc     DS
pop     SI
pop     DX
pop     CX
pop     BX
pop     AX
; Выход из подпрограммы
ret

```

init:  
; Секция инициализации резидентной программы  
; Настроим ES на начало области данных BIOS

```

; Отправим в 40h:F0h смещение, а в 40h:F2h сегментный адрес
mov     ES:0F0h, offset entry
mov     ES:0F2h, CS
; Завершим программу, оставив ее резидентную часть в памяти
mov     AX, 3100h
mov     DX, (init-мупроц+10Fh)/16
int     21h

```

### Основные фрагменты транзитной программы комплекса.

; Программа включает только два сегмента - программный и стека.  
; Сегментный регистр DS используем для адресации к области  
; межадочной связи BIOS.

; Настроим DS на начало области данных BIOS

```

mov     AX, 0FFFFh
mov     DS:0F4h, AX
call    dword ptr DS:0F0h
; Выведем преобразованное двоичное число в ASCII-представление
; на экран функцией 40h прерывания 21h. Для этой функции адрес выводимой
; строки указывается в DS:DX. DS у нас уже настроен, осталось занести
; F6h в DX и настроить должным образом остальные регистры
; Завершим программу

```

Задача 10.2. Передача параметров через вектор 60h, предназначенный для прерываний пользователя.

В рассматриваемой задаче резидентная программа в процессе своей инициализации помещает в вектор 60h свой двухсловный адрес. Транзитная программа передает управление резидентной программе командой `int 60h`. Резидентная программа имеет те же функции, что и в задаче 10.1, однако не пользуется областью межадочной связи BIOS, а взаимодействует с транзитной программой через регистры общего назначения, принимая исходное двоичное число в регистре AX, а двухсловный адрес результата - в регистрах ES:BP. Соответственно транзитная программа должна перед вызовом резидентной загрузить эти регистры.

### Основные фрагменты резидентной программы комплекса

```

muproc proc far
        jmp     init
entry:  push    CX
        push    DX
        push    SI
; На секцию инициализации
; Сохраним все используемые
; регистры (но не AX, ES и BP)
; Программа преобразования двоичного числа в ASCII-представление
; (А.Скэнлон). Приведена в задаче 10.1, однако относительный адрес
; строки теперь не в BX, а в ES:[BP]. Поэтому следует скорректировать
; все строки, где встречается ссылка на [BX] или BX.

```

```

endres equ     $-мупроц
init:
; Секция инициализации резидентной программы
; Настроим ES на начало области векторов прерываний
; Заполним вектор прерывания пользователя 60h
lea     AX, entry
mov     ES:180h, AX
mov     ES:182h, CS
; Завершим программу, оставив ее резидентную часть в памяти

```

Основные фрагменты транзитной программы комплекса.

Программа включает, как обычно, три сегмента - программный, данных и стека.

```

...
mov     AX, seg string      ; Сегмент буфера результата
mov     ES, AX              ; Подготовим его для передачи
mov     AX, 0FFh            ; Исходное число
mov     BP, offset string   ; Смещение буфера результата
mov     int 60h             ; Вызов резидентной программы
; Выведем полученную строку функцией 09h прерывания 21h

...
; Область данных
string db 8 dup ('-'), 10, 13, '$'

```

Задача 10.3. Синхронная активизация резидентной программы командой с клавиатуры.

В рассматриваемой задаче передача управления резидентной программе осуществляется, как и в задаче 10.2, через вектор 60h. Транзитная программа вводит данные с клавиатуры и как-то их обрабатывает. При поступлении с клавиатуры кода "горячей клавиши" (в примере - <Alt>/<F1>) выполняется команда int 60h, приводящая к вызову резидентной программы.

В резидентной части программы обращением к функции 2Ch прерывания 21h определяется системное время, преобразуется в строку ASCII и выводится на экран функцией 09h прерывания 21h. После этого управление возвращается транзитной программой.

Основные фрагменты резидентной программы комплекса

```

murgos proc far
        jmp     init      ; На секцию инициализации
; Поля данных резидентной программы
string db 27, 's'        ; Сохраним позицию курсора
        db 27, '[1h'      ; Назначим позицию курсора
        db 27, '[34:41m'  ; Изменим атрибуты
time    db '00:00:00'     ; Шаблон выводимой строки
        db 27, '[0m'      ; Отменим атрибуты
        db 27, '[a'       ; Восстановим позицию курсора
        db '$'            ; Конец строки
ten     db 10             ; Делитель
; Подпрограмма преобразования восьмиразрядного двоичного числа
; (часы, минуты или секунды) в двухразрядное десятичное
; в ASCII-представлении
; На входе: AL - число
; BX - адрес буфера для ASCII-представления
binasc proc near
        xor     AH, AH     ; Очистим AH
        div     ten        ; Разделим на 10
        add     AL, '0'     ; Преобразуем в код ASCII остаток
        mov     [BX], AL    ; Отправим его в строку
        add     AH, '0'     ; Преобразуем в код ASCII частное

```

```

        mov     1[BX], AH   ; Отправим его в строку
        ret     2           ; Возврат из подпрограммы
binasc endp
entry:
; Сохраним в стеке используемые в программе регистры:
; AX, BX, CX, DX, ES и DS
...
; Настроим DS на сегментный адрес программного сегмента
...
; Получим время
mov     AH, 2Ch
int     21h                ; Функция получения времени
; Преобразуем часы (CH)
mov     AL, CH
lea     BX, time
call    binasc             ; Настроим регистры для
; Аналогично преобразуем минуты (CL), отправив их в time+3
; Аналогично преобразуем секунды (DH), отправив их в time+6
; Выведем всю строку string функцией 09h прерывания 21h
; Восстановим все сохраненные регистры в обратном порядке
...
        iret              ; Возврат в транзитную программу
endres=$-murgos
init:
; Секция инициализации резидентной программы
; Заполним вектор 60h
...
; Выведем сообщение message о загрузке резидентного таймера
...
; Завершим программу, оставив в памяти ее резидентную часть
...
; Поля данных секции инициализации резидентной программы
message db 27, '[2J', 27, '[12:22h', 27, '[31:40m'
        db 'Резидентный таймер загружен в память'
        db 27, '[0m$'

```

Основные фрагменты транзитной программы комплекса.

Программа включает лишь сегменты программы и стека

```

...
; Будем вводить и анализировать символы
again:  mov     AH, 01h     ; Введем символ. Функция
        int     21h        ; чувствительна к <Ctrl>/C
        cmp     AL, 0       ; Расширенный код ASCII?
        je      ext_ascii  ; Да
        jmp     again      ; Нет, продолжим ввод
ext_ascii: mov     AH, 01h  ; Введем второй байт
        int     21h        ; расширенного кода ASCII
        cmp     AL, 68h     ; <Alt>/<F1>?
        jne     again      ; Нет, будем вводить снова

```

```

intr:    int     80h      ;Да, на резидентную программу
        jmp     again    ;Будем вводить снова

```

**Задача 10.4.** Защита резидентной программы от повторной загрузки. В программе иллюстрируется методика защиты с помощью сигнатуры, включенной в резидентную часть загрузочного модуля. Поскольку изучаемые в настоящем примере действия выполняются в секции инициализации, рабочая секция программы содержит единственную команду `jmp CS:old_09h` и лишь имитирует реальную резидентную программу.

```

;Основные фрагменты программы
text     segment 'code'
        assume  CS:text, DS:text
        org     100h

myproc   proc
        jmp     init
old_09h  dd      0
mark     dw      1234h      ;Сигнатура
entry:   jmp     CS:old_09h

ressize=$-myproc
init:    mov     AX, 3509h    ;Получим и сохраним содержимое
        int     21h          ;старого вектора
        mov     word ptr old_09h, BX
        mov     word ptr old_09h+2, ES
        cmp     word ptr ES:[BX-2], 1234h ;Анализ сигнатуры
        je      installed    ;Уже установлена!

;Не установлена. Установим.
        mov     AX, 2509h    ;Заполним вектор адресом нашей
        mov     DX, offset entry ;программы
        int     21h

;Выведем сообщение mes1 об установке (любой подходящей функцией DOS)
...
;Завершим и оставим в памяти резидентную часть программы
; (до символа ressize)
...
installed:
;Выведем сообщение mes2 о недопустимости повторной загрузки
...
;Завершим программу обычной функцией завершения 4Ch
        mov     AX, 4Ch
        mov     AL, 1        ;Код завершения
        int     21h

myproc   endp

;Подля данных резидентной части программы
mes1     db      10, 13, 'Резидентный обработчик'
        db      'загружен', 10, 13
mes2     db      27, '[2J]', 27, '[12, 22H]', 27, '[31; 40m]'
        db      'Резидентный обработчик уже загружен'
        db      27, '[13, 25H]'
        db      'Повторная установка запрещена'
        db      27, '[0m]'

```

**Задача 10.5.** Проверка на повторную загрузку с помощью мультиплексного прерывания 2Fh. Из функций этого прерывания, предназначенных для прикладных программ, произвольно выбрана функция C8h, причем в соответствии с общепринятым интерфейсом для проверки на повторную установку используется подфункция 00h. Поскольку в задаче исследуются только средства проверки на повторную загрузку, содержательная часть резидентной программы полностью отсутствует.

```

;Основные фрагменты программы
text     segment 'code'
        assume  CS:text, DS:text
        org     256          ;Программа
                                ;типа .COM!

main     proc
        jmp     init
old_2fh  dd      0
new_2fh  proc
;Обработчик мультиплексного прерывания
        cmp     AH, 0C8h      ;Наша функция прерывания 2Fh?
        jne     out_2fh       ;Не наша, на выход
        cmp     AL, 00h       ;Подфункция проверки на повторную
                                ;установку?
        jne     out_2fh       ;Неизвестная подфункция, на выход
        mov     AL, 0FFh      ;Программа уже установлена
        iret
out_2fh: jmp     CS:old_2fh    ;Переход по цепочке обработчиков
2Fh      endp
new_2fh  endp
entry:   ;Здесь могла бы быть рабочая секция резидентной программы
        iret
main     endp
end_res=$

;Процедура инициализации
init     proc
;Проверим, не установлена ли уже данная программа
        mov     AH, 0C8h
        mov     AL, 0
        int     2Fh
        cmp     AL, 0FFh
        je      installed
;Сохраним вектор мультиплексного прерывания 2Fh в ячейке old_2fh
...
;Заполним вектор мультиплексного прерывания 2Fh адресом new_2fh
...
;Выведем на экран информационное сообщение mes функцией 09h DOS
...
;Завершим программу, оставив ее резидентной в памяти
...
installed:
;Выведем на экран информационное сообщение mes1 функцией 09h DOS
...

```



```

;Заверши программу обычным образом
mov     AX,4C01h
int     21h
;Резидентная программа загружена',10,13,'$'
mes     db
mes1    db
db      'Резидентная программа уже загружена',10,13
db      'Повторная загрузка запрещена',10,13,'$'
init     endp
text     ends
main    end
;Конец сегмента кода

```

Задача 10.6. Резидентный в памяти обработчик прерываний от клавиатуры, сцепленный с системным обработчиком. Написать программу типа .COM, анализирующую коды нажимаемых клавиш (или комбинаций клавиш) и преобразующую их заданным образом (прообраз русскоязычного драйвера). Например, коды клавиш при включенном режиме <Caps Lock> (или <Num Lock>) преобразовывать в коды псевдографических символов, служащих для изображения рамок:

Клавиши	1/1	0/2	1/3	3/4	5/5	6/6	7/7	8/8	9/9	0/0	1/-
Требуемые коды	179	180	191	192	193	194	195	196	197	217	218
Символы		+	?	!	~	т	†	-	+	]	г

Предусмотреть также "горячую клавишу" (например, <Alt>/Q либо другое сочетание) для "деактивации" обработчика и возвращения системы в исходное состояние. Отладив программу, проверить правильность ее функционирования при использовании совместно с каким-либо текстовым редактором.

;Основные фрагменты программы

```

;Определение
alt_q    equ     1000h      ;Расширенный ASCII <Alt>/q
org      100h              ;Для программы типа .COM

```

```

;Проц
proc     _init
;Обязательная первая строка

```

;Поля данных для резидентной секции программы

```

old_09h dd     0           ;Место для старого вектора
mesg    db      ' <Alt>/Q - вектор восстановлен! ',1f,0f

```

```

mesglen equ     $-mesg
boarders db 0,0,179,180,191,192,193,194,195,196,197,217,218

```

;Резидентная секция программы

```

new_09h: cll              ;Запретим прерывания
pushf                    ;Сохраним флаги для команды iret
call     CS:old_09h       ;Сначала перейдем в драйвер 09h
push     AX               ;Вернулись из него. Сохраним
push     BX               ;все требуемые регистры
push     CX
push     DX
push     BP
push     DS
push     ES
mov     AX,CS             ;Настроим регистр DS
mov     DS,AX             ;на этот сегмент

```

```

mov     AX,40h
mov     ES,AX             ;Настроим регистр ES
;Начнем обработку символа, скопировав его из кольца B10S
;буфера ввода в AX, но не копируя его при этом из буфера
mov     BX,ES:1Ah
mov     AX,ES:[BX]
cmp     AX,alt_q
je      altq              ;Введенный символ -> в AX
cmp     AH,2
jb      home              ;Нажата клавиша завершения?
;Да, на восстановление вектора
cmp     AH,12
ja      home              ;Скен-код меньше 27
;Да, на выход из прерывания
;Скен-код больше 127
;Да, на выход из прерывания
test    byte ptr ES:17h,40h ;Флаг <Caps Lock> - бит 6
jz      home              ;Нет, на выход из прерывания
;Режим <Caps Lock> включен и нажата клавиша <1/1> - <1/>
push     BX               ;BX нам понадобится
lea     BX,boarders       ;Адрес таблицы трансляции
xchg     AH,AL            ;Скен-код в AL для XLAT
xlat                    ;Трансляция
pop      BX               ;Восстановим BX
mov     byte ptr ES:[BX],AL ;Заменяем код ASCII
;в кольцевом буфере ввода

```

;Вернемся в прерванную программу

```

home:    pop     ES        ;Восстановим
pop      DS               ;сохранившиеся
pop      BP
pop      DX
pop      CX               ;ранее
pop      BX               ;регистры
pop      AX
sti
iret

```

altq:

;Заверши работу обработчика. Выведем диагностическое сообщение и восстановим вектор 09h

```

mov     AX,DS             ;Нам понадобится регистр ES
mov     ES,AX             ;Настроим его
mov     AH,13h
mov     AL,1
mov     BX,0020h
mov     CX,mesglen
mov     DH,16
mov     DL,22
mov     BP,offset mesg
int     10h

```

;Восстановим вектор "вручную", чтобы не обращаться к функциям DOS

```

mov     AX,0              ;Настроим ES на область
mov     ES,AX             ;векторов прерывания
mov     AX,word ptr old_09h ;Восстановим старое
mov     word ptr ES:24h,AX ;значение IP
mov     AX,word ptr old_09h+2 ;Восстановим старое

```

```

mov     word ptr ES:26h,AX ; значение CS
jwr     home               ; На выход из прерывания
endres  equ     $-шургов
;Секция инициализации
init:   mov     AH,35h      ;Функция получения вектора
        mov     AL,09h      ;Номер вектора
        int     21h
        mov     word ptr old_09h,BX ;Сохраним старое IP
        mov     word ptr old_09h+2,ES ;Сохраним старое CS
        mov     AH,25h      ;Функция заполнения вектора
        mov     AL,09h      ;Номер вектора
        mov     DX,offset new_09h ;Адрес нового значения в DS:DX
        int     21h
;Выведем сообщение mes о загрузке обработчика
...
;Оставим программу резидентной в памяти
...
;Поля данных
mes     db      27,'[34;46m'
        db      'Обработчик загружен и оставлен резидентным'
        db      27,'[0m',cr,lf
meslen  equ     $-mes
шургов endp

```

## 11. Некоторые проблемы разработки резидентных программ и обработчиков прерываний

### 11.1. Обзор проблем

При разработке реальных резидентных программ и обработчиков аппаратных прерываний возникает целый ряд проблем, существенного усложнения программ. Некоторые из этих проблем относятся только к резидентным обработчикам или вообще к резидентным программам; другие свойственны как резидентным, так и транзитным обработчикам аппаратных прерываний. Чтобы лучше представить возможные разновидности резидентных программ и обработчиков прерываний, мы изобразили их в виде топологической карты (рис. 11.1), где левая окружность обозначает обработчики аппаратных прерываний, а правая - резидентные программы. Тогда фигура в центре, образованная пересечением этих окружностей, относится к резидентным обработчикам, полумесяц слева - к транзитным обработчикам, а полумесяц справа - к резидентным программам, не обслуживаемым синхронно.

Некоторые из отмеченных на рис. 11.1 проблем уже обсуждались в предыдущих главах и приведены здесь для общности; другие будут рассмотрены в настоящей главе. Проведем сначала краткое обсуждение всего спектра возникающих трудностей.

Если обработчик аппаратного прерывания обслуживает нестандартную для компьютера аппаратуру (например, измерительную или управляющую), он должен представлять собой законченную программу, включающую в себя все программные элементы, необходимые для обеспечения работоспособности обслуживаемого устройства. Если, однако, прикладной обработчик прерываний пишется для стандартной аппаратуры (клавиатуры, таймера, КМОП-микросхемы, дисков и проч.), целесообразно в программу прикладного обработчика включить лишь необходимые дополнения к системному алгоритму обслуживания устройства. В этом случае прикладной обработчик надо сцепить с

системным, организовав, в зависимости от конкретных требований, прикладную обработку либо до, либо после системной (а иногда и до, и после). Вопросы сцепления обработчиков уже были рассмотрены в предыдущей главе.

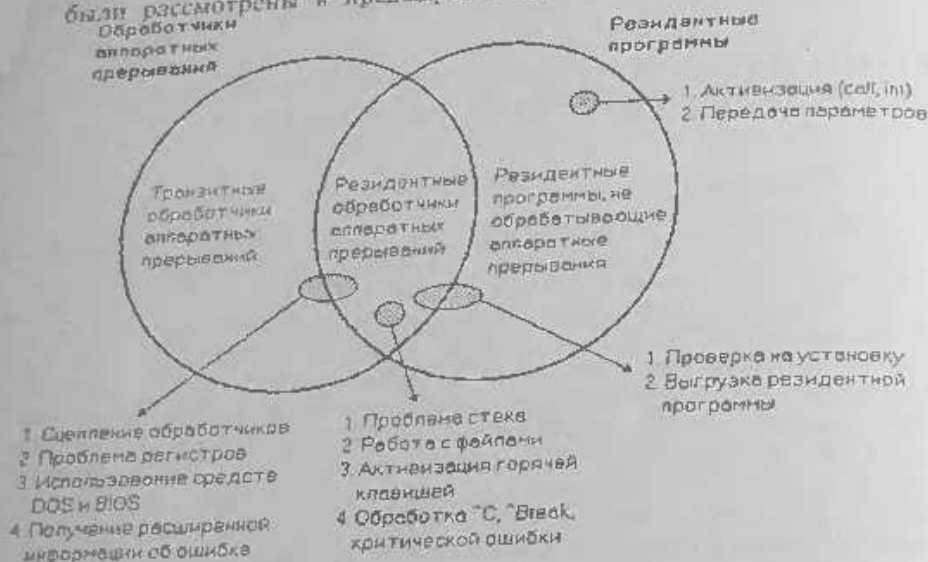


Рис. 11.1. Проблемы, возникающие при разработке обработчиков аппаратных прерываний и резидентных программ.

Для обработчиков аппаратных прерываний характерен асинхронный способ активизации: аппаратное прерывание может возникнуть в любой момент времени, и состояние программы, в частности, содержимое регистров на момент прерывания никогда не бывает известно. Процессор, выполняя процедуру прерывания, сохраняет только регистры CS:IP и флаги; все остальные регистры, если они используются в программе обработчика, необходимо при входе в обработчик сохранить, а перед выходом восстановить. Обычно сохранение осуществляется в стеке. Если обработчик является транзитным, т.е. включен в качестве составного элемента в текущую транзитную программу, используется, естественно, ее стек, общий для всей программы и для обработчика. Предвидя эту ситуацию, необходимо зарезервировать в стеке достаточно места, что нетрудно сделать на этапе разработки программы. Сложности возрастают, если обработчик является резидентным; этот вопрос будет обсужден чуть позже.

В силу асинхронного способа активизации обработчик аппарата прерывания может получить управление в тот момент, когда в основной программе выполняется запрошенная ею функция DOS или BIOS. И DOS, и BIOS состоят из нересентерабельных программ; нельзя, прервав выполнение какой-то функции DOS, вызвать ту же или другую функцию - это неминуемо приведет к разрушению системы. Поэтому без принятия специальных мер в обработчике аппаратного прерывания нецелесообразно обращаться к функциям DOS или BIOS. Это исключено в видеобуфере, однако другие ресурсы компьютера оказываются практически недоступными. Отмеченная проблема является рена ниже.

Если проблема нересентерабельности DOS преодолена и в обработчике аппаратных прерываний используются вызовы DOS, конфликт при получении расширенной информации об ошибке. Ошибка, зафиксированная системой при выполнении функции DOS в обработчике, затрет расширенную информацию об ошибке, будет, в сущности, анализировать ошибку обработчика. Таким образом, в обработчике следует предусмотреть средства сохранения и последующего восстановления "чужой" расширенной информации об ошибке.

Ситуация усугубляется, если обработчик прерываний является резидентным, т.е. не входит в состав какой-либо программы. В этом случае он существует независимо от транзитных программ и может получить управление, прервав выполнение произвольной транзитной (или даже другой резидентной) программы. При переключении из программы обработчика текущим стеком остается стек прерванной программы. Активное использование стека в обработчике может привести к переполнению стека и разрушению текущей, "ни в чем не виноватой" программы. В таком обработчике необходимо предусмотреть собственный стек, а также процедуры переключения стека сразу после входа в обработчик и перед выходом из него.

Даже если преодолеть трудности, возникающие из-за нересентерабельности DOS, в резидентном обработчике аппаратных прерываний нельзя обращаться к услугам файловой системы. Это связано с тем, что при открытии файла DOS использует для связи с системной областью файлов (SFT) таблицу файлов задания, расположенную в PSP текущей программы. Поскольку при переходе на программу обработчика текущей для системы остается прерванная программа, файлы, открываемые в



обработчике, будут использовать ее PSP (или, например, PSP программы COMMAND.COM, если в памяти нет запущенных транзитных программ). Ясно, что завершение текущей транзитной программы и запуск вместо нее другой разрушит цепочку связи с SFT и лишит обработчик возможности работать с нужным ему файлом. Таким образом, в обработчике, обращающемся к файловым функциям DOS, необходимо выполнять переключение текущей программы, для чего в DOS предусмотрены соответствующие средства.

Многие резидентные программы (электронные справочники, базы данных, калькуляторы, блокноты и проч.) должны активизироваться нажатием некоторой клавиши или комбинации клавиш. Обычно такую клавишу называют "горячей". Таким образом, программа, например, резидентного электронного справочника, сама по себе не имеющая отношения к аппаратным прерываниям, должна тем не менее перехватывать прерывания от клавиатуры и вылавливать из потока нажимаемых клавиш свои горячие клавиши, чтобы активизироваться или, наоборот, "свернуться" и перейти в пассивное состояние. Такого рода программы обычно широко используют системные средства для чтения файлов, вывода на экран и ввода с клавиатуры; всеми этими средствами нельзя воспользоваться, находясь "внутри" программы обработки аппаратного прерывания от клавиатуры.

Резидентные обработчики аппаратных прерываний обычно представляют собой относительно короткие программы, в которых не предусматривается возможность их аварийного завершения вводом с клавиатуры команд <Ctrl>/C или <Ctrl>/<Break>. Однако такая команда может быть введена случайно в том момент времени, когда фактически выполняется программа обработчика, что может привести к нарушению работы системы. Схожая ситуация возникает и при возникновении в обработчике критической ошибки. Поэтому в резидентных обработчиках обычно предусматривают перехват векторов 23h, 24h и 1Bh и собственные программы (часто просто программы-заглушки) обработки этих программных прерываний.

Выше уже отмечалось, что в любой резидентной программе должны предусматриваться средства проверки на повторную установку. Такую проверку можно осуществить разными методами, однако наиболее распространенным является использование мультиплексного прерывания 2Fh.

Другой важный вопрос, которого мы пока не касались - выгрузка ненужной более резидентной программы из памяти. В руководствах по использованию DOS, особенно, старых можно встретить утверждение, что резидентная программа может быть удалена из памяти только перезагрузкой компьютера. Это, конечно, не так, и сегодня большая часть поступающих на

рынок резидентных программ имеет средства выгрузки из памяти по команде пользователя. Более того, имеется целый класс программ (RELEASE, POPDROP и другие), которые предназначены именно для выгрузки резидентных программ с целью освобождения места в памяти. Эта операция может быть осуществлена разными, принципиально различными методами и сопряжена с определенными трудностями.

Резидентные программы, совсем не связанные с аппаратными прерываниями, используются относительно редко. В таких программах должны быть предусмотрены средства их синхронной активизации и передачи (в одну или обе стороны) параметров. Эти вопросы были рассмотрены в гл. 10.

## 11.2. Использование средств BIOS в обработчиках аппаратных прерываний

И DOS, и BIOS являются несертифицированными программами, однако их несертифицированность вызывается разными причинами и проявляется по-разному. Рассмотрим, например, прерывание 13h BIOS, закрепленное за программами управления дисками. Программный запрос на выполнение функции 02h прерывания 13h должен привести к чтению в программу указанной группы секторов жесткого или гибкого дисков. Этот запрос реализуется программами BIOS с помощью целого ряда команд, направляемых в контроллер диска: инициализации, поиска требуемой дорожки, чтения состояния контроллера, собственно чтения. При этом многие команды выполняются путем многократного обращения к контроллеру: сначала в него посылается код команды, затем - требуемые для выполнения этой команды параметры. Например, команда чтения данных требует посылки, кроме кода команды, еще 8 параметров (номеров накопителя, цилиндра, головки, сектора и т.д.).

Если в обработчике аппаратного прерывания, активизированного в момент выполнения текущей программой функции дискового прерывания 13h, в свою очередь, вызывается какая-то функция прерывания 13h, то произойдет неминуемое нарушение работы программы, так как после выполнения части команд первого запроса (например, поиска дорожки) начнут выполняться команды второго запроса (например, тоже поиск, но другой дорожки). После завершения работы обработчика и возврата в прерванную программу прерванную функцию завершить, конечно, не удастся.

В то же время прерывание дисковой функции BIOS само по себе не страшно. Если в обработчике при этом вызываются какие-то другие прерывания BIOS, например int 10h для управления экраном или int 16h для ввода с клавиатуры, нарушения

работы не произойдет. Таким образом, прерывания BIOS неперезагружаемы только по отношению к самим себе - нельзя, прерывая выполнение int 13h, вызвать в обработчике то же прерывание int 13h или, прервав работу с экраном на уровне BIOS через прерывание 10h, вызвать в обработчике функцию того же прерывания int 10h.

Итак, вызов функций BIOS возможен в обработчике только в том случае или, лучше сказать, только в те моменты времени, когда прерываемая программа не выполняет функции того прерывания, к которому мы хотим обратиться в обработчике. Но как программа обработчика узнает, не выполняется ли сейчас интересующее нас прерывание? Для этого в состав резидентного обработчика включаются секции перехвата всех прерываний BIOS, которые предполагается использовать в обработчике. Входные адреса этих секций используются в соответствующие векторы прерываний, а сами программы перехватчиков строятся так, что часть своей работы они выполняют перед системным обработчиком данного прерывания, а часть - после. Методика написания таких программ обсуждалась в гл. 9. Перехватчик прерывания BIOS перед передачей управления BIOS устанавливает флаг "занятости" данного прерывания, а после возврата из BIOS сбрасывает этот флаг. Основная же программа обработчика, перед тем, как вызывать функции BIOS, анализирует состояние этого флага и обращается к BIOS только в том случае, если флаг сброшен, т.е. данное прерывание BIOS "свободно". Программа перехватчика выглядит следующим образом:

```

capt_13h proc                ;Процедура перехватчика int 13h
flag_13h db 0                ;Флаг занятости прерывания 13h
old_13h dd 0                  ;Ячейка для хранения исходного
                               ;содержимого вектора 13h
new_13h:                      ;Точка входа в перехватчик
inc CS:flag                    ;Текущая программа вызвала int
                               ;13h, установим флаг занятости
pushf                          ;Вызовем системный обработчик
call CS:old_13h                ;Прерывания 13h с возвратом
dec CS:flag                    ;Системная обработка int 13h
                               ;закончилась, сбросим флаг
                               ;занятости
iret                           ;Назад в текущую программу
capt_13h endp                  ;Конец процедуры перехватчика int
13h

```

Перехватчики прерываний BIOS, включенные в программу обработчика аппаратного прерывания, позволяют выяснить, свободна ли в настоящий момент BIOS. Если данное прерывание BIOS сейчас не выполняется, обработчик может свободно вызывать любые функции этого прерывания. Но что делать, если BIOS оказывается занята? Тогда обработчик должен установить

флаг незавершенности и вернуть управление в прерываемую программу (фактически - в программу BIOS), так как только в этом случае BIOS может продолжить свое выполнение, что, в конце концов, приведет к ее освобождению. Теперь для того, чтобы обработчик завершил свою работу, его надо вызывать повторно до тех пор, пока он не обнаружит, что BIOS свободна. Это даст возможность обработчику вызвать требуемую функцию BIOS и, сбросив флаг незавершенности, окончательно вернуть управление в прерываемую программу. Для таких повторных вызовов обычно используются прерывания от таймера, поступающие 18,2 раза в секунду. Подробнее эта методика будет рассмотрена ниже.

### 11.3. Использование средств DOS в обработчиках аппаратных прерываний

Функции DOS, как и прерывания BIOS, неперезагружаемы, однако неперезагружаемость DOS носит иной характер. Чтобы преодолеть этого недостатка мы должны рассмотреть некоторые вопросы внутренней организации DOS.

Среди системных областей DOS имеется весьма важная структура, которую мы будем называть областью текущих данных (в оригинальной литературе она носит название области свопируемых данных - Swappable Data Area, сокращенно SDA). Это довольно большая область объемом около 2 Кбайт, адрес которой можно получить с помощью недокументированной функции DOS 5D06h:

```

mov AX, 5D06h
int 21h

```

Функция возвращает в регистрах DS:SI адрес SDA, а в регистре CX ее размер. Наиболее интересные для прикладного программиста поля SDA представлены в таблице 11.1.

Рассмотрим поля области текущих данных, имеющие отношение к разработке резидентных программ.

Флаг критической ошибки ErrorMode устанавливается DOS, если зафиксировано состояние критической ошибки, которое может возникнуть по разным причинам, в большинстве своем связанным с дисковыми операциями: попытка записи на защищенный или отсутствующий диск, на диске не найден требуемый сектор, ошибка в контрольной сумме данных в секторе и т.д. Обнаружив такую ситуацию, DOS устанавливает флаг ErrorMode, записывая в этот байт SDA 1 и выполняет команду int 24h. В этом векторе (содержимое которого, кстати



дублируется в PSP каждой запускаясь задачи, см. табл. 8.1) хранится адрес системного обработчика критической ошибки, который выводит на экран аварийное сообщение и вводит ответную команду пользователя.

Таблица 11.1. Некоторые поля области текущих данных SDA.

Смещение	Число байтов	Назначение
00h	1	Флаг критической ошибки (флаг ErrorMode)
01h	1	Флаг занятости DOS (флаг InDOS)
02h	1	Дисковод, на котором зафиксирована последняя критическая ошибка, или FFh
03h	1	Устройство, на котором зафиксирована последняя ошибка
04h	2	Код расширенной ошибки для последней ошибки
06h	1	Предлагаемое действие для исправления последней ошибки
07h	1	Класс последней ошибки
08h	4	Содержимое ES:DI при последней ошибке
0Ch	4	Адрес текущей области дисковой передачи DTA
10h	2	Сегментный адрес текущего PSP (ID текущей программы)
12h	2	Ячейка для хранения SP при вызове int 23h
14h	2	Код возврата завершившегося процесса
16h	1	Текущий дисковод
17h	1	Расширенный флаг BREAK
30h	1	День месяца
31h	1	Месяц
32h	2	Год-1980
34h	2	Номер дня от 01.01.1980
36h	1	День недели (0=воскресенье, 1=понедельник, ...)
264h	4	Прошлый кадр стека
200h	4	Позапрошлый кадр стека
336h	330	Вспомогательный стек (для функций 01h...0Ch при наличии критической ошибки)
480h	384	Дисковый стек (для функций 00h, 00h...6Ch)
600h	384	Стек ввода-вывода (для функций 01h...0Ch)

Флаг занятости DOS, обычно называемый флагом InDOS ("внутри DOS"), устанавливается диспетчером DOS сразу после анализа номера вызванной функции DOS, и сбрасывается перед возвратом из DOS в прикладную программу. Таким образом, значение этого флага, равное 1, говорит о том, что выполняется программа DOS. Функции DOS в прикладной программе можно вызывать, только флаг InDOS сброшен.

Сегментный адрес текущего PSP (смещение 10h) является важнейшей системной переменной. В этом поле записывается адрес PSP (идентификатор ID) той программы, которую DOS считает текущей. Именно эта программа будет завершена, если

выдать запрос на выполнение функции 4Ch (независимо от того, какая программа выдала этот запрос). Далее, при создании JET текущей программы, опять же независимо от того, какая программа реально выдала запрос на работу с файлом. Понятие текущей программы приобретает особую важность при разработке обработчиков аппаратных прерываний. Действительно, при вызове обработчика изменяются только значения CS:IP, текущей же остается прерванная программа. Поэтому, например, дескрипторы файлов, открытых в обработчике, находятся не в PSP обработчика, а в PSP прерванной программы, и при завершении этой программы работа обработчика (если он, будучи резидентным, остался в памяти) будет нарушена.

Адрес текущей области дисковой передачи (Disk Transfer Area, DTA) важен в тех случаях, когда в обработчике аппарата прерывания вызываются функции DOS, использующие эту область. Раньше, когда для работы с файлами использовались блоки управления файлами (FCB), с помощью DTA осуществлялись все файловые операции; в настоящее время эта область нужна только при выполнении функций поиска файлов.

В области текущих данных находятся и все три системных стека - стек ввода-вывода, который используется DOS при выполнении функций ввода-вывода из диапазона 01h...0Ch, дисковый стек для всех остальных функций DOS (файловых, службы времени, управления памятью и процессами и др.) и вспомогательный стек, на который DOS переключается в том случае, если в процессе обработки критической ошибки возникла необходимость обратиться к функциям ввода-вывода (главным образом через терминал, т.е. к функциям ввода с клавиатуры и вывода на экран). Наличие внутренних стеков повышает надежность работы DOS, так как устраняется возможность переполнения стека прикладной программы при выполнении запросов к DOS, которые в своем большинстве активно используют стек для вызова системных подпрограмм, сохранения текущих значений и передачи параметров.

Диспетчер DOS, получив управление в результате выполнения команды int 21h, совершает целый ряд операций, из которых прежде всего надо отметить следующие:

- сохранение регистров задачи на стеке задачи;
- инкремент флага InDOS, который при первом (невложенном) вызове DOS становится равен 1;
- переносит прошлый кадр стека (SS:SP) в ячейку для позапрошлого кадра, что в данном случае не имеет особого значения;



- сохраняет в ячейке для прошлого кадра стека SS:SP прерванной задачи;

- заносит в SS:SP кадр одного из системных стеков (в соответствии с вызванной функцией)

- вызывает по номеру функции требующую программу DOS.

После завершения вызванной функции диспетчер выполняет описанные шаги в обратном порядке: восстанавливает кадр стека задачи и кадр прошлого стека, декрементирует флаг InDOS, который опять становится равен 0, восстанавливает из стека задачи ее регистры и выполняет команду `iret` возврата в прерванную задачу.

Нереентерабельность DOS связана с тем, что при выполнении большинства функций используется один и тот же системный стек. Если выполнение какой-либо функции DOS будет прервано аппаратным прерыванием, и в обработчике прерывания будет вызвана функция DOS из той же группы, т.е. использующая тот же стек, то диспетчер загрузит в SS:SP в точности те же значения, что и при первом вызове. В результате вложенный вызов будет затирать те данные, которые были сохранены в стеке первым вызовом. Однако при вызове функции из другой группы ничего страшного не произойдет, так как функции ввода-вывода и "дисковые" работают на разных системных стеках. При этом наличие в SDA двух ячеек для хранения не только прошлого, но и позапрошлого кадров стека, позволяет правильно обрабатывать один вложенный вызов DOS. Эта возможность широко используется в "всплывающих" программах, активизируемых с клавиатуры.

Фактический адрес флага занятости DOS можно получить с помощью документированной функции DOS 34h. Она возвращает двухсловный адрес флага InDOS в регистрах ES:BX. Получив и сохранив этот адрес на этапе инициализации обработчика прерывания, в самом обработчике перед вызовом функций DOS следует выполнить проверку флага занятости. Будем считать, что адрес флага сохранен в ячейке `in-dos`:

```
in_dos  dd      0          ;Адрес флага InDOS
task_req db      0        ;Флаг требования запуска task

...
dos_busy: les     BX,CS:in_dos ;Получили адрес флага InDOS
          cmp     ES:[BX],0    ;DOS свободна?
          jne     wait_dos     ;Нет, придется ждать
          call    task         ;Да, можно вызывать функции DOS
          iret                ;Завершим обработчик
wait_dos inc     CS:task_req    ;Установим флаг требования запуска
          iret                ;Завершим обработчик
```

В этом фрагменте предполагается, что процедура `task` как раз и содержит вызовы DOS. Если флаг InDOS сброшен,

вызывается эта процедура и после ее завершения завершается и весь обработчик. Если же флаг InDOS установлен, обработчик устанавливает флаг `task_req`, который говорит о том, что обработчик из-за занятости DOS "недовыполнил" свои функции, и процедура `task` требует своего запуска. Управление возвращается в прерванную задачу (фактически - в завершить свою работу. DOS) и DOS имеет возможность

Каким же образом можно теперь запустить процедуру `task`? Для этого нужен дополнительный "активизатор" нашего обработчика. В качестве такого активизатора естественно воспользоваться прерываниями от таймера. Таким образом, обработчик, помимо своей основной точки входа, должен иметь еще точку входа для обработки прерываний от системного таймера:

```
in_dos  dd      0          ;Адрес флага InDOS
task_req db      0        ;Флаг требования запуска task
old_08h dd      0        ;Ячейка для хранения исходного
                           ;содержимого вектора 08h

new_08h: pushf
          call   CS:old_08h ;Передадим управление в системный
          cmp    CS:task_req;обработчик прерывания от таймера
          jne    out_08h   ;Процедура task требует запуска?
          les     BX,CS:in_dos ;Нет, можно завершить обработку
          cmp     ES:[BX],0   ;Да, снова выясним
          jne     out_08h    ;свободна ли DOS?
          dec     CS:task_req ;Нет, придется опять ждать
                           ;Да, сбросим флаг требования
                           ;запуска
          call    task       ;и вызовем task
out_08h: iret                ;Завершим обработчик
```

Проверка занятости DOS по состоянию флага InDOS необходима, но не достаточна в тех ситуациях, когда возможно возникновение критической ошибки. Дело в том, что обнаружив критическую ошибку, DOS выполняет декремент флага InDOS и инкремент флага критической ошибки `ErrorMode`, находящегося, как и флаг InDOS, в области текущих данных. После этого DOS с помощью прерывания `int 24h` вызывает обработчик критической ошибки, который выводит на экран запрос, соответствующий сложившейся ситуации, например,

```
Not ready reading drive A
Abort, Retry, Fail?
```

и ждет указаний пользователя. И вывод на экран, и ввод с клавиатуры осуществляются с помощью функций DOS из диапазона `01h...0Ch`, причем они вызываются "изнутри" той функции DOS, в процессе выполнения которой возникла критическая ошибка. Диспетчер DOS перед переключением стека

проверяет состояние флага критической ошибки и если этот флаг установлен, делает текущим не стек ввода-вывода, как обычно, а вспомогательный стек. В результате такой вложенный вызов DOS не приводит к разрушению системы.

Таким образом, в обработчике аппаратного прерывания перед вызовом какой-либо функции DOS следует анализировать состояние не только флага InDOS, но и флага ErrorMode. Если хотя бы один из них не равен 0, вызывать DOS нельзя.

В версиях DOS отсутствуют документированные средства для определения адреса флага критической ошибки. Известно, однако, что в DOS начиная с версии 3.10 он расположен в первом байте области текущих данных перед флагом InDOS, и для получения его адреса можно воспользоваться недокументированной функцией 5D06h. Поскольку сама эта функция не является рендерябельной, определение адреса следует выполнить на этапе инициализации обработчика (вместе с определением адреса флага InDOS).

С учетом сказанного, приведенный выше фрагмент проверки занятости DOS будет выглядеть следующим образом (предполагается, что адрес флага критической ошибки помещен в ячейку crit\_err):

```
in_dos  dd      0          ;Адрес флага InDOS
crit_err dd      0          ;Адрес флага ErrorMode
task_req db      0          ;Флаг требования запуска task

...
dos_busy: les     BX,CS:in_dos    ;Получили адрес флага InDOS
          lds     SI,CS:crit_err  ;Получили адрес флага ErrorMode
          cmp     ES:[BX],0        ;Флаг InDOS сброшен?
          jne     wait_dos        ;Нет, придется ждать
          cmp     DS:[SI],0        ;Флаг ErrorMode сброшен?
          jne     wait_dos        ;Нет, придется ждать
          call    task            ;Да, оба флага сброшены,
                                ;можно вызывать функции DOS
          iret                    ;Завершим обработчик
wait_dos inc     CS:task_req      ;Установим флаг требования запуска
          iret                    ;Завершим обработчик
```

Аналогичную коррекцию следует ввести и в процедуру обработки прерываний от таймера.

Описанная методика годится далеко не для всех программ. Если текущая программа ждет ввода с клавиатуры, она не выходит из соответствующей функции DOS и флаг занятости DOS непрерывно установлен. То же получится, если текущей программы вообще нет, так как в этом случае активным является командный процессор COMMAND.COM, который ждет ввода с клавиатуры очередной команды пользователя (функцией DOS 0Ah). В такой ситуации наш обработчик никогда не сможет вызвать требуемую функцию DOS. Для преодоления этого

затруднения в DOS включено специальное прерывание int 28h, вызываемое функциями ввода с клавиатуры. Выполнение, как показано на рис. 11.2.

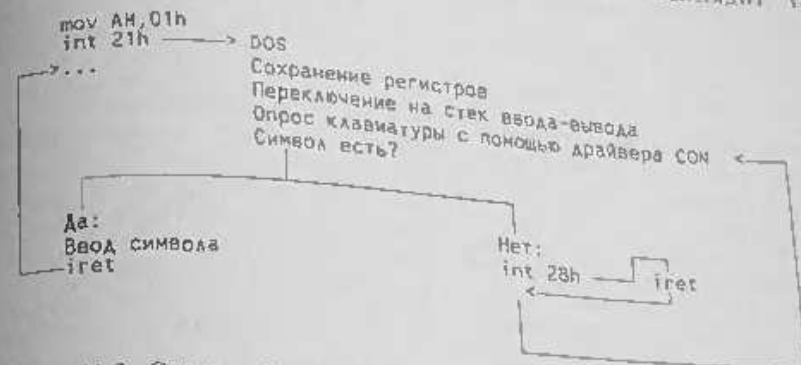


Рис. 11.2. Системный алгоритм выполнения функции ввода с клавиатуры.

Системный обработчик прерывания 28h содержит единственную команду iret, поэтому вызов int 28h при выполнении функции ввода-вывода никак не нарушает ход программы. Однако прикладная программа может поместить в вектор 28h адрес своей процедуры обработки этого прерывания. Поскольку прерывание 28h возникает только при выполнении функций, использующих стек ввода-вывода, в обработчике прерывания 28h. Надо только иметь в виду, что при входе в обработчик 28h все регистры заполнены системными значениями, и для выполнения прикладных функций их надо соответствующим образом инициализировать.

Таким образом, прикладной обработчик аппаратных прерываний, в котором вызываются функции DOS, должен включать, помимо активизатора по прерываниям от таймера (через вектор 08h), еще и активизатор по прерыванию 28h со схожим алгоритмом:

```
in_dos  dd      0          ;Адрес флага InDOS
crit_err dd      0          ;Адрес флага ErrorMode
task_req db      0          ;Флаг требования запуска task
old_28h dd      0          ;Ячейка для хранения исходного
                                ;содержимого вектора 28h

new_28h: cmp     CS:task_req,1    ;Процедура task требует запуска?
          jne     out_28h        ;Нет, можно завершить обработку
          les     BX,CS:in_dos    ;Да, получим адрес флага InDOS
          lds     SI,CS:crit_err  ;и адрес флага ErrorMode
```

```

cmp     ds:[SI],0      ;Флаг ErrorMode сброшен?
jne     out_28h        ;Нет, придется ждать
cmp     es:[BX],1      ;Флаг InDOS не больше 1?
ja      out_28h        ;Больше, вложенный вызов DOS,
                        ;вызов DOS запрещен
dec     cs:task_req     ;Сбросим флаг требования запуска
call    task           ;и вызовем task
out_28h jmp     cs:old_28h ;Завершим обработчик

```

Прикладной обработчик прерывания 28h (включенный в состав обработчика аппаратного прерывания) прежде всего проверяет, установлен ли флаг запроса запуска задачи. Если этот флаг сброшен, надо просто вернуться в прерывную функцию DOS, для чего в простейшем случае можно выполнить команду `iret`. Если, однако, в системе имеется несколько обработчиков прерывания 28h, такое завершение лишит эти обработчики возможности фиксировать прерывание 28h. Поэтому наш обработчик лучше завершить командой `jmp CS:old_28h` передачи управления на тот обработчик, адрес которого мы вытеснили из вектора 28h.

Если флаг запроса запуска задачи установлен, сначала проверяется состояние флага критической ошибки, и, если он сброшен, то и состояние флага занятости DOS. Этот флаг должен быть равен 1 (ведь сейчас выполняется функция ввода с клавиатуры). Если значение флага  $>1$ , это свидетельствует о том, что "внутри" прерывания 28h уже вызвана функция DOS, и вызывать функции DOS, увеличивая тем самым уровень вложенности DOS, нельзя. Бывают ситуации (например, при использовании программы Norton Commander), когда в обработчике `int 28h` флаг `InDOS` оказывается равен не 1, а 0. Проверка значения флага на условие  $>1$  учитывает и эту возможность.

Наконец, после успешного прохождения всех проверок вызывается процедура `task` с вызовами DOS, после завершения которой управление передается предыдущему обработчику `int 28h`.

Вызов `int 28h` выполняется функцией DOS на стеке ввода-вывода. Поэтому в прикладном обработчике прерывания 28h можно вызывать только функции "дисковой" группы. Кроме того, недопустим вызов файловых функций с указанием стандартных дескрипторов клавиатуры или экрана (0...2). Каким же образом в таком обработчике можно выполнить ввод-вывод через терминал? Для этого следует "открыть" терминал, как файл, и полученный от DOS дескриптор использовать для операций ввода-вывода:

```

fname   db      'CON',0      ;Имя консоли в формате файловых
                                ;функций
handle   dw      0           ;Ячейка для дескриптора консоли

```

```

...
mov     ah,30h
mov     al,2
mov     dx,offset fname     ;Функция открытия файла
int     21h                 ;Режим ввода и вывода
mov     handle,ax           ;Адрес имени файла
;Введен сообщение с клавиатуры
mov     ah,3Fh
mov     bx,handle           ;Сохраним дескриптор консоли
mov     cx,80
;Файловая функция ввода
;Дескриптор консоли
;Предельное число вводимых
;Адрес буфера
mov     dx,offset buf_in
int     21h
;Выведен сообщение
mov     ah,40h
mov     bx,handle           ;Файловая функция вывода
mov     cx,mes_len         ;Дескриптор консоли
mov     dx,offset mes       ;Длина сообщения
int     21h                 ;Адрес сообщения

```

Другой способ организовать ввод-вывод через терминал в обработчике прерывания 28h - воспользоваться функциями BIOS. Задача 11.1. Использование средств DOS в обработчике аппаратных прерываний. В приводимой ниже программе обработчик прерываний от клавиатуры анализирует скен-код нажатой клавиши и при поступлении кода "серого плюса" активизирует процедуру `task`, которая получает из системы текущее время и после преобразования в символьную форму выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран - функцией DOS 09h. Основная программа в бесконечном цикле выводит на экран некоторое сообщение функцией DOS 40h.

В обработчике прерываний от клавиатуры предусмотрена проверка состояния флага `InDOS` и, если DOS свободна - запуск `task`, а если DOS занята - установка флага запроса запуска и выход из прерывания. В последнем случае повторные попытки активизации `task` осуществляются с помощью обработчика прерываний от таймера, который, обнаружив установленный флаг запроса запуска, в свою очередь проверяет флаг `InDOS` и либо завершается (если DOS занята), либо запускает `task` (если DOS свободна). С целью упрощения программы в ней не проверяется состояние флага критической ошибки.

Для того, чтобы аккуратно завершить задачу, используется обработчик прерывания от `<Ctrl>/C` (вектор 23h). В этом обработчике выполняется единственная команда перехода на секцию завершения программы. В процессе завершения программы восстанавливаются векторы 08h и 09h и выполняется функция завершения 4Ch.



Отладьте программу и убедитесь в ее правильной работе. Нажатие клавиши "серый плюс" должно приводить к выводу в правый нижний угол экрана текущего времени. Нажатие <Ctrl>/C должно завершать задачу.

Выполните исследования подготовленной программы. Рассмотрите реакцию программы на нажатие других клавиш. Обратите внимание, что после нажатия любой клавиши перестает работать <Ctrl>/C (программа не забирает коды из кольцевого буфера, и они "закрывают" введенное позже <Ctrl>/C от DOS). В этом случае используйте для завершения программы <Ctrl>/<Break> - эта комбинация обслуживается прикладной программой через тот вектор 23h, но нажатие <Ctrl>/<Break> очищает кольцевой буфер клавиатуры и стирает ранее введенные в него символы.

Исключите из обработчика прерываний от клавиатуры строки проверки флага InDOS:

```

; cmp     byte ptr ES:[BX],0 ;DOS свободна?
; jne     wait_dos          ;Нет, придется ждать

```

Убедитесь, что нажатие горячей клавиши приводит к разрушению DOS. Увеличьте продолжительность задержки между вызовами int 40h в основной программе. Если нажать горячую клавишу, когда длится задержка, разрушения системы не будет даже при отсутствии проверки флага InDOS

```

text     segment 'code'
        assume CS:text,DS:data

```

```

;Подпрограмма преобразования 2х-разрядного двоичного числа
;в десятичное число в ASCII-представлении
;На входе AL - число
;BX - адрес буфера для ASCII-представления

```

```

binasc   proc
xor       AH,AH                ;Очистим AH. Теперь число в AX
div       ten                  ;Разделим на 10. AL=десятки,
                                ;AH=единицы
        add     AL,'0'         ;Преобразуем AL в символьную форму
        mov     [BX],AL        ;Зашлем в строку
        add     AH,'0'         ;Преобразуем AH в символьную форму
        mov     I[BX],AH       ;Зашлем в строку
        ret
binasc   endp

```

;Обработчик прерываний от клавиатуры

```

new_09h  proc
push     AX                    ;Сохраним
push     BX                    ;используемые
push     ES                    ;регистры
in       AL,60h                ;Введем скен-код
cmp      AL,4Eh                ;Серый плюс?
je       plus                  ;Да, на продолжение

```

```

pop      ES
pop      BX
pop      AX                    ;Нет, восстановим
                                ;сохраненные
                                ;регистры
plus:    jmp     CS:old_09h     ;и вызовем системный обработчик
in       AL,60h                ;Выполним волшебные действия
or       AL,80h                ;с контроллером клавиатуры,
                                ;чтобы снять скен-код нажатой
                                ;клавиши и разрешить
                                ;дальнейшую работу контроллера
out      AL,7Fh                ;Пошлем в контроллер прерываний
out      60h,AL                ;сигнал EOI
mov      AL,20h                ;Получим адрес флага InDOS
les      BX,CS:in_dos           ;DOS свободна?
cmp      byte ptr ES:[BX],0
jne      wait_dos              ;Нет, придется ждать
call     task                   ;Да, вызовем процедуру task
jmp      out_09h               ;На выход из обработчика
wait_dos:inc                    ;Установим флаг запроса запуска
out_09h:mov                     ;Пошлем в контроллер прерываний
        out     20h,AL          ;сигнал EOI
        pop     ES              ;Восстановим
        pop     BX              ;сохраненные
        pop     AX              ;регистры
        iret                    ;Возврат в прерванную программу
new_09h  endp

```

;Обработчик прерываний от таймера

```

new_08h  proc
push     BX                    ;Сохраним используемые
push     ES                    ;регистры
pushf
call     CS:old_08h            ;Перейдем в системный обработчик
cmp      CS:task_req,1         ;с возвратом
jne      out_08h               ;Флаг запроса запуска установлен?
les      BX,CS:in_dos          ;Нет, не выход из обработчика
cmp      byte ptr ES:[BX],0    ;Да, получим адреса флага InDOS
jne      out_08h               ;Флаг InDOS сброшен?
dec      CS:task_req           ;Нет, на выход из обработчика
call     task                   ;Да, сбросим флаг запроса запуска
out_08h: pop     ES              ;и вызовем task
        pop     BX              ;Восстановим сохраненные
        iret                    ;регистры
        ;и вернемся в прерванную программу
new_08h  endp

```

;Обработчик прерываний по <Ctrl>/C и <Ctrl>/<Break>

```

new_23h  proc
jmp      home                  ;На завершение программы
new_23h  endp

```

;Основная программа

```

турproc  proc
mov      AX,data
mov      DS,AX
;Выполним подготовительные действия
;Получим адрес флага InDOS

```

```

mov     AH,34h
int     21h
mov     word ptr CS:in_dos,BX
mov     word ptr CS:in_dos+2,ES
;Сохраним вектор прерывания от таймера 08h в ячейке old_08h
mov     AX,3508h
int     21h
mov     word ptr CS:old_08h,BX
mov     word ptr CS:old_08h+2,ES
;Заполним вектор 08h адресом нашего обработчика new_08h
mov     AX,2508h
push    DS
push    CS
pop     DS
mov     DX,offset new_08h
int     21h
pop     DS
;Сохраним вектор прерывания от клавиатуры (09h) в ячейке old_09h
...
;Заполним вектор 09h адресом нашего обработчика new_09h
...
;Сохраним вектор прерывания по <Ctrl>/C (23h) в ячейке old_23h
...
;Заполним вектор 23h адресом нашего обработчика new_23h
...
;Будем периодически выводить сообщение
;Сначала line1 с возвратом курсора в начало строки
go:     mov     AH,40h
        mov     BX,1
        mov     CX,line1_len
        mov     DX,offset line1
        int     21h
;Ведем небольшую задержку
        mov     CX,0
tttt:   loop    tttt
;Затем line2 с возвратом курсора в начало строки
        mov     AH,40h
        mov     CX,line2_len
        mov     DX,offset line2
        int     21h
;Ведем небольшую задержку
        mov     CX,0
ssss:   loop    ssss
        jmp     go
;Бесконечный цикл

```

;Процедура task, активизируемая горячей клавишей  
и использующая функции DOS

```

task    proc
        push    AX
        push    BX
        push    CX
        push    DX
        push    DS
        push    ES
;Сохраним все
;регистры,
;используемые
;в task
;и в binasc

```

```

mov     AX,data
mov     DS,AX
;Получим время с помощью функции DOS
mov     AH,2Ch
int     21h
;Преобразуем часы
mov     AL,CH
mov     BX,offset time
call    binasc
;Преобразуем минуты
mov     AL,CL
lea     BX,time+3
call    binasc
;Преобразуем секунды
mov     AL,DH
lea     BX,time+6
call    binasc
;Выведем всю строку функцией DOS
mov     AH,09h
mov     DX,offset string
int     21h
pop     ES
pop     DS
pop     DX
pop     CX
pop     BX
pop     AX
ret

```

;Настроим DS на наш  
;сегмент данных

;Функция получения времени

;Часы в AL  
;Адрес поля в BX  
;Вызовем binasc

;Минуты в AL  
;Адрес поля в BX  
;Вызовем binasc

;Секунды в AL  
;Адрес поля в BX  
;Вызовем binasc

;Восстановим  
;все регистры

;Вернемся в тот наш обработчик,  
;откуда была вызвана task

```

task    endp
home:
;Завершим программу, восстановив сначала все перехваченные векторы
;Восстановим вектор 09h

```

```

        mov     AX,2509h
        lds     DX,CS:old_09h
        int     21h
;Восстановим вектор 08h

```

```

        ...
;Завершим программу

```

```

        ...
murgos  endp

```

;Поля данных, размещенные для удобства адресации в сегменте команд

```

old_09h dd 0
old_08h dd 0
old_23h dd 0
in_dos  dd 0
task_req db 0

```

```

text    ends

```

;Поля данных, размещенные в сегменте данных

```

string  db 27,'[s',27,'[25;71H',27,'[34;41m'
time    db 27,'[00:00:00'
        db 27,'[0m'
        db 27,'[u'

```

```

ten      db      '$'
line1    db      10
line1    db      40 dup ('-'),13
line1    len=$-line1
line2    db      40 dup (' '),13
line2    len=$-line2

```

Задача 11.2. Данная программа является слегка измененным вариантом предыдущей. Здесь в основной программе для вывода на экран используется функция из диапазона 01h...0Ch (конкретно 02h), а в процедуре task - только функции из диапазона 0Dh...6Ch (конкретно 40h и 2Ch). В этом случае устранение строк проверки флага InDOS не нарушают работоспособности задачи. (Фактически в такой программе отпадает необходимость во всех средствах, так или иначе связанных с проверкой занятости DOS: получении адреса флага InDOS и его анализа, перехвата прерываний от таймера, работы с флагом запроса запуска. Однако проще только изменить функции вывода и удалить две строки проверки флага InDOS в обработчике прерываний от клавиатуры).

;Измененные фрагменты программы

new\_09h proc

;Обработчик прерываний от клавиатуры

```

plus:    les      BX,CS:in_dos    ;Получим адрес флага InDOS
;        cmp      byte ptr ES:[BX],0 ;DOS свободна?
;        jne      wait_dos        ;Нет, придется ждать
;        call     task            ;Да, вызовем процедуру task

```

new\_09h endp

;Будем периодически выводить сообщение

```

go:      mov      AH,02h
;        mov      DL,'_'
;        int      21h

```

```

tttt:    loop     tttt
;        mov      AH,02h
;        mov      DL,'|'
;        int      21h

```

```

ssss:    mov      CX,0
;        loop     ssss
;        jmp      go

```

```

task     proc

```

;Процедура, активизируемая горячей клавишей и использующая функции DOS

;Выведем всю строку функцией DOS

```

;        mov      AH,40h
;        mov      BX,1
;        mov      CX,len
;        mov      DX,offset string
;        int      21h

```

```

;Поля данных
string    db      27,'[s',27,'[25;71h',27,'[34;41h'
time      db      '00:00:00'
;        db      27,'[0m'
;        db      27,'[u'
len=$-string

```

Задача 11.3. Использование средства DOS в обработке ап-паратных прерываний, если основная программа вводит данные с клавиатуры (включение в программу обработчика прерывания int 28h).

Модифицировать программу 11.1, заменив фрагмент вывода на экран строк line1 и line2 функцией DOS 01h: фрагментом ввода символа с клавиатуры функцией DOS 01h:

```

;Будем периодически вводить сообщение
go:      mov      AH,01h
;        int      21h
;        jmp      go

```

Убедиться, что в такой редакции горячая клавиша не действует, так как программа почти непрерывно находится "внутри DOS". Добавить в программу средства обработки прерывания 28h:

;В секции подготовительных действий:

;Сохраним вектор прерывания 28h в ячейке old\_28h

```

;        ...
;Заполним вектор 28h адресом обработчика new_28h
;        ...

```

;В секции обработчиков прерываний:

;Новый обработчик прерывания 28h

```

new_28h  proc
;        push     BX
;        push     ES
;        cmp      CS:task_req,1
;        jne      out_28h
;        les      BX,CS:in_dos
;        cmp      byte ptr ES:[BX],1
;        jne      out_28h
;        dec      CS:task_req
;        call     task
;        pop      ES
;        pop      BX
;        jmp      CS:old_28h
;Сохраним используемые регистры
;Флаг запроса запуска установлен?
;Нет, на выход
;Да, получим адрес флага InDOS
;Флаг InDOS=1?
;Нет, на выход
;Да, сбросим флаг запроса запуска
;и вызовем процедуру task
;Восстановим регистры
;Перейдем в системный обработчик

```

int 28h

new\_28h endp

;Будем периодически вводить сообщение

```

go:      mov      AH,01h
;        int      21h
;        jmp      go

```



;8 секции завершения (в обработчике <Ctrl>/C);  
;Восстановим вектор 28h

#### 11.4. Асинхронная активизация резидентных программ командами с клавиатуры

Для управления резидентной программой командами, подаваемыми с клавиатуры, в частности, для активизации программы, в ее состав необходимо включить обработчик прерываний от клавиатуры (прерывание 09h). Выполнять резидентную программу, находясь "внутри" такого обработчика, можно только в том случае, если в программе не используются вызовы DOS или BIOS. Поэтому обычно в обработчике прерывания от клавиатуры лишь выполняется анализ введенной команды и установка флага запроса запуска задачи, если команда соответствует запросу на активизацию. Сама же активизация задачи осуществляется в обработчике прерывания 28h, как это было описано в предыдущем разделе. Структура секции запуска задачи (для случая активизации нажатием одной клавиши "серый плюс") может выглядеть следующим образом:

```
old_28h dd 0 ;Старое содержимое вектора 28h
old_09h dd 0 ;Старое содержимое вектора 09h
task_req db 0 ;Флаг запроса запуска задачи
...
new_09h proc
push AX ;Сохраним AX
in AL,60h ;Введем скен-код нажатой клавиши
cmp AL,4Eh ;Нажат серый плюс?
je plus ;Да, на установку флага
pop AX ;Восстановим AX
jmp CS:old_09h ;В обработчик прерывания 09h
plus: inc CS:task_req ;Установим флаг запроса запуска
pop AX ;Восстановим AX
jmp CS:old_09h ;В обработчик прерывания 09h
new_09h endp

new_28h proc
cmp CS:task_req,1 ;Задача требует активизации?
je pop_up ;Да
jmp CS:old_28h ;Нет, в обработчик прерывания 28h

pop_up:
;Проверка флагов критической ошибки и занятости DOS
;Сброс флага запроса запуска задачи
;Запуск задачи
iret
```

Задача 11.4. Активизация "всплывающей" (pop-up) программы горячей клавишей. Программа написана на базе задачи 11.2, однако со многими изменениями:

программа сделана резидентной;  
в процедуре task предусмотрено сохранение в буфере программы исходного содержимого той части экрана (8 символов), куда выводится информация о текущем времени, с целью возможности восстановления первоначального вида экрана;  
в обработчике прерываний от клавиатуры теперь фиксируется нажатие двух клавиш, "серого плюса" для активизации содержимого экрана;  
обработчик прерываний от таймера исключен, и процедура task может активизироваться только в том случае, если текущая программа ждет ввода с клавиатуры;  
в обработчике прерывания 28h проверяются состояния обоих флагов - и занятости DOS, и критической ошибки, как это и следует всегда делать.

С целью упрощения программы в ней не отрабатываются некоторые возможные ситуации, приводящие к сбоям. Так, после каждого нажатия "серого плюса" необходимо до подачи с клавиатуры каких-либо команд восстанавливать экран нажатием "серого минуса", в противном случае восстановление экрана будет выполняться неправильно. В программе не предусмотрена процедура ее выгрузки; освободиться от резидентной программы можно только перезагрузкой DOS или с помощью утилиты типа RELEASE (задачи, иллюстрирующие выгрузку резидентной программы, рассматриваются в следующем разделе).

```
text segment 'code'
assume CS:text
org 100h
init
start: jmp
old_09h dd 0
old_28h dd 0
in_dos dd 0
crit_err dd 0
task_req db 0
screen db 16 dup (0)
string db 27,'[s',27,'[12;36H',27,'[34;41m'
time db '00:00:00'
db 27,'[0m'
db 27,'[u'

len=$-string
ten db 10
handle dw 0

binasc proc
...
ret
binasc endp
```

```

new_09h proc
;Обработчик прерываний от клавиатуры
push AX
in AL,80h
cmp AL,4Eh
je plus
cmp AL,4Ah
je minus
pop AX
jmp CS:old_09h
plus: inc CS:task_req
jmp out_09h
;Выведем на экран поверх нашей строки с текущим временем
;исходной содержимое экрана из буфера screen
minus: push DS
push ES
push SI
push DI
push CS
pop DS
mov SI,offset screen
mov AX,0B800h
mov ES,AX
mov DI,11*160+35*2
mov CX,16
cld
rep movsb
pop DI
pop SI
pop ES
pop DS
out_09h: in AL,61h
or AL,80h
out 61h,AL
and AL,07Fh
out 61h,AL
mov AL,20h
out 20h,AL
pop AX
iret
new_09h endp
new_28h proc
push BX
push ES
cmp CS:task_req,1
jne out_28h
les BX,CS:crit_err
cmp byte ptr ES:[BX],0
jne out_28h
les BX,CS:in_dos
cmp byte ptr ES:[BX],1
je out_28h
;Сохраним AX
;Введем скен-код
;Серый плюс?
;Да, на продолжение
;Серый минус?
;Да, на восстановление экрана
;Нет, восстановим AX
;и вызовем системный обработчик
;Установим флаг активизации
;и на выход
;Сохраним используемые регистры
;Настроим DS
;на сегмент резидентной программы
;DS:SI->буфер screen
;Настроим ES на сегмент
;видеобуфера
;ES:DI->Строка 12, столбец 35
;8 символов, 16 байтов
;Вперед
;Пересылка
;Восстановим используемые регистры
;Прочитаем содержимое порта B
;Сначала установим бит 7 в порте
;B, а затем сбросим его, чтобы
;послать в контроллер клавиатуры
;сигнал подтверждения приема
;скен-кода
;Сигнал
;EOI
;Восстановим сохраненный ранее AX
;Возврат в прерванную программу

```

```

dec call CS:task_req
out_28h: pop task
pop ES
pop BX
jmp CS:old_28h
int_28h endp
new_28h proc
task
;Процедура, активизируемая горячей клавишей и использующая функции DOS
;Сохраним регистры
push AX
push BX
push CX
push DX
push SI
push DI
push DS
push ES
;Сохраним те 16 байтов экрана, куда будет выводиться время
push CS
pop ES
mov DI,offset screen
mov AX,0B800h
mov DS,AX
mov SI,11*160+35*2
mov CX,16
cld
rep movsb
push CS
pop DS
;Получим время с помощью функции DOS
...
;Преобразуем часы
...
;Преобразуем минуты
...
;Преобразуем секунды
...
;Выведем всю строку функцией DOS из диапазона 00h...6Ch
mov AH,40h
mov BX,1
mov CX,16
mov DX,offset string
int 21h
;Восстановим регистры
pop ES
pop DS
pop DI
pop SI
pop DX
pop CX
pop BX

```

;Нет, сбросим флаг запроса запуска  
;и вызовем процедуру task  
;Восстановим  
;регистры  
;Перейдем в системный обработчик

;Настроим ES на  
;наш сегмент  
;ES:DI-> буфер screen  
;Настроим DS  
;на видеобуфер  
;DS:SI->область, куда будет  
;выводиться текущее время  
;8 символов, 16 байтов  
;Вперед  
;Пересылка  
;Получим адресованность  
;наших данных через DS

```

        pop     AX                ;Возврат в вызывающий обработчик
        get
    task   endp

init     proc
;Выполним подготовительные действия
;Проверим версию DOS
        mov     AX,3000h        ;Функция проверки версии MS-DOS
        int     21h             ;Версия 4 или больше?
        cmp     AL,4            ;Да, можно работать
        jae     ok              ;Да, можно работать
;Версия DOS < 4, программа не предназначена для таких версий
;Выведем сообщение vermes и завершим программу функцией 4Ch
        ...
;Получим адрес флага InDOS
ok:      mov     AH,34h
        int     21h
        mov     word ptr in_dos,BX
        mov     word ptr in_dos+2,ES
;Сохраним адрес флага критической ошибки ErrorMode
        dec     BX
        mov     word ptr crit_err,BX
        mov     word ptr crit_err+2,ES
;Сохраним вектор прерывания от клавиатуры 09h
        ...
;Заполним вектор 09h
        ...
;Сохраним вектор прерывания 28h
        ...
;Заполним вектор 28h
        ...
;Выведем сообщение mes об установке программы любыми средствами DOS
        ...
;Завершим программу, оставив в памяти резидентную часть
        mov     AX,3100h
        mov     DX,(init-start+10Fh)/16
        int     21h
init     endp
mes      db      'Программа установлена',10,13
        db      'Активизация - "серый плюс"',10,13
        db      'Свертывание - "серый минус"',10,13,'$'
vermes   db      'Неправильная версия DOS$'
text     ends
end       start

```

### 11.5. Работа с файлами в резидентном обработчике аппаратных прерываний

Работа с файлами в резидентном обработчике аппаратных прерываний, как уже отмечалось, осложняется тем, что при переходе в обработчик текущей остается прерванная программа, а дескрипторы открываемых файлов создаются системой в

таблице файлов задания ее PSP. Для того, чтобы дескрипторы файлов, открываемых в обработчике, полностью принадлежали самому обработчику, необходимо при входе в обработчик объявить его для DOS текущей программой. Для манипуляций с идентификаторами программ (т.е. сегментными адресами их PSP) в DOS предусмотрены функции 51h (Получить PSP) и 50h (Установить PSP). Обе функции не используют системные стеки и являются реентерабельными, поэтому их можно безопасно вызывать в обработчиках аппаратных прерываний.

Структура обработчика, работающего с файлами, выглядит следующим образом:

```

entry:    ;Сохранение регистров прерванной программы
        ...
;Сделаем обработчик текущей программой
        mov     AH,51h
        int     21h        ;Функция получения текущего PSP
        ;BX=ID текущей (т.е. прерванной)
        ;программы
        push     BX
        mov     AH,50h
        mov     BX,CX
        int     21h        ;Сохраним ID текущей программы
        ;Функция установки текущего PSP
        ;В .COM-программе CS=ID программы

;Теперь текущей считается программа обработчика. При открытии файлов
;дескрипторы будут создаваться в JFT обработчика
        ...
;Работа с файлами обработчика (естественно, если DOS свободна)
        ...
;Сделаем прерванную программу текущей
        pop      BX
        ;Восстановим ID прерванной
        ;программы
        mov     AH,50h
        int     21h        ;Функция установки текущего PSP
;Восстановим регистры прерванной программы
        iret

```

В тех случаях, когда в обработчике вызываются файловые функции поиска файлов 4Eh и 4Fh, использующие область дисковой передачи DTA, переключения ID программы недостаточно. Действительно, при загрузке программы DOS записывает в SDA не только адрес PSP этой программы, т.е. ее ID, но и двухсловный адрес текущей DTA (см. табл. 11.1). Поэтому в обработчике следует при входе сохранить адрес текущей DTA и установить в качестве текущей DTA обработчика, а при выходе восстановить DTA прерванной программы. Для получения и установки дисковой области передачи предусмотрены функции 2Fh и 1Ah; они не являются реентерабельными, поэтому перед их использованием следует убедиться, что DOS свободна. Фрагменты обработчика, выполняющие переключения DTA, приведены ниже.



Переключение DTA перед использованием файловой функции

```

mov     AH, 2Fh      ; Функция получения текущей DTA
int     21h          ; ES:BX-адрес DTA
push    CS            ; Настроим DS на PSP программы
pop     DS            ; (если это не сделано ранее с
                    ; целью упростить адресацию данных)
push    ES            ; Сохраним сегментный адрес DTA
push    BX            ; Сохраним смещение DTA
mov     AH, 1Ah      ; Функция установки текущей DTA
mov     DX, 80h      ; Пусть DTA обработчика будет на
int     21h          ; своем исходном месте - начиная со
                    ; смещения 80h в PSP

```

Теперь можно вызывать файловые функции, использующие DTA

Переключение DTA перед завершением обработчика

```

mov     AH, 1Ah      ; Функция установки текущей DTA
pop     DX            ; Восстановим сегментный адрес DTA
pop     DS            ; Восстановим смещение DTA
int     21h

```

Как известно, DTA программы по умолчанию находится в ее PSP, начиная со смещения 80h (то же поле используется и для копирования параметров командной строки). Функция установки DTA 1Ah требует загрузки двухсловного адреса DTA в регистры DS:DX. При вызове обработчика прерывания, выполненного в формате .COM, в CS находится сегментный адрес PSP. Командами

```

push    CS
pop     DS

```

DS также настраивается на сегментный адрес PSP. Теперь достаточно занести в DX смещение 80h и можно вызывать функцию установки DTA.

При использовании в резидентном обработчике функций DOS не исключено возникновение ошибок. Код ошибки возвращается системой в регистре AX, а расширенная информация об ошибке записывается в предназначенные для этого поля SDA. Функция DOS 59h позволяет получить из SDA расширенную информацию об ошибке и программно проанализировать ее. Если прерывание, активизировавшее наш обработчик, возникло после выполнения в прерванной программе функции DOS, завершившейся с ошибкой, и, кроме того, ошибка возникла в самом обработчике, то информация об ошибке прерванной программы в SDA будет затерта новой информацией, поступившей в SDA результате ошибки в обработчике. Если прерванная программа (после возвращения в нее) вызовет функцию DOS 59h для получения расширенной информации об ошибке, она получит информацию, относящуюся к ошибке в обработчике. Поэтому в резидентном обработчике аппаратных прерываний

следует перед вызовом функций DOS получить из SDA расширенную информацию об ошибке (с помощью функции DOS 59h), а перед выходом из обработчика - восстановить ее с помощью функции 5D, подфункции 0Ah. Функция 59h возвращает расширенную информацию об ошибке в следующих регистрах:

- AX - расширенный код ошибки (табл. 11.2)
- BH - класс ошибки (табл. 11.3)
- BL - рекомендуемое действие (табл. 11.4)
- CH - местоположение ошибки (табл. 11.5)
- ES:DI - содержимое этих регистров для последней ошибки

Таблица 11.2. Расширенный код ошибки, возвращаемый функцией 59h.

Код	Значение	Код	Значение
00h	Нет ошибки	14h	Неизвестное устройство
01h	Неверный номер функции	15h	Дисковод не готов
02h	Файл не найден	16h	Неизвестная команда
03h	Путь не найден	17h	Ошибка данных (CRC)
04h	Нет свободных дескрипторов	18h	Неверная длина структуры запроса
05h	Доступ запрещен	19h	Ошибка поиска
06h	Недопустимый дескриптор	1Ah	Не DOS-диск
07h	Разрушен блок управления памятью	1Bh	Сектор не найден
08h	Нехватка памяти	1Ch	В принтере нет бумаги
09h	Недопустимый адрес блока памяти	1Dh	Ошибка записи
0Ah	Ошибка окружения	1Eh	Ошибка чтения
0Bh	Недопустимый формат	1Fh	Общий отказ
0Ch	Недопустимый код доступа	20h	Нарушение разделения
0Dh	Недопустимые данные	21h	Нарушение записи
0Eh	Недопустимый дисковод	22h	Недопустимая смена диска
0Fh	Попытка удалить текущий каталог	23h	FCB недоступен
10h	Не то же устройство	24h	Переполнение буфера разделения
11h	Нет больше файлов	25h	Несоответствие кодовой страницы
12h	Диск защищен от записи	26h	Невозможно завершить ввод из файла
13h	28h...5Ah зарезервированы и сетевые ошибки	27h	Нехватка места на диске

Таблица 11.3. Класс ошибки, возвращаемый функцией 59h.

Код	Значение	Код	Значение
01h	Нехватка ресурсов	07h	Ошибка прикладной программы
02h	Файл или запись заперты	08h	Не найдено
03h	Доступ запрещен	09h	Неверный формат
04h	Ошибка системной программы	0Ah	Заперто
05h	Отказ оборудования	0Bh	Ошибка носителя
06h	Отказ системы	0Ch	Уже существует

Таблица 11.4. Рекомендуемое действие, возвращаемое функцией 59h.

Код	Значение
01h	Повтор
02h	Задержанный повтор
03h	Запрос пользователю на повторный ввод
04h	Прекратить выполнение после очистки ресурсов
05h	Немедленно прекратить выполнение
06h	Игнорировать
07h	Повтор после вмешательства пользователя

Таблица 11.5. Местоположение ошибки, возвращаемое функцией 59h (код, возвращаемый в CH, может иметь другой смысл).

Код	Значение
01h	Неизвестное
02h	Блочное устройство (ошибка диска)
03h	Сетевая ошибка
04h	Таймаут последовательного порта
05h	Ошибка памяти

Документированная, начиная с DOS V5.0, функция 5D0Ah позволяет записать в соответствующие поля SDA расширенную информацию об ошибке, полученную предварительно с помощью функции 59h. Функция 5D0Ah требует в качестве параметра адрес массива из 11 слов (табл. 11.6), который называется списком параметров DOS. Этот список используется недокументированной функцией 5D00h для занесения в регистры значений, необходимых для выполнения сетевым сервером вызовов DOS, возникающих на других машинах. Функция 5D0Ah тоже использует этот список, однако в SDA поступают только значения, соответствующие регистрам AX, BX, DI и ES. Эти значения записываются в поля SDA со смещениями, соответственно, 04h, 06h, 08h и 0Ah (см. табл. 11.1), предназначенные для хранения расширенной информации об ошибке. Таким образом, в массиве для функции 5D0Ah достаточно заполнить поля этих регистров.

Фрагменты программы, выполняющие сохранение и восстановление расширенной информации об ошибке, могут выглядеть следующим образом:

Получим расширенную информацию об ошибке

```
mov     AH, 59h
mov     BX, 0
int     21h
```

При возврате из этой функции практически все регистры разрушены!

Следует либо восстановить значение DS, либо обращаться к полям данных через CS (в случае программы типа .COM).

Сохраним полученную информацию

```
mov     CS, AX
mov     CS, BX
mov     CS, DI
mov     CS, ES
```

Теперь можно вызывать функции DOS

Восстановим в SDA расширенную информацию об ошибке

в прерванной программе. DS должен быть настроен на сегмент с данными

```
mov     AX, 5D0Ah
mov     DX, offset _ax
```

Поля данных		
ax	dw	0
bx	dw	0
cx	dw	0
dx	dw	0
si	dw	0
di	dw	0
ds	dw	0
es	dw	0,0,0

### Задача 11.5. Работа с файлом в резидентном обработчике аппаратных прерываний.

```
text     segment 'code'
         assume CS:text
         org     100h
start:   jmp     init
old_05h  dd      0
old_28h  dd      0
in_dos   dd      0
crit_err dd      0
task_req db      0
handle   dw      0
screen   db      2000 dup (?)
dname    db      'f:\current\'
nulbyte  db      0
new_05h  db      12 dup (0)
new_05h  proc
         mov     CS, task_req, 1
         iret
new_05h  endp
new_28h  proc
         .
         .
         .
new_28h  endp
task     proc
;Процедура, активизируемая клавишей Print Screen и работающая с файлами
         push    AX
         push    BX
```

```
;Обработчик прерываний от клавиши
;PrintScreen
;Установим флаг активизации
```

```
;В этом обработчике
;нет никаких изменений
;по сравнению с предыдущей задачей
```

```
         push    AX
         push    BX
;Сохраним все используемые
;в процедуре регистры
```

```

push    CX
push    DX
push    SI
push    DI
push    DS
push    ES
mov     AX, 0B800h
mov     DS, AX
mov     SI, 0
mov     CS
push    ES
pop     DI, offset screen
mov     CX, 2000

abcd:   old
        lodsw
        stasb

        loop abcd
;Сбросим буфер программы в файл
        push CS
        pop  DS
;Объявим на время нашу резидентную программу текущей
        mov  AH, 51h
        int  21h
        push BX
        mov  AH, 50h

        mov  BX, CS
        int  21h
;Создадим временный файл
        mov  AH, 5Ah
        mov  DX, offset dname
        mov  CX, 0
        int  21h
        mov  handle, AX
;Выведем содержимое буфера в созданный файл
        mov  AH, 40h
        mov  BX, handle
        mov  CX, 2000
        mov  DX, offset screen
        int  21h
;Закроем файл
        mov  AH, 3Eh
        mov  BX, handle
        int  21h
;Уберем из спецификации пути имя только что созданного файла,
;которое туда поместила DOS
        mov  nulbyte, 0
;Восстановим ID прерванной программы
        pop  BX

```

;Настроим DS на адрес  
;видеобуфера  
;DS:SI->видеобуфер  
;Настроим ES  
;на наш сегмент  
;ES:DI->буфер screen  
;Столько символов читать из  
;видеобуфера  
;Вперед  
;Заберем из видеобуфера  
;символ+атрибут  
;Сохраним в буфере программы  
;только символ  
;Цикл из 2000 шагов  
;Настроим DS на наш  
;сегмент  
;Получим ID прерванной  
;программы (он сейчас в SDA)  
;Сохраним ID прерванной программы  
;Установим в SDA ID нашей  
;программы  
;Наш ID в CS  
;Функция создания временного файла  
;Адрес пути к файлу  
;Атрибуты файла (отсутствуют)  
;Сохраним дескриптор нового файла  
;Функция закрытия файла  
;Дескриптор

```

mov     AH, 50h
int     21h
;Функция установки PSP
;Восстановим сохраненные регистры и вернемся из подпрограммы task
        pop  ES
        pop  DS
        pop  DI
        pop  SI
        pop  DX
        pop  CX
        pop  BX
        ret
        endp

task:   proc
        init
;Выполним подготовительные действия
;Проверим версию DOS
        ...
;Версия DOS < 4, программа не предназначена для таких версий
        ...
;Получим адрес флага InDOS
        ...
;Сохраним адрес флага критической ошибки ErrorMode
        ...
;Сохраним вектор прерывания 05h
        ...
;Заполним вектор 05h
        ...
;Сохраним вектор прерывания 26h
        ...
;Заполним вектор 26h
        ...
;Выведем сообщение об установке программы
        ...
;Завершим программу, оставив в памяти резидентную часть
        mov  AX, 3100h
        mov  DX, (init-start+10Fh)/16
        int  21h
        endp
init    db    'Программа установлена', 10, 13
res     db    'Неправильная версия DOS'
vermes
text    ends
        end    start

```

## 11.6. Выгрузка из памяти резидентных программ

При разработке резидентных программ обычно предусматривают средства выгрузки их из памяти, если отпала необходимость в их использовании. Следует заметить, что в DOS вообще отсутствуют средства (команды DOS или утилиты, входящие в состав DOS) выгрузки резидентных программ. Единственный предусмотренный для этого механизм - перезагрузка



компьютера. Практически, однако, большинство резидентных программных продуктов имеют встроенные средства выгрузки. Обычно выгрузка резидентной программы осуществляется соответствующей командой, подаваемой с клавиатуры и воспринимаемой резидентной программой. Для этого резидентная программа должна перехватывать прерывания, поступающие с клавиатуры, и "вылавливать" команды выгрузки. Другой, более простой способ заключается в запуске некоторой программы, которая с помощью, например, мультиплексного прерывания 2Fh передает резидентной программе команду выгрузки. Чаше всего в качестве "выгружающей" используют саму резидентную программу, точнее, ее вторую копию, которая, если ее запустить в определенном режиме, не только не пытается остаться в памяти, но, наоборот, выгружает из памяти свою первую копию.

Выгрузку резидентной программы из памяти можно осуществить разными способами. Наиболее простой - освободить блоки памяти, занимаемые программой (собственно программой и ее окружением) с помощью функции DOS 49h. Другой, более сложный - использовать в выгружающей программе функцию завершения 4Ch, заставив ее завершить не саму выгружающую, а резидентную программу, да еще после этого вернуть управление в выгружающую. В любом случае перед освобождением памяти необходимо восстановить все векторы прерываний, перехваченные резидентной программой. Следует подчеркнуть, что восстановление векторов представляет в общем случае значительную и иногда даже неразрешимую проблему. Во-первых, старое содержимое вектора, которое хранится где-то в полях данных резидентной программы, невозможно извлечь "снаружи", из другой программы, так как нет никаких способов определить, где именно его спрятала резидентная программа в процессе инициализации. Поэтому выгрузку резидентной программы легче осуществить из нее самой, чем из другой программы. Во-вторых, даже если выгрузку осуществляет сама резидентная программа, она может правильно восстановить старое содержимое вектора лишь в том случае, если этот вектор не был позже перехвачен другой резидентной программой. Если же это произошло, в таблице векторов находится уже адрес не выгружаемой, а следующей резидентной программы, и если восстановить старое содержимое вектора, эта следующая программа "повиснет", лишившись средств своего запуска. Поэтому надежно можно выгрузить только последнюю из загруженных резидентных программ.

Для того, чтобы удалить из памяти резидентную программу, надо в какой-то программе вызвать прерывание 2Fh с функцией, присвоенной этой программе, и подфункцией 01h. Проще

всего создать для этого специальную "выгружающую" программу. Очевидно, однако, что такая методика выгрузки резидентной программы довольно неуклюжа. Для каждой резидентной программы нам придется создавать свою выгружающую программу. Гораздо изящнее использовать в качестве выгружающей саму резидентную программу. Пусть, например, наша резидентная программа имеет имя DUMP.COM. Предусмотрим в секции инициализации программы механизм анализа командной строки так, чтобы команда

DUMP

загружала эту программу в память, оставляя ее резидентной, а команда

DUMP OFF

"деактивировала" программу и выгружала ее из памяти.

Как известно, если программа запускается с клавиатуры с указанием каких-то параметров (имен файлов, ключей, определяющих режим работы программы и проч.), то DOS, загрузив программу в память, помещает все символы, введенные после имени программы (так называемый хвост команды) в префикс программного сегмента программы, начиная с относительного адреса 80h. Хвост команды помещается в PSP во вполне определенном формате. В байт по адресу 80h DOS заносит число символов в хвосте команды (включая пробел, разделяющий на командной строке саму команду и ее хвост). Далее (начиная с байта по адресу 81h) следуют все символы, введенные с клавиатуры до нажатия клавиши <Enter>. Завершается хвост кодом возврата каретки (13).

Таким образом, если программа с именем DUMP была вызвана командой

dump off

то в PSP будет записана следующая информация:

4, 'off', 13

Программа может проанализировать наличие и содержимое параметров ее запуска в PSP и, обнаружив там слово 'off', вызвать предусмотренную заранее функцию выгрузки мультиплексного прерывания 2Fh, например, функцию 01h. Естественно, в обработчике прерывания 2Fh в транзитной программе должна быть предусмотрена процедура выгрузки программы из памяти при получении "своей" функции с подфункцией 01h.

Рассмотрим в заключение принципиально иной способ завершения работы резидентной программы - выполнение в управляющей (выгружающей) программе функции DOS 4Ch

завершения текущего процесса. Применительно к конкретному приложению - выгрузке резидентной программы, этот способ вряд ли целесообразен, однако его рассмотрение позволит нам разобраться в системной процедуре завершения задачи, что может оказаться полезным, например, при организации многозадачного режима.

Как известно, функция DOS 4Ch всегда вызывается в конце программы, чтобы завершить ее выполнение и передать управление системе (конкретно - командному процессору COMMAND.COM). Однако откуда DOS узнает, какую именно программу (какой процесс) следует завершить, и куда именно передать управление? Эта информация хранится в PSP выполняемой программы. В слове со смещением 16h от начала PSP записан идентификатор (т.е. сегментный адрес PSP) родительского процесса, а в двухсловной ячейке со смещением 0Ah - конкретный адрес возврата в него (рис. 11.3). В то же время в области текущих данных SDA в слове со смещением 10h хранится идентификатор текущего процесса. Если компьютер не выполняет никакую программу, текущим является COMMAND.COM, и его ID записан в SDA. При запуске с клавиатуры любой программы (в том числе и той, которой предстоит сделаться резидентной), DOS записывает в SDA ее ID, а в PSP запущенной программы - ID родительского процесса и адрес возврата в него, чем и обеспечивается возможность завершения запущенной программы и возврата в систему. Для всех программ, запускаемых с клавиатуры (т.е. с помощью командного процессора) родительским процессом является COMMAND.COM.



Рис. 11.3. Поля PSP и SDA, имеющие отношение к завершению задачи.

Функция завершения 4Ch, выполнив все предназначенные ей действия по закрытию файлов, восстановлению векторов 22h, 23h и 24h и освобождению памяти, занимаемой текущим процессом, переносит в SDA содержимое слова PSP со смещением

16h, назначая родительский (по отношению к данному) процесс текущим, и передаст управление по адресу возврата, записанному в PSP.

Таким образом, для того, чтобы из некоторой программы, выполняющей функцию выгружающей, завершить другую (резидентную) программу, надо выполнить следующие действия (рис. 11.4):

- 1) занести в SDA ID завершаемой программы, чтобы объявить ее текущей, и чтобы функция 4Ch завершала именно ее;
- 2) занести в слово со смещением 16h в PSP завершаемой программы ID выгружающей программы, чтобы после завершения резидентной программы текущая снова стала выгружающей;
- 3) занести в двухсловную ячейку со смещением 0Ah в PSP завершаемой программы требуемый адрес возврата в выгружающую программу;
- 4) восстановить кадр стека выгружающей программы и, при необходимости, сегментные регистры, а также и регистры общего назначения, учитывая, что функция 4Ch разрушает все регистры.

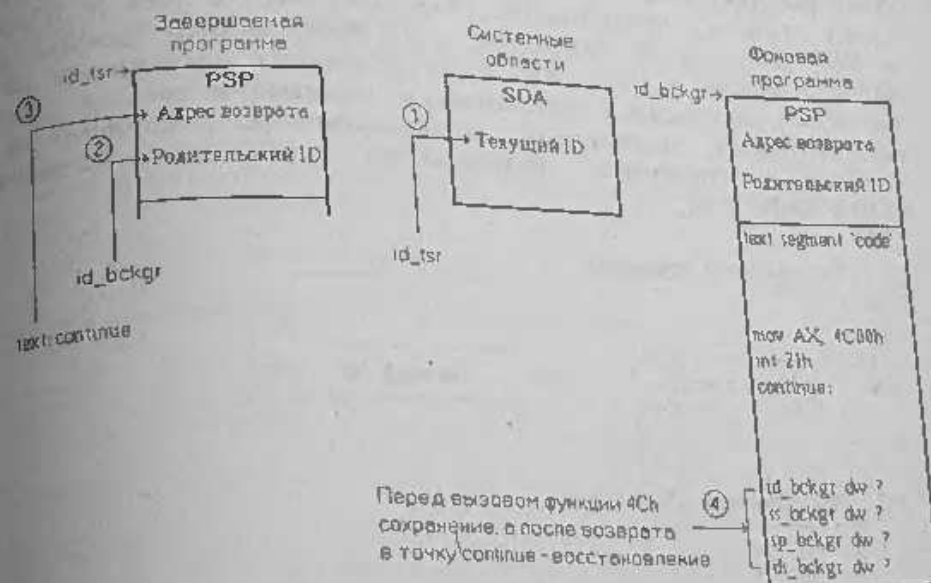


Рис. 11.4. Завершение резидентной программы из транзитной. Индексом tsr обозначена завершаемая (резидентная) программа, индексом bckgr - выгружающая (фооновая).

Рассмотрим несколько задач, в которых реализованы различные механизмы выгрузки из памяти резидентных программ.

**Задача 11.6.** Выгрузка резидентной программы с помощью специальной транзитной программы выгрузки. Подготовьте резидентную программу со средствами проверки на повторную загрузку и выгрузки с помощью мультимплексного прерывания 2Fh. Загрузите программу в память, проверьте невозможность повторной загрузки. Подготовьте транзитную программу для выгрузки резидентной. Выгрузите резидентную программу, проверьте реакцию транзитной программы на попытку выгрузки отсутствующей резидентной программы.

Основные фрагменты резидентной программы комплекса

```

text segment 'code'
    assume CS:text, DS:text
    org 256
main proc
    jmp init
old_2fh dd 0
old_09h dd 0
new_2fh proc
    cmp AH, 0C8h
    jne out_2fh
    cmp AL, 00h
    je ins
    cmp AL, 01h
    je uninstall
    jmp short out_2fh
ins: mov AL, 0FFh
    iret
out_2fh: jmp CS:old_2fh
uninstall:
;Выгрузим программу из памяти, предварительно восстановив все
;перехваченные ею векторы
    push DS
    push ES
;Восстановим вектор 09h
    mov AX, 2509h
    lds DX, CS:old_09h
    int 21h
;Восстановим вектор 2Fh
    ...
;Получим из PSP адрес собственного окружения и выгрузим его
    mov ES, CS:20Ch
    mov AH, 49h
    int 21h
;Выгрузим теперь саму программу
    push CS
    pop ES
;Загрузим в ES сегментный адрес
;PSP программы

```

```

mov AH, 49h
int 21h
pop ES
pop DS
mov AL, 99h
iret
endp
proc new_2fh
    jmp CS:old_09h
endp
proc new_09h
    jmp CS:old_09h
endp
main:
end_res=3
;Процедура инициализации
init proc
    ;Проверим, не установлена ли уже данная программа
    mov AH, 0C8h
    mov AL, 0
    int 2Fh
    cmp AL, 0FFh
    je installed
;Вернулся другой код. Приступим к установке программы (по-настоящему
;программу следует устанавливать, только если код возврата в AL равен 0)
    ;Сохраним вектор мультимплексного прерывания 2Fh в ячейке old_2fh
    ...
    ;Заполним вектор мультимплексного прерывания 2Fh адресом new_2fh
    ...
    ;Сохраним вектор прерывания от клавиатуры 09h в ячейке old_09h
    ...
    ;Заполним вектор 09h адресом new_09h (фактически в программе прерывания
    ;от клавиатуры не используются. Вектор 09h перехватывается программой
    ;для придания ей правдоподобности)
    ...
    ;Выведем на экран информационное сообщение mes об успешной установке
    ...
    ;Завершим программу, оставив ее резидентной в памяти
    installed:
    ;Выведем на экран информационное сообщение mes1 о наличии в памяти
    ;первой копии программы и отказе выполнить повторную загрузку
    ...
    ;Завершим программу функцией 4Ch. Можно установить код возврата=1
    ...
;Поля данных
mes db 'Программа динамического дампа загружена', 10, 13, '$'
mes1 db 'Программа динамического дампа уже загружена', 10, 13
db 'Повторная загрузка запрещена', 10, 13, '$'
init endp
text ends
end main

```



### Транзитная (выгружающая) программа комплекса

Программа написана в формате EXE, но, ввиду ее простоты и краткости, без сегментов данных и стека.

```

text segment 'code'
assume CS:text
main: mov     AH,0C8h      ;Наша функция мультиплексного
                             прерывания
        mov     AL,01h     ;Подфункция выгрузки
        int     2fh         ;Вызов нашей резидентной программы
        push    CS          ;Настроим DS на сегмент команд
        pop     DS          ;для упрощения адресации к данным
        cwr     AL,99h     ;Вернулся код 99h успешной
                             выгрузки?
        jne     error       ;Если нет, ошибка
;Выведем сообщение mes1 об успешной выгрузке из памяти
        jmp     fin         ;На завершение

error:
;Выведем сообщение mes2 об ошибке выгрузки
        jmp     fin

fin:    mov     AX,4C00h
        int     21h
mes1 db 'Резидентная программа успешно выгружена из памяти',10,13,'$'
mes2 db 'Произошел сбой при выгрузке резидентной программы',10,13,'$'
text ends
end main

```

**Задача 11.7.** Модифицируем резидентную программу предыдущей задачи так, чтобы ее можно было выгрузить с клавиатуры путем повторного запуска с параметром 'off'. По сравнению с задачей 11.6 изменяется только секция инициализации, которая и приведена ниже.

```

;Процедура инициализации
tail db 'off'
flag db 0
init proc
;Получим хвост команды из PSP
        mov     CL,ES:80h
        cmp     CL,0
        je      ahead
параметров
        xor     CH,CH
        mov     DI,81h
        mov     SI,offset tail
        mov     AL,' '
gere scasb
        dec     DI
gere mov     CX,3
        cmpsb
        jne     ahead
;Ожидаемый хвост команды
;Флаг требования выгрузки
;Получим длину хвоста в PSP
;Длина хвоста=0?
;Да, программа запущена без
;Теперь CX=CL=длина хвоста
;ES:DI->хвост в PSP
;DS:SI->поле tail в программе
;Уберем пробелы из начала хвоста
;Сканируем хвост, пока пробелы
;DI->на первый символ после
;пробелов
;Ожидаемая длина параметра
;Сравниваем введенный хвост
;с ожидаемым
;Введена неизвестная команда

```

```

inc     flag
;Проверим, установлена ли данная программа
ahead: mov     AH,0C8h
        mov     AL,0
        int     2fh
        cmp     AL,0FFh
        je      installed
;Введено 'off', установим флаг
;запроса на выгрузку
;Наша функция прерывания 2fh
;Подфункция проверки установки
;Мультиплексное прерывание
;Проанализируем код возврата
;Программ установлена, при
;наличии запроса на выгрузку
;ее можно выгрузить
;Действия по установке программы в память
;Сохраним вектор мультиплексного прерывания 2fh
;Заполним вектор мультиплексного прерывания 2fh
;Сохраним вектор прерывания от клавиатуры 09h
;Заполним вектор прерывания от клавиатуры 09h
;Выведем на экран информационное сообщение mes об успешной установке
;Завершим программу, оставив ее резидентной в памяти
installed:
        cmp     flag,1
        je      uninstall
;Запрос на выгрузку установлен?
;Да, на выгрузку
;Если запроса на выгрузку нет, произошел повторный (ошибочный)
;вызов программы. Выведем на экран информационное сообщение mes1
;о невозможности повторной загрузки
;Завершим программу обычным образом, функцией 4Ch
uninstall:
;Перешлем в первую (резидентную) копию программы запрос на выгрузку
        mov     AX,0C801h
        int     2fh
;Функция C8h, подфункция 01h
;Мультиплексное прерывание
;Выведем сообщение mes2 о выгрузке программы
;Первая копия программы выгружена, завершим и эту
        mov     AX,4C00h
        int     21h
mes db 'Программа динамического дампа загружена',10,13,'$'
mes1 db 'Программа динамического дампа уже загружена',10,13,'$'
mes2 db 'Повторная загрузка запрещена',10,13,'$'
mes2 db 'Программа выгружена из памяти',10,13,'$'

```

**Задача 11.8.** Завершение резидентной программы из другой (транзитной) программы. Резидентная программа активизируется из транзитной синхронно, через вектор 60h. Она открывает файл с известным ей именем и дописывает в его конец некоторые данные.

Транзитная программа сначала активизирует резидентную программу, после чего завершает ее работу и выгружает из памяти функцией 4Ch.

Резидентная программа комплекса

```

text segment 'code'
assume cs:text,ds:text
org 100h
myproc proc init
; Резидентные данные
id_bckgr dw ?
handle dw ?
fname db 'file.dat', 0
data db 'Данные, добавляемые в файл резидентной программой', 10, 13
date_len = $ - data
; Резидентная программная секция
entry:
; Сохраним используемые регистры
push AX
push BX
push CX
push DX
push DS
; Установим адресацию наших данных через DS
push CS
pop DS
; Получим и сохраним ID текущего процесса, т.е. ID вызвавшей
; транзитной программы
mov AH, 51h
int 21h
mov id_bckgr, BX
; Установим в SDA свой ID. Программа типа .COM, поэтому ID в CS
mov AH, 50h
mov BX, CS
int 21h
; Откроем файл уже на своем PSP
; Поскольку вызов резидентной программы синхронный,
; можно не проверять флаг занятости DOS
mov AX, 3D02h
mov DX, offset fname
int 21h
mov handle, AX
; Установим указатель на конец файла для дописывания к нему
; "новых данных"
mov AX, 4202h
mov BX, handle
xor CX, CX
xor DX, DX
int 21h
; Допишем в файл "новые данные"
mov AH, 40h
mov BX, handle

```

```

mov CX, data_len
mov DX, offset data
int 21h
; Файл не закрываем умышленно
; Восстановим ID прерванной программы
mov AH, 50h
mov BX, id_bckgr
int 21h
; Восстановим все регистры
pop DS
pop DX
pop CX
pop BX
pop AX
; Вернемся в прерванную программу
iret
endp
myproc
; Секция инициализации резидентной программы
init proc
; Загрузим вектор пользователя 60h для связи с транзитной программой
mov AX, 2560h
mov DX, offset entry
int 21h
; Сохраним ID резидентной программы в области межзадачных связей
; для передачи его транзитной программе
mov AX, 40h
mov ES, AX
mov ES:0F0h, CS
; Выведем сообщение о загрузке резидентной программы
mov AH, 09h
mov DX, offset mes
int 21h
; Завершим программу, оставив ее в памяти
mov AX, 3100h
mov DX, (init-myproc+10Fh)/16
int 21h
; Транзитные поля данных
mes db 'Резидентная программа загружена', 10, 13, '$'
init endp
text end
myproc

```

Основные фрагменты транзитной (выгружающей) программы комплекса

```

; Активируем резидентную программу через вектор 60h
int 60h
; Начнем действия по завершению резидентной программы
mov AX, data
mov DS, AX
; Инициализируем сегментный регистр DS
; Работая по всем правилам, получим свой ID,
; который, впрочем, находится в ES
mov AH, 51h

```

```

int 21h
;Сохраняем свой ID в ячейке памяти
mov CS:ld_bckgr,BX
;Настроимся на область межзадачных связей
;и получим ID резидентной программы
mov AX,40h
mov ES,AX
mov SI,ES:0F0h
;Выполним шаг 1 - установим в SDA ID резидентной программы
mov AH,50h
mov BX,SI
int 21h
;Выполним шаг 2 - установим в PSP завершаемой программы
;ID родительского процесса, т.е. адрес PSP настоящей программы
mov AX,CS:ld_bckgr
mov ES,SI
mov ES:16h,AX
;Выполним шаг 3 - установим в PSP завершаемой программы адрес возврата
;в родительский процесс, т.е. полный адрес точки continue
mov word ptr ES:0Ah,offset continue
mov word ptr ES:0Ch,CS
;Выполним шаг 4 - сохраним регистры
mov CS:ss_bckgr,SS
mov CS:sp_bckgr,SP
mov CS:ds_bckgr,DS
;Завершим резидентную программу
mov AX,4C00h
int 21h
continue:
;Восстановим регистры
cli
mov SS,CS:ss_bckgr
mov SP,CS:sp_bckgr
sti
mov DS,CS:ds_bckgr
;Выведем на экран строку текста mes
...
;Завершим программу
...
;Поля данных для сохранения регистров
;удобно расположить в сегменте команд
ld_bckgr dw ?
ss_bckgr dw ?
sp_bckgr dw ?
ds_bckgr dw ?
;Поля данных
mes db 'Резидентная программа завершена и выгружена',10,13,'$'

```

## 11.7. Свопинг области текущих данных DOS

В заключение рассмотрим методику, использование которой позволяет относительно легко решить обсуждавшиеся выше

проблемы с нересентерабельностью DOS. Как было показано в предыдущих разделах, нересентерабельность DOS связана с использованием ею в процессе выполнения системных функций области текущих данных SDA, где находятся стеки DOS, а также ячейки для хранения целого ряда характерных адресов и других величин, в частности, кадров стеков. Наиболее радикальным методом придания DOS свойств ресентерабельности (при вызовах DOS в обработчиках обратных прерываний) является сохранение в прикладном обработчике перед вызовом каких-либо системных функций всей области текущих данных, и восстановление исходного содержимого SDA после завершения работы с DOS. Как уже отмечалось, адрес SDA можно получить с помощью недокументированной функции 5D06h, которая также возвращает и размер SDA. Такого рода процедура часто называется свопингом, что и дало название области текущих данных (Swappable Data Area).

Перед выполнением свопинга целесообразно по-прежнему проверить состояние флага InDOS, так как если этот флаг окажется сброшен, в свопинге нет необходимости.

**Задача 11.9.** Модифицируем задачу 11.1, заменив в ней "отсроченную" (с помощью системного таймера) активизацию процедуры task с вызовами системных функций на активизацию task без всяких условий, но с предварительным свопингом SDA. Общая структура задачи остается без изменений. Добавляются процедура swapi, в которой выполняется сохранение SDA в предусмотренном в задаче буфере, и процедура swap2, восстанавливающая исходное содержимое SDA из буфера пользователя. Удаляется перехватчик прерываний от системного таймера, в котором уже нет необходимости, и несколько изменяется алгоритм обработчика прерываний от клавиатуры.

```

text segment 'code'
assume CS:text,DS:data
;Подпрограмма преобразования 2х-разрядного двоичного числа
;в десятичное число в ASCII-представлении
binasc proc
...
binasc endp
;Обработчик прерываний от клавиатуры
new_09h proc

```

```

push AX
push BX
push ES
in AL,60h
cmp AL,4Eh
je plus
pop ES
pop BX

```

```

;Сохраняем
;используемые
;регистры
;Введем скен-код
;Серый плюс?
;Да, на продолжение
;Нет, восстановим
;сохраненные

```



302

```

; регистры
; и вызовем системный обработчик
; Выполним волшебные действия
; с контроллером клавиатуры,
; чтобы снять скен-код нажатой
; клавиши и разрешить
; дальнейшую работу контроллера
; Пошлем в контроллер прерывания
; сигнал EOI
; Получим адрес флага InDOS
; DOS свободна?
; Нет, выполним свопинг
; Да, вызовем процедуру task
; На выход из обработчика
; Сохраним SDA
; Вызовем task
; Восстановим SDA
; Восстановим
; сохраненные
; регистры
; Возврат в прерванную программу

; Процедура сохранения в буфере пользователя всей SDA
swap1 proc
    push    AX
    push    CX
    push    SI
    push    DI
    push    DS
    push    ES
    mov     AX,data
    mov     ES,AX
    mov     SI,word ptr ES:old_sda
    mov     DS,word ptr ES:old_sda+2
    mov     DI,offset new_sda
    mov     CX,ES:sda_len
    cld
    movsb
    rep     movsb
    pop     ES
    pop     DS
    pop     DI
    pop     SI
    pop     CX
    pop     AX
    ret
swap1 endp

```

; Процедура восстановления первоначального содержимого SDA

```

swap2 proc
    push    AX
    push    CX
    push    SI
    push    DI

```

303

```

    push    DS
    push    ES
    mov     AX,data
    mov     DS,AX
    mov     SI,offset new_sda
    mov     ES,word ptr old_sda+2
    mov     DI,word ptr old_sda
    mov     CX,sda_len
    cld
    movsb
    rep     movsb
    pop     ES
    pop     DS
    pop     DI
    pop     SI
    pop     CX
    pop     AX
    ret
swap2 endp

```

; Обработчик прерываний по <Ctrl>/C и <Ctrl>/Break

```

new_23h proc
    ...
new_23h endp

```

; Основная программа

```

main proc
    mov     AX,data
    mov     DS,AX
    mov     ES,AX
    ; Выполним подготовительные действия
    ; Получим и сохраним адрес и длину SDA
    mov     AX,5D06h
    int     21h
    mov     word ptr ES:old_sda,SI
    mov     word ptr ES:old_sda+2,DS
    mov     ES:sda_len,CX
    push    ES
    pop     DS
    ; Получим адрес флага InDOS
    mov     AH,34h
    int     21h
    mov     word ptr CS:in_dos,BX
    mov     word ptr CS:in_dos+2,ES

```

; Сохраним вектор прерывания от клавиатуры (09h) в ячейке old\_09h

; Заполним вектор 09h адресом нашего обработчика new\_09h

; Сохраним вектор прерывания по <Ctrl>/C (23h) в ячейке old\_23h

; Заполним вектор 23h адресом нашего обработчика new\_23h

; Будем периодически выводить строки line1 и line2 на одно и то же место экрана с помощью функции DOS 40h, как это делалось в задаче 11.1.

```

;Процедура task, активируемая горячей клавишей
;и использующая функции DOS
task proc
...
task endp
home:
;Заверши программу, восстанови сначала все перехваченные векторы
;Восстанови вектор 09h
...
;Заверши программу
...
wproc endp
;Поля данных, размещенные для удобства адресации в сегменте кода
;Удалены поля task_reg и old_08h
old_09h dd 0
old_23h dd 0
in_dos dd 0
text ends
;Поля данных, размещенные в сегменте данных
old_sda dd 0 ;Адрес SDA
sda_len dw 78Ch ;Размер SDA
new_sda db 78Ch dup (0) ;Буфер для хранения SDA (известно,
;что размер SDA равен 78Ch байтов)
string db 27,'[s',27,'[25;71H',27,'[34;41m'
time db 27,'[0m'
db 27,'[u'
db '$'
ten db 10
line1 db 40 dup ('-'),13
line1_len=$-line1
line2 db 40 dup ('|'),13
line2_len=$-line2

```

## Приложение 1. Справочные данные по функциям DOS

**INT 21h, функция 01h. Ввод символа с эхом.**

Вводит символ из устройства стандартного ввода и отображает его на устройстве стандартного вывода. При отсутствии символа ждет ввода. Допустимо перенаправление ввода. Если ввод перенаправлен, выполняет обработку <Ctrl>/C. Если ввод включенном режиме BREAK (BREAK-ON). Для чтения расширенного кода ASCII требуется повторное выполнение функции.

При вызове: AH=01h

При возврате: AL=байт входных данных

**INT 21h, функция 02h. Вывод символа.**

Выводит символ на экран. Допустимо перенаправление вывода. Выполняет обработку <Ctrl>/C при вводе этой комбинации с клавиатуры перед выводом каждого 64-го символа. Коды ASCII: 07h - звонок, 08h - шаг назад, 09 - табуляция, 0Dh - возврат каретки, 0Ah - перевод строки, рассматриваются как управляющие и выполняются соответствующие им действия.

При вызове: AH=02h

DL=байт данных

**INT 21h, функция 05h. Вывод символа на принтер.**

Выводит символ на стандартный принтер. Если принтер занят, функция ждет его освобождения. Выполняет обработку <Ctrl>/C при вводе этой комбинации с клавиатуры.

При вызове: AH=05h

DL=байт данных

**INT 21h, функция 06h. Прямой ввод - вывод.**

Вводит из устройства стандартного ввода или выводит на устройство стандартного вывода коды символов. В режиме вывода коды ASCII: 07h - звонок, 0Dh - возврат каретки, 0Ah - перевод строки, рассматриваются как управляющие и выполняются соответствующие им действия. Код 08h - возврат на шаг отбрасывается только если вывод не перенаправлен. Допустимо перенаправление ввода - вывода. Для чтения расширенного кода ASCII требуется повторное выполнение функции. При

отсутствии символа не ждет его ввода, а возвращает управление в программу.

При вызове: AH=06h

DL=код символа (00h - FEh) (при выводе)  
DL=FFh (при вводе)

При возврате: AL=код символа (при вводе); если символа нет, то ZF=1

INT 21h, функция 07h. Нефильтрованный ввод без эха.

Вводит символ из устройства стандартного ввода без его отображения. При отсутствии символа ждет ввода. Допустимо перенаправление ввода. Не выполняет обработку <Ctrl>/C. Для чтения расширенного кода ASCII требуется повторное выполнение функции.

При вызове: AH=07h

При возврате: AL=байт входных данных

INT 21h, функция 08h. Ввод символа без эха.

Вводит символ из устройства стандартного ввода без его отображения. При отсутствии символа ждет ввода. Допустимо перенаправление ввода. Для чтения расширенного кода ASCII требуется повторное выполнение функции. Если ввод не перенаправлен, выполняет обработку <Ctrl>/C. Если ввод перенаправлен, выполняет обработку <Ctrl>/C при включенном режиме BREAK.

При вызове: AH=08h

При возврате: AL=байт входных данных

INT 21h, функция 09h. Вывод строки.

Выводит строку символов на устройство стандартного вывода. Строка должна заканчиваться символом \$. Допустимо перенаправление вывода. Допустимо использование Esc-последовательностей. Коды ASCII: 07h - звонок, 08h - шаг назад, 0Dh - возврат каретки, 0Ah - перевод строки, рассматриваются как управляющие и выполняются соответствующие им действия. Выполняет обработку <Ctrl>/C при вводе этой комбинации с клавиатуры перед выводом каждого 64-го символа.

При вызове: AH=09h

DS:DX=адрес строки

INT 21h, функция 0Ah. Буферизованный ввод с клавиатуры.

Вводит строку байт из устройства стандартного ввода в буфер пользователя с отображением на устройстве стандартного вывода. Строка должна заканчиваться символом возврата каретки (0Dh). Допустимо перенаправление ввода. Если ввод не перенаправлен, выполняет обработку <Ctrl>/C. Если ввод перенаправлен, выполняет обработку <Ctrl>/C при включенном режиме BREAK.

При вызове: AH=0Ah

DS:DX=адрес буфера

При возврате: Данные помещены в буфер. Формат буфера:  
байт 0 - ожидаемая длина строки  
байт 1 - фактическая длина введенной строки  
байт 2 и далее - строка, заканчивающаяся 0Dh

INT 21h, функция 0Bh. Проверка состояния ввода.

Проверяет наличие символа от устройства стандартного ввода. Допустимо перенаправление ввода. Если ввод не перенаправлен, выполняет обработку <Ctrl>/C. Если ввод перенаправлен, выполняет обработку <Ctrl>/C при включенном режиме BREAK.

При вызове: AH=0Bh

При возврате: AL=00h если символ не ждет  
AL=FFh если символ ждет

INT 21h, функция 0Ch. Очистка входного буфера и ввод.

Очищает кольцевой буфер клавиатуры и активизирует функцию ввода. Допустимо перенаправление ввода.

При вызове: AH=0Ch

AL=номер требуемой функции ввода.

Допустимы функции 01, 07, 08, 0Ah

DS:DX=адрес буфера (если AL=0Ah)

При возврате: AL=байт входных данных (если при вызове AL=0Ah, данные помещаются в буфер)

INT 21h, функция 0Eh. Выбор диска.

Назначает текущий диск и возвращает число логических дисководов в системе.

При вызове: AH=0Eh

AL=код дисковода (0=A, 1=B и т.д.)

При возврате: AL=число логических дисководов в системе

INT 21h, функция 19h. Получение текущего диска.

Возвращает код текущего диска.

При вызове: AH=19h

При возврате: AL=код текущего диска (0=A, 1=B и т.д.)

INT 21h, функция 1Ah. Установка адреса области обмена с диском.

Позволяет определить адрес дисковой области передачи (DTA) для последующих операций с блоками управления файлами.

При вызове: AH=1Ah

DS:DX=адрес DTA



**INT 21h, функция 1Bh. Получение информации о текущем диске.**

Возвращает характеристики текущего диска.

При вызове: AH=1Bh

При возврате: AL=количество секторов в кластере  
CX=количество байтов в секторе  
DX=общее количество кластеров на диске  
DS:BX->байт описания носителя:

FFh - дискета 320 Кбайт

FEh - дискета 160 Кбайт

FDh - дискета 360 Кбайт

FCh - дискета 180 Кбайт

F9h - дискета 1,2 Мбайт

F8h - жесткий диск

F0h - другие

**INT 21h, функция 1Ch. Получение информации о заданном диске.**

Возвращает характеристики заданного диска.

При вызове: AH=1Ch

При возврате: AL=количество секторов в кластере  
CX=количество байтов в секторе  
DX=общее количество кластеров на диске  
DS:BX->байт описания носителя (см. функцию 1Bh)

**INT 21h, функция 1Fh. Получение адреса блока параметров текущего диска.**

Позволяет получить детальную информацию о параметрах текущего диска.

При вызове: AH=1Fh

При возврате: AL=00h (успешное выполнение)

DS:BX->блок параметров диска (см. табл. 4.5)

При ошибке: AL=FFh (недопустимый дисковод)

**INT 21h, функция 25h. Установка вектора прерывания.**

Позволяет заполнить вектор прерывания адресом программы обработки прерываний.

При вызове: AH=25h

AL=номер вектора прерывания

DS:DX=адрес программы обработки прерываний

**INT 21h, функция 2Ah. Получение системной даты.**

Позволяет получить значение текущей даты.

При вызове: AH=2Ah

При возврате: CX=год (от 1980 до 2099)  
DH=месяц (от 1 до 12)  
DL=день (от 1 до 31)  
AL=день недели (0-воскресенье и т.д.)

**INT 21h, функция 2Bh. Установка системной даты.**

Позволяет изменить дату внутреннего календаря.

При вызове: AH=2Bh

CX=год (от 1980 до 2099)

DH=месяц (от 1 до 12)

DL=день (от 1 до 31)

При возврате: AL=0 (успешное выполнение)

При ошибке: AL=FFh (недопустимая дата, системная дата не изменилась)

**INT 21h, функция 2Ch. Получение времени.**

Позволяет получить значение текущего времени.

При вызове: AH=2Ch

При возврате: CH=часы (от 0 до 23)

CL=минуты (от 0 до 59)

DH=секунды (от 0 до 59)

**INT 21h, функция 2Dh. Установка системного времени.**

Позволяет изменить время внутренних часов.

При вызове: AH=2Dh

CH=часы (от 0 до 23)

CL=минуты (от 0 до 59)

DH=секунды (от 0 до 59)

При возврате: AL=00h (успешное выполнение)

При ошибке: AL=FFh (недопустимое время, системное время не изменилось)

**INT 21h, функция 2Eh. Установка флага проверки.**

Изменяет состояние флага проверки записи на диск.

При вызове: AH=2Eh

AL=00h установить флаг проверки

AL=01h сбросить флаг проверки

**INT 21h, функция 2Fh. Получение адреса области обмена с диском.**

Возвращает адрес текущей области обмена с диском (DTA).

При вызове: AH=2Fh

При возврате: ES:DX=адрес DTA

**INT 21h, функция 30h. Получение версии DOS.**

Возвращает номер используемой версии MS-DOS.

При вызове: AH=30h

AL=00h в BH вернуть номер OEM (V5.0+)

AL=01h в BH вернуть флаг версии (V5.0+)

При возврате: AL=номер основной версии  
AH=номер подверсии

BH=номер OEM:

00h IBM

16h DEC

99h архитектура STARLITE

FFh Phoenix

BH=флаг версии:

08h DOS находится в ПЗУ

10h DOS находится в области старшей памяти (HMA)

INT 21h, функция 31h. Завершение программы и сохранение ее резидентной в памяти

Завершает выполнение активной программы, резервируя при этом для завершаемой программы указанный объем памяти и возвращая управление родительскому процессу. Возвращает в DOS код возврата. В процессе завершения сбрасывает на диск буферы, закрывает дескрипторы, восстанавливает из ячеек PSP векторы 22h, 23h, 24h.

При вызове: AH=31h

AL=код возврата

DX=объем резервируемой памяти в параграфах

INT 21h, функция 32h. Получение адреса блока параметров заданного диска.

Позволяет получить детальную информацию о параметрах заданного диска.

При вызове: AH=32h

DL=номер диска (00h=по умолчанию, 01h=A; и т.д.)

При возврате: AL=00h (успешное выполнение)

DS:BX->блок параметров диска (см. табл. 4.5)

При ошибке: AL=FFh (недопустимый дисковод)

INT 21h, функция 33h, подфункции 00h и 01h. Получение или установка состояния Break.

Позволяет определить или задать условия реакции DOS на ввод с клавиатуры комбинации <Ctrl>/C или <Ctrl>/<Break>. Функция не использует внутренние стеки DOS и поэтому реентерабельна.

При вызове: AH=33h

AL=00h получить состояние Break

AL=01h установить состояние Break:

DL=00h состояние Break выключено, проверка только для функций DOS 01...0Ch

DL=01h состояние Break включено, проверка для всех функций DOS

При возврате: DL=текущее состояние Break (если при вызове AL=00h):  
00h состояние Break выключено (OFF)  
01h состояние Break включено (ON)

INT 21h, функция 33h, подфункция 02h. Получение и установка состояния Break.

Позволяет определить и задать условия реакции DOS на ввод с клавиатуры комбинации <Ctrl>/C или <Ctrl>/<Break> в одной операции. Функция не использует внутренние стеки DOS и поэтому реентерабельна.

При вызове: AX=3302h

DL=00h состояние Break выключено, проверка только для функций DOS 01...0Ch

DL=01h состояние Break включено, проверка для всех функций DOS

При возврате: DL=прошрое состояние Break:  
00h состояние Break выключено (OFF)  
01h состояние Break включено (ON)

INT 21h, функция 33h, подфункция 05h. Получение дисковода загрузки.

Определяет дисковод, с которого была загружена система.

При вызове: AX=3305h

При возврате: DL=дисковод загрузки (l=A; и т.д.)

INT 21h, функция 34h. Получение адреса флага занятости DOS (флага InDOS).

Возвращает адрес байта области текущих данных DOS (SDA), содержащего флаг InDOS.

При вызове: AH=34h

При возврате: ES:BX->однобайтовый флаг InDOS

INT 21h, функция 35h. Получение вектора прерывания.

Возвращает содержимое указанного вектора прерывания.

При вызове: AH=35h

AL=номер вектора прерывания

При возврате: ES:BX=адрес программы обработки прерываний

**INT 21h, функция 36h. Получение объема свободного пространства на диске.**  
Возвращает информацию о дисковом, из которой можно вычислить емкость носителя и объем незанятого пространства. Потерянные кластеры считаются занятыми.

При вызове: AH=36h  
DL=код дисковода (00h-текущий, 01h=A и т.д.)

При возврате: AX=число секторов в кластере  
BX=число свободных кластеров  
CX=размер сектора в байтах  
DX=полное число кластеров на диске

При ошибке: AX=FFFFh

**INT 21h, функция 39h. Создание каталога.**  
Создает каталог в конце указанного пути.

При вызове: AH=39h  
DS:DX=адрес пути в виде строки ASCIIZ

При ошибке: CF=1  
AX=код ошибки

**INT 21h, функция 3Ah. Удаление каталога.**  
Удаляет указанный каталог.

При вызове: AH=3Ah  
DS:DX=адрес каталога в виде строки ASCIIZ

При ошибке: CF=1  
AX=код ошибки

**INT 21h, функция 3Bh. Смена текущего каталога.**  
Устанавливает новый текущий каталог.

При вызове: AH=3Bh  
DS:DX=адрес каталога в виде строки ASCIIZ

При ошибке: CF=1  
AX=код ошибки

**INT 21h, функция 3Ch. Создание или усеменение файла.**  
Создает новый файл с указанной спецификацией. Если указанный файл существует, он усекается до нулевой длины. В любом случае файл открывается и возвращается его дескриптор для дальнейших операций над файлом.

При вызове: AH=3Ch  
CX=атрибуты файла (могут комбинироваться):  
1 - только для чтения  
2 - скрытый  
4 - системный  
8 - метка тома  
20h - атрибут архива

DS:DX=адрес спецификации файла в виде строки ASCIIZ  
AX=дескриптор  
CF=1  
AX=код ошибки

**INT 21h, функция 3Dh. Открытие файла.**  
Открывает файл с указанной спецификацией. Возвращает дескриптор для последующих операций над файлом. Устанавливает указатель на начало файла (байт 0).

При вызове: AH=3Dh  
AL=режим доступа:  
0 - чтение  
1 - запись  
2 - запись и чтение

Примечание. Если к режиму добавлено 80h, дескриптор наследуется дочерним процессом. В противном случае дескриптор не наследуется дочерним процессом.

DS:DX=адрес спецификации файла в виде строки ASCIIZ  
AX=дескриптор  
CF=1  
AX=код ошибки

**INT 21h, функция 3Eh. Закрытие файла.**  
Сбрасывает на диск внутренние буферы файла, закрывает файл и освобождает дескриптор. Если файл был модифицирован, в записи каталога устанавливаются новые значения длины файла, а также даты и времени создания файла.

При вызове: AH=3Eh  
BX=дескриптор  
CF=1  
AX=код ошибки

**INT 21h, функция 3Fh. Чтение из файла или устройства.**  
Пересылает из файла данные в буфер пользователя и модифицирует указатель. При чтении из символического устройства в режиме ASCII читается строка указанной длины, либо до символа возврата каретки, если он встретился раньше.

При вызове: AH=3Fh  
BX=дескриптор  
CX=число пересылаемых байтов  
DS:DX=адрес буфера пользователя

При возврате: AX=число переданных байтов  
CF=1  
AX=код ошибки



INT 21h, функция 40h. Запись в файл или в устройство.  
Пересылает в файл данные из буфера пользователя и модифицирует указатель. Если при вызове CX=0, длина файла устанавливается в соответствии с текущим положением указателя. Если перед выводом с клавиатуры вводится <Ctrl>/C и режим BREAK включен (BREAK=ON), выполняется обработка <Ctrl>/C.

При вызове: AH=40h  
BX=дескриптор  
CX=число пересылаемых байтов  
DS:DX=адрес буфера пользователя  
При возврате: AX=число переданных байтов  
При ошибке: CF=1  
AX=код ошибки

INT 21h, функция 41h. Удаление файла.

Удаляет указанный файл.

При вызове: AH=41h  
DS:DX=спецификация файла в виде строки ASCII  
При ошибке: CF=1  
AX=код ошибки

INT 21h, функция 42h. Установка указателя в файле.

Позволяет установить текущее положение указателя на любой байт файла для выполнения последующих операций прямого доступа к файлу (чтения или записи).

При вызове: AH=42h  
AL=режим установки указателя:  
00h - абсолютное смещение от начала файла  
01h - знаковое смещение от текущего положения указателя  
02h - знаковое смещение от конца файла  
BX=дескриптор  
CX=старшая часть смещения  
DX=младшая часть смещения  
При возврате: DX=старшая часть возвращенного указателя  
AX=младшая часть возвращенного указателя

INT 21h, функция 43h. Получение или установка атрибутов файла.

Позволяет получить или изменить значения атрибутов файла или каталога. Файл нельзя преобразовать в каталог или метку тома. Каталог можно сделать скрытым.

При вызове: AH=43h  
AL=00h для получения атрибутов

AL=01h для установки атрибутов  
CX=атрибуты файла (могут комбинироваться):  
0001h - только для чтения  
0002h - скрытый  
0004h - системный  
0020h - атрибут архивации  
DS:DX=адрес спецификации файла или каталога

При возврате: CX=возвращаемые атрибуты файла (если при вызове AL=0)  
При ошибке: CF=1  
AX=код ошибки

INT 21h, функция 45h. Дублирование дескриптора файла.  
Создает новый дескриптор файла, который связан с заданным файлом или устройством через тот же элемент системной таблицы файлов (SFT).

При вызове: AH=45  
BX=дескриптор файла  
При возврате: AX=новый дескриптор  
При ошибке: CF=1  
AX=код ошибки

INT 21h, функция 46h. Принудительное дублирование дескриптора файла.

Принудительно объявляет указанный дескриптор дубликатом заданного. Если дескриптор в BX был открыт, он закрывается.

При вызове: AH=46h  
BX=дескриптор файла  
CX=дескриптор, который должен стать дубликатом первого  
При ошибке: CF=1  
AX=код ошибки

INT 21h, функция 47h. Получение текущего каталога.

Возвращает строку ASCII с полным путем (от корневого каталога) к текущему каталогу, включая его имя. Не возвращается обозначение текущего дискового и корневого каталога (знак \).

При вызове: AH=47h  
DL=код дискового (0-текущий, 1-A: и т.д.)  
DS:SI=адрес буфера размером 64 байтов  
При возврате: буфер заполнен спецификацией текущего каталога  
При ошибке: CF=1  
AX=код ошибки

**INT 21h. функция 48h. Выделение блока памяти.**  
Выделяет блок памяти и возвращает его сегментный адрес.

При вызове: AH=48h  
BX=требуемое число параграфов памяти  
При возврате: AX=сегментный адрес выделенного блока  
CF=1  
При ошибке: AX=код ошибки  
BX=размер наибольшего доступного блока памяти в параграфах

**INT 21h. функция 49h. Освобождение блока памяти.**  
Освобождает блок памяти и передает его системе для использования другими программами.

При вызове: AH=49h  
ES=сегментный адрес освобождаемого блока  
При ошибке: CF=1  
AX=код ошибки

**INT 21h. функция 4Ah. Изменение размера выделенного блока памяти.**  
Уменьшает или увеличивает размер выделенного блока памяти.

При вызове: AH=4Ah  
BX=требуемый размер блока в параграфах  
ES=сегментный адрес модифицируемого блока  
При ошибке: CF=1  
AX=код ошибки  
BX=размер наибольшего доступного блока памяти в параграфах

**INT 21h. функция 4Bh. Запуск программы (функция Ehex).**  
Позволяет родительскому процессу, в частности, активной прикладной программе, загрузить и запустить другую программу (дочерний процесс). После завершения запущенной программы управление возвращается родительскому процессу. Формат блока параметров см. в разделе 8.2.

При вызове: AH=4Bh  
AL=00h загрузить и выполнить программу  
AL=01h загрузить и не выполнять программу  
AL=03h загрузить оверлей  
ES:BX=адрес блока параметров  
DS:DX=адрес спецификации запускаемой программы в виде строки ASCIIZ

При ошибке: CF=1  
AX=код ошибки

**INT 21h. функция 4Ch. Завершение процесса с кодом возврата.**

Завершает текущий процесс (программу), помещая указанный код завершения в предназначенный для него байт области текущих данных DOS (SDA). В процессе завершения освобождает всю выделенную процессу память, сбрасывает все дескрипторы, закрывает все открытые файлы, сбрасывает на диск буферы, анализирует векторы 22h, 23h и 24h.

При вызове: AH=4Ch  
AL=код возврата

**INT 21h. функция 4Dh. Получение кода возврата и типа завершения.**

Используется родительским процессом после возврата из дочернего процесса, активизированного функцией Ehex (INT 21h, функция 4Bh), для получения из области текущих данных DOS (SDA) кодов возврата и завершения процесса. Код возврата в SDA в результате использования данной функции очищается, поэтому ее можно вызывать только однажды.

При вызове: AH=4Dh  
При возврате: AH=тип завершения

00h - нормальное завершение с помощью INT 20h или INT 21h (функции 00h или 4Ch)

01h - пользователь ввел <Ctrl>/C

02h - завершение через драйвер критической ошибки

03h - завершение с помощью INT 21h (функции 31h или 27h)

AL=код возврата, передаваемый из дочернего процесса.

**INT 21h. функция 4Eh. Нахождение первого файла, соответствующего заданной спецификации.**

Осуществляет поиск в указанном каталоге первого файла, соответствующего указанному шаблону групповой операции. Если CX=0, ищутся только нормальные файлы (без атрибутов). Если CX=8, ищется только метка тома. При указании других атрибутов или их комбинаций ищутся файлы с указанными атрибутами и нормальные файлы.

При вызове: AH=4Eh  
CX=атрибуты искомых файлов (могут комбинироваться)

- 1 - только для чтения
- 2 - скрытый
- 4 - системный
- 8 - метка тома

10h - каталог

20h - атрибут архивации

DS:DX=адрес спецификации искомого файла

При возврате: имя файла и расширение записаны в область обмена с диском в байты 1Eh...2Ah

При ошибке: CF=1

AX=код ошибки

INT 21h, функция 4Fh. Нахождение следующего файла.

Осуществляет поиск следующего файла после того, как функция 4Eh нашла первый файл, соответствующий указанному шаблону групповой операции. Используется только после успешного выполнения функции 4Eh. Если предполагается, что файлов, соответствующих шаблону, может быть больше двух, функцию следует выполнять многократно до получения CF=1 (файлов, соответствующих шаблону, больше нет).

При вызове: AH=4Fh

При возврате: см. функцию 4Eh

При ошибке: CF=1

AX=код ошибки

INT 21h, функция 50h. Установка идентификатора текущего процесса.

Позволяет записать в область текущих данных DOS адрес PSP программы, которую требуется объявить текущей. Функция не использует внутренние стеки DOS и поэтому реентерабельна. Функция документирована начиная с V5.0

При вызове: AH=50h

BX=сегментный адрес PSP процесса, объявляемого текущим

INT 21h, функция 51h. Получение идентификатора текущего процесса.

Позволяет получить адрес PSP программы, которую DOS считает текущей. Функция не использует внутренние стеки DOS и поэтому реентерабельна. Функция документирована начиная с DOS V5.0

При вызове: AH=51h

При возврате: BX=сегментный адрес PSP текущего процесса

INT 21h, функция 52h (недокументирована). Получение адреса списка списков.

Возвращает адрес "списка списков" - базовой системной таблицы, содержащей информацию о ряде других таблиц DOS.

При вызове: AH=52h

При возврате: ES:BX=адрес списка списков. Формат списка списков см. в табл. 4.2.

INT 21h, функция 54h. Получение флага проверки.

Позволяет получить состояние флага проверки.

При вызове: AH=54h

При возврате: AL=флаг проверки;

00h=выключен

01h=включен

INT 21h, функция 56h. Переименование файла.

Переименовывает файл или перемещает его в другой каталог на том же диске. Допустимо переименование каталога. Недопустимо использование шаблонов групповых операций.

При вызове: AH=56h

DS:DX=адрес текущей спецификации файла

ES:DI=адрес новой спецификации файла

При ошибке: CF=1

AX=код ошибки

INT 21h, функция 57h, подфункция 00h. Получение даты и времени создания или модификации файла.

Позволяет получить дату и время создания файла, записанные в каталоге. Файл должен быть предварительно создан или открыт.

При вызове: AX=5700h

BX=дескриптор

При возврате: CX=новое время. Биты:

0...4 2х-секундные интервалы

5h...Ah минуты

Bh...Fh часы

DX=дата. Биты:

0...4 день

5...8 месяц

9h...Fh год относительно 1980)

При ошибке: CF=1

AX=код ошибки

INT 21h, функция 57h, подфункция 01h. Установка даты и времени создания файла.

Позволяет модифицировать дату и время создания файла, записанные в каталоге. Файл должен быть предварительно создан или открыт.

При вызове: AX=5701h

BX=дескриптор

CX=новое время (см. функцию 57h, подфункцию 00h)

DX=новая дата (см. функцию 57h, подфункцию 00h)



При ошибке: CF=1  
AX=код ошибки

INT 21h, функция 59h. Получение расширенной информации об ошибке.

Позволяет получить после предыдущего неудачного вызова DOS детальную информацию об ошибке, которая помещается системой в поля области текущих данных (SDA). Функция разрушает регистры CL, DX, SI, DI, BP, DS и ES. Функцию можно вызывать, в частности, в обработчике критической ошибки. Расшифровку возвращаемых значений см. табл. 11.2...11.5

При вызове: AH=59h  
BX=0000h

При возврате: AX=расширенный код ошибки  
BH=класс ошибки  
BL=рекомендуемое действие  
CH=местоположение ошибки

INT 21h, функция 5Ah. Создание временного файла.

Создает файл с указанными атрибутами в указанном каталоге и возвращает дескриптор и имя файла. Имя файлу назначается системой. При завершении программы файл не удаляется. Функцию удобно использовать, если в программе требуется создать большое и неопределенное заранее количество файлов, конкретные имена которых не имеют особого значения.

При вызове: AH=5Ah

CX=атрибуты файла (могут комбинироваться):

1 - только для чтения

2 - скрытый

4 - системный

20h - атрибут архива

DS:DX=адрес спецификации каталога в виде строки ASCIIZ

При возврате: AX=дескриптор

DS:DX=адрес полной спецификации файла в виде строки ASCIIZ

При ошибке: CF=1

AX=код ошибки

INT 21h, функция 5Bh. Создание нового файла.

Создает новый файл с указанной спецификацией и атрибутами и возвращает дескриптор. Если указанный файл уже существует, функция завершается с ошибкой.

При вызове: AH=5Bh

CX=атрибуты файла (могут комбинироваться):

1 - только для чтения

2 - скрытый

4 - системный

8 - метка тома

20h - атрибут архива

DS:DX=адрес спецификации файла в виде строки ASCIIZ

При возврате: AX=дескриптор

При ошибке: CF=1

AX=код ошибки

INT 21h, функция 5Dh, подфункция 06h. Получение адреса области текущих данных DOS (недокументированная функция).

Возвращает адрес области текущих данных DOS (Swappable Data Area, SDA), в которой хранится ряд системных переменных и, в частности, находятся все три стека DOS.

При вызове: AX=5D06h

При возврате: DS:SI->SDA

CX=размер в байтах части SDA, которая должна сохраняться при переходе на другой процесс, если прерывается функция DOS

DX=размер в байтах части SDA, которая должна сохраняться при переходе на другой процесс во всех случаях

При ошибке: CF=1

AX=код ошибки

INT 21h, функция 5Dh, подфункция 0Ah. Установка расширенной информации об ошибке.

Позволяет восстановить в SDA расширенную информацию об ошибке. Предварительно эта информация должна быть получена из SDA с помощью функции DOS 59h и сохранена в оперативных прерываний.

При вызове: AX=5D0Ah

DS:DX->3-словный список параметров, составляющих расширенную информацию об ошибке

INT 21h, функция 62h. Получение идентификатора текущего процесса.

Позволяет получить адрес PSP программы, которую DOS считает текущей. Функция не использует внутренние стеки DOS и поэтому реентерабельна. Идентична функции 51h

При вызове: AH=62h

При возврате: BX=сегментный адрес PSP текущего процесса

**INT 21h, функция 67h. Установка числа дескрипторов.**  
Устанавливает максимальное число файлов и устройств, которые могут быть одновременно открыты текущим процессом.

Фактически функция создает новую таблицу файлов задания JFT, копируя в ее начало исходную JFT, находящуюся в PSP программы и имеющую размер 20 байтов. Новая таблица создается в свободной памяти за пределами программы и для ее успешного выполнения требуется, чтобы в системе был свободный блок памяти соответствующего размера. Поскольку максимальное число открытых файлов лимитируется не только размером JFT, но также и числом блоков описания файлов в системной таблице файлов SFT, наряду с расширением JFT требуется также расширить SFT с помощью директивы файла CONFIG.SYS FILES=.

При вызове: AH=67h

При возврате: BX=требуемое число дескрипторов  
в PSP текущей программы записан адрес новой JFT

**INT 21h, функция 68h. Сброс буферов DOS в файл.**

Выполняет принудительное обновление файла на диске. Все данные, находящиеся в буферах DOS, записываются в файл. Обновляется запись каталога.

При вызове: AH=68h

BX=дескриптор

При ошибке: CF=1

AX=код ошибки

**INT 21h, функция 69h. Получение или установка серийного номера тома.**

Читает или записывает метку тома и серийный номер тома для данного диска.

При вызове: AH=69h

AL=подфункция:

00h получить серийный номер

01h установить серийный номер

BL=дисковод (0=текущий, 1=A: и т.д.)

DS:BX->буфер размером 32 байта

При возврате: (AL=00h) в буфер помещается копия содержимого расширенного блока параметров BIOS (BPB) на диске (AL=01h) в расширенный BPB на диске копируется информация из буфера

При ошибке: CF=1

AX=код ошибки (см. Примечание)

**Примечание**

формат буфера:

Содержимое	Число байтов	Содержимое
00h	2	0
02h	4	Серийный номер диска
06h	11	Метка тома или 'VOLUME ID' (при AL=00h) тип файловой системы - строка 'FAT12'
11h	8	

**INT 25h. Абсолютное чтение с диска.**

Позволяет прочитать в память с диска один или группу секторов с заданным начальным относительным номером. Секторы нумеруются от 0 от начала логического (не физического!) диска. Таким образом, загрузочный сектор данного логического диска имеет номер 0, следующий за ним на диске логического сектор первой копии FAT - номер 1 и т.д.

После выполнения прерываний int 25h и int 26h в стеке задачи остается слово, содержащее значение регистра флагов. Если это слово не удалить, то нарушится дальнейший ход программы.

При вызове:

AL=номер дисководов (0=A, 1=B и т.д.)

CX=число читаемых секторов

DX=относительный номер первого читаемого сектора

DS:BX=адрес буфера

При ошибке:

CF=1

AX=коды ошибки (см. Примечание)

**Примечание**

Код ошибки в регистре AH:

01h - неправильная команда

02h - неправильная адресная метка

04h - запрошенный сектор не найден

08h - ошибка прямого доступа к памяти

10h - ошибка данных (неправильная контрольная сумма)

20h - ошибка контроллера

40h - ошибка позиционирования

Код ошибки в регистре AL:

00h - ошибка защиты записи

01h - неизвестное устройство

02h - дисковод не готов

03h - неизвестная команда

04h - ошибка данных (неправильная контрольная сумма)

06h - ошибка позиционирования

07h - неизвестный тип носителя

08h - сектор не найден

**INT 26h. Абсолютная запись на диск.**

Позволяет записать из памяти на диск один или группу секторов с заданным начальным относительным номером. Секторы нумеруются от 0 от начала логического (не физического!) диска. Таким образом, загрузочный сектор данного логического диска имеет номер 0, следующий за ним на диске первый сектор первой копии FAT - номер 1 и т.д.

После выполнения прерываний `int 25h` и `int 26h` в стеке за- дачи остается слово, содержащее значение регистра флагов. Если это слово не удалить, то нарушится дальнейший ход про- граммы.

При вызове: `AL`-номер дисковод (0-A, 1-B и т.д.)  
`CX`-число читаемых секторов  
`DX`-относительный номер первого читаемого сектора  
`DS:BX`-адрес буфера

При ошибке: `CF`=1  
`AX`-коды ошибки (см. Примечание к `INT 25h`)

**INT 2Fh. Мультиплексное прерывание.**

Прерывание предназначено для организации связи между процессами и, в частности, для обмена информацией с систем- ными и прикладными резидентными программами. Для пользо- вателя зарезервированы функции `C0h...FFh`.

При вызове: `AH`-функция  
`AL`-подфункция

Другие регистры используются по мере необходимости

При возврате: `AL`=0 если программа не установлена, и ее можно установить  
`AL`=1 если программа не установлена, и ее нельзя установить  
`AL`=FFh если программа установлена

При ошибке: `CF`=1  
`AX`=код ошибки

Приложение 2. Коды ошибок при выполнении функций DOS

- 01h - Неправильный номер функции
- 02h - Файл не найден
- 03h - Путь не найден
- 04h - Слишком много открытых файлов
- 05h - Доступ запрещен
- 06h - Неправильный дескриптор
- 07h - Уничтожен блок управления памятью
- 08h - Нехватка памяти
- 09h - Неправильный адрес блока памяти
- 0Ah - Неправильное окружение
- 0Bh - Неправильный формат
- 0Ch - Неправильный код доступа
- 0Dh - Неправильные данные
- 0Eh - Неизвестное устройство
- 0Fh - Неправильный дисковод
- 10h - Попытка удалить текущий каталог
- 11h - Не то же устройство
- 12h - Больше нет файлов
- 13h - Диск с защитой от записи
- 14h - Неизвестное устройство
- 15h - Дисковод не готов
- 16h - Неизвестная команда
- 17h - Ошибка контрольной суммы
- 18h - Неверная длина структуры запроса
- 19h - Ошибка поиска дорожки
- 1Ah - Неизвестный носитель
- 1Bh - Сектор не найден
- 1Ch - В принтере нет бумаги
- 1Dh - Отказ записи
- 1Eh - Отказ чтения
- 1Fh - Общая ошибка
- 20h - Нарушение разделения
- 21h - Нарушение запирающего файла
- 22h - Недопустимая смена дискеты
- 23h - Отсутствует блок управления файлом FCB
- 24h - Разделяемый буфер переполнен

коды ошибок при выполнении функций DOS



- 50h - Файл уже существует
- 52h - Каталог не может быть создан
- 53h - Отказ по прерыванию Int 21h (критическая ошибка)
- 54h - Слишком много перенаправлений
- 55h - Двойное перенаправление
- 57h - Неправильный параметр

### Приложение 3. Справочные данные по функциям BIOS

INT 10h, функция 00h. Установка видеорежима.  
Устанавливает текущий видеорежим.  
При вызове: AH=00h

AL=видеорежим:

03h текстовый, 80x25, 16 цветов  
10h графический, 640x350, 16 цветов  
(EGA)

04h графический, 320x200, 4 цвета  
(CGA)

12h графический, 640x480, 16 цветов  
(VGA)

13h графический, 320x200, 256 цветов  
(VGA)

INT 10h, функция 01h. Установка конфигурации курсора.  
Позволяет задать начальную и конечную строки развертки  
меряющего аппаратного курсора в текстовых видеорежимах.  
При вызове: AH=01h

CH биты 0...4 = начальная строка развертки  
курсора

CL биты 0...4 = конечная строка развертки  
курсора

INT 10h, функция 02h. Установка позиции курсора.  
Задаст положение курсора на экране в текстовых координатах на указанной странице (в том числе не активной). Курсор можно установить как в текстовом, так и в графическом режиме, однако в графическом режиме курсор не виден.  
При вызове: AH=02h

BH=страница

DH=строка

DL=столбец

INT 10h, функция 03h. Получение позиции и размера курсора.  
Возвращает положение курсора на экране для заданной страницы (в том числе не активной).

При вызове: AH=03h  
BH=страница  
При возврате: CH=начальная строка развертки для курсора  
CL=конечная строка развертки для курсора  
DH=строка  
DL=столбец

INT 10h, функция 05h. Установка видеостраницы.  
Устанавливает активную видеостраницу (как текстовую, так и графическую).  
При вызове: AH=05h  
AL=страница

INT 10h, функция 06h. Инициализация или прокрутка окна вверх.

Инициализирует окно с указанными координатами пробелами ASCII с заданным атрибутом или прокручивает содержимое окна вверх на заданное число строк. Действует только для активной страницы. При прокрутке появляющиеся внизу строки заполняются пробелами ASCII с заданным атрибутом. Функцию удобно использовать для быстрой очистки всего экрана или любой прямоугольной области на экране.

При вызове: AH=06h  
AL=число строк прокрутки; если AL=0, все окно очищается  
BH=атрибут символов в окне  
CH=Y-координата верхнего левого угла окна  
CL=X-координата верхнего левого угла окна  
DH=Y-координата нижнего правого угла окна  
DL=X-координата нижнего правого угла окна

INT 10h, функция 07h. Инициализация или прокрутка окна вниз.

Инициализирует окно с указанными координатами пробелами ASCII с заданным атрибутом или прокручивает содержимое окна вниз на заданное число строк. Действует только для активной страницы. При прокрутке появляющиеся вверху строки заполняются пробелами ASCII с заданным атрибутом. Функцию удобно использовать для быстрой очистки всего экрана или любой прямоугольной области на экране.

При вызове: AH=07h  
AL=число строк прокрутки; если AL=0, все окно очищается  
BH=атрибут символов в окне  
CH=Y-координата верхнего левого угла окна  
CL=X-координата верхнего левого угла окна  
DH=Y-координата нижнего правого угла окна

DL=X-координата нижнего правого угла окна  
INT 10h, функция 08h. Чтение символа и атрибута в позиции курсора.

Возвращает символ ASCII и его атрибут в позиции курсора на заданной странице (не только активной).  
При вызове: AH=08h  
BH=страница  
При возврате: AH=атрибут  
AL=символ

INT 10h, функция 09h. Запись символа и атрибута в позицию курсора.

Записывает символ и его атрибут в текущую позицию курсора как в графическом, так и в текстовом режимах. В графическом режиме символы не должны переходить на следующую строку. Все коды в AL рассматриваются, как знаки и не управляют положением курсора. После вывода символа курсор следует сместить к следующей позиции функцией 02h. Коэффициент повторения позволяет выводить строки одинаковых символов (но курсор не смещается). В текстовом режиме символ выводится с указанным атрибутом, т.е. заданного цвета на заданном фоне. В графическом режиме содержимое BL влияет только на цвет символа, но не фона под ним. Однако графическое изображение под знакоместом затирается.

При вызове: AH=09h  
AL=символ  
BH=страница  
BL=атрибут (текстовый режим) или цвет (графический режим)  
CX=коэффициент повторения

INT 10h, функция 0Ah. Запись символа в позицию курсора.

Записывает символ ASCII в текущую позицию курсора как в графическом, так и в текстовом режимах. Символ принимает атрибут, установленный ранее для этой позиции. В графическом режиме символы не должны переходить на следующую строку. Все коды в AL рассматриваются как знаки и не управляют положением курсора. После вывода символа курсор следует сместить к следующей позиции функцией 02h. Коэффициент повторения позволяет выводить строки одинаковых символов (но курсор не смещается).

При вызове: AH=0Ah  
AL=символ  
BH=страница  
CX=коэффициент повторения

INT 10h, функция 0Ch. Запись пиксела.  
Записывает в видеобuffer точку заданного цвета в заданной графической позиции.

При вызове: AH=0Ch  
AL=цвет (номер цветового регистра)  
BH=страница  
CX=графический столбец  
DX=графическая строка

INT 10h, функция 0Dh. Чтение пиксела.  
Читает из видеобuffer цвет пиксела в заданной графической позиции.

При вызове: AH=0Ch  
BH=страница  
CX=графический столбец  
DX=графическая строка  
При возврате: AL=цвет (номер цветового регистра)

INT 10h, функция 0Eh. Запись символа в режиме телетайпа.  
Записывает символ ASCII в текущую позицию курсора на активной странице и сдвигает курсор к следующей позиции. Коды ASCII: 07h - звонок, 08h - шаг назад, 0Dh - возврат каретки, 0Ah - перевод строки, рассматриваются как управляющие и выполняются соответствующие им действия. Остальные управляющие коды рассматриваются как знаки и выводятся на экран. Действует автоматический перевод курсора на следующую строку и скроллинг экрана. Атрибут символа указать нельзя; при записи действует атрибут, установленный ранее для текущей позиции.

При вызове: AH=0Eh  
AL=символ  
BL=цвет символа (в графическом режиме)

INT 10h, функция 0Fh. Получение видеорежима.

Позволяет получить текущий видеорежим видеоконтроллера.

При вызове: AH=0Fh  
При возврате: AH=число символьных столбцов на экране  
AL=видеорежим  
BH=активная видеостраница

INT 10h, функция 10h, подфункция 00h. Настройка цветового регистра.

Устанавливает соответствие номера цветового регистра цвету пиксела.

При вызове: AX=1000h  
BH=значение цвета в коде кэсКЗС  
BL=номер цветового регистра (0...15)

INT 10h, функция 10h, подфункция 01h. Установка цвета края экрана.  
Устанавливает цвет края экрана (выбегов развертки).

При вызове: AX=1001h  
BH=значение цвета в коде кэсКЗС

INT 10h, функция 10h, подфункция 02h. Настройка цветовой палитры и установка цвета края экрана

Устанавливает соответствие номеров цветовых регистров цветам пикселей, а также цвет края экрана (выбегов развертки).

При вызове: AX=1002h  
ES:DX=адрес 17-байтовой таблицы цветов. Таблица заполняется кодами кэсКЗС, загружаемыми в цветовые регистры 0...15 (первые 16 байтов) и в регистр края экрана (последний байт).

INT 10h, функция 10h, подфункция 03h. Переключение бита "мерцание/яркость".

Определяет назначение старшего бита (7) атрибута символа: мерцание символа или повышенная яркость фона.

При вызове: AX=1003h  
BL=назначение старшего бита атрибута:  
0 - повышенная яркость фона  
1 - мерцание символа

INT 10h, функция 10h, подфункция 07h. Чтение цветового регистра.

Возвращает содержимое указанного цветового регистра.

При вызове: AX=1007h  
При возврате: BH=значение цвета в коде кэсКЗС

INT 10h, функция 10h, подфункция 09h. Чтение цветовой палитры и цвета края экрана

Позволяет одной командой получить содержимое всех 17 цветовых регистров.

При вызове: AX=1009h  
ES:DX->17-байтовый буфер для содержимого цветовых регистров

INT 10h, функция 11h, подфункция 03h. Задание спецификатора блока знакогенератора.

Позволяет определить назначение бита 3 байта атрибутов символа в текстовых видеорежимах: яркость символа или номера блоков знакогенератора.

При вызове: AX=1103h  
BL=код блока знакогенератора



**Примечание**

Биты 0...1 регистра BL определяют номер блока (от 0 до 3) знакогенератора, символы из которого поступают на экран, если бит 3 байта атрибутов равен 0. Биты 2...3 регистра BL определяют номер блока (от 0 до 3) знакогенератора, символы из которого поступают на экран, если бит 3 байта атрибутов равен 1. Если значение битов 0...1 и 2...3 совпадают, используется только один блок знакогенератора (256 символов), а бит 3 байта атрибутов управляет яркостью символов.

INT 10h, функция 11h, подфункция 10h. Загрузка шрифта пользователя.

Загружает таблицу с определением шрифта пользователя в указанный блок генератора символов. Перепрограммирует контроллер на новый размер шрифта. При использовании подфункции 10h должна быть активна видеостраница 0. Подфункция 00h выполняет те же действия, но не перепрограммирует видеоконтроллер; при использовании этой подфункции активной может быть любая видеостраница.

При вызове: AX=1110h

BH=высота символа в числе графических точек

BL=блок генератора

CX=число символов, описанных в таблице

DX=код, назначаемый первому символу таблицы

ES:BP=адрес таблицы

INT 10h, функция 11h, подфункция 11h. Загрузка шрифта ПЗУ 8x14.

Загружает шрифт ПЗУ BIOS размером 8x14 графических точек, используемый по умолчанию, в указанный блок генератора символов. При использовании подфункции 11h должна быть активна видеостраница 0. Подфункция 01h выполняет те же действия, но не перепрограммирует видеоконтроллер; при использовании этой подфункции активной может быть любая видеостраница.

При вызове: AH=1111h

BL=блок генератора

INT 10h, функция 11h, подфункция 12h. Загрузка шрифта ПЗУ 8x8.

Загружает шрифт ПЗУ BIOS размером 8x8 графических точек, используемый по умолчанию, в указанный блок генератора символов. При использовании подфункции 12h должна быть активна видеостраница 0. Подфункция 02h выполняет те же действия, но не перепрограммирует

видеоконтроллер; при использовании этой подфункции активной может быть любая видеостраница. При вызове: AX=1112h  
BL=блок генератора

INT 10h, функция 11h, подфункция 21h. Установка вектора 43h на шрифт пользователя.

Загружает в вектор 43h адрес таблицы шрифтов пользователя для использования в графическом режиме. Одновременно модифицируется область видеоданных BIOS.

При вызове: AX=1121h

BL=код числа строк на экране

00h указывается пользователем (см. регистр DL)

01h=14 строк

02h=25 строк

03h=43 строки

CX=число строк пикселей (байтов) на символ

DL=число строк на экране (если BL=00h)

ES:BP=адрес таблицы шрифтов пользователя

INT 10h, функция 11h, подфункция 22h. Установка вектора 43h на шрифт ПЗУ 8x14.

Загружает в вектор 43h из ПЗУ BIOS адрес таблицы шрифтов с размером матрицы 8x14 точек для использования в графическом режиме. Одновременно модифицируется область видеоданных BIOS.

При вызове: AX=1122h

INT 10h, функция 11h, подфункция 23h. Установка вектора 43h на шрифт ПЗУ 8x8.

Загружает в вектор 43h из ПЗУ BIOS адрес таблицы шрифтов с размером матрицы 8x8 точек для использования в графическом режиме. Одновременно модифицируется область видеоданных BIOS.

При вызове: AX=1123h

INT 10h, функция 11h, подфункция 30h. Получение информации о шрифтах.

Позволяет получить адреса таблиц шрифтов, а также число строк пикселей (байтов) на символ для данного шрифта.

При вызове: AX=1130h

BH=код шрифта

00h=текущее содержимое вектора 1Fh

01h=текущее содержимое вектора 43h

02h=шрифт ПЗУ 8x14

03h=шрифт ПЗУ 8x8 (1-я половина)

04h=шрифт ПЗУ 8x8 (2-я половина)  
 При возврате: CX=число строк пикселей (байтов) на символ  
 DL=номер последней строки на экране  
 ES:BP=адрес таблицы со шрифтом

INT 10h, функция 13h. Запись строки в режиме телетайпа.  
 Записывает строку в текущую страницу видеобуфера начи-  
 ная с указанной позиции. Коды ASCII: 07h - звонок, 08h -  
 шаг назад, 09h - табуляция, 0Ah - перевод строки, 0Dh - воз-  
 врат каретки, рассматриваются как управляющие и выполняются  
 соответствующие им действия.

При вызове: AH=13h

AL=режим записи:

- 0 - атрибут в BL строка содержит  
только коды символов, курсор  
не смещается после записи
- 1 - атрибут в BL строка содержит  
только коды символов, курсор  
смещается после записи
- 2 - строка содержит попеременно коды  
символов и атрибутов; курсор не  
смещается после записи
- 3 - строка содержит попеременно коды  
символов и атрибутов; курсор  
смещается после записи

BH=страница

BL=атрибут (если AL=0 или 1)

CX=длина символьной строки (в длину  
входят только коды символов, но не  
байты атрибутов)

DH=номер строки на экране

DL=номер столбца на экране

ES:BP=адрес строки

INT 13h, функция 00h. Сброс дисковой системы.

Приводит дисковый контроллер в исходное состояние, пози-  
 ционирует головки на цилиндр 0 и подготавливает систему в  
 вводу-выводу.

При вызове: AH=00h

DL=дисковод

00h...7Fh - гибкий диск

80h...ffh - жесткий диск

При ошибке: CF=1

AH=состояние (см. Примечание)

### Примечание

Коды состояния могут принимать следующие значения:

- 00h - отсутствие ошибки
- 01h - неправильная команда
- 02h - не найдена адресная метка
- 03h - дискета защищена от записи
- 04h - сектор не найден
- 05h - сброс жесткого диска не прошел
- 06h - дискета вынута
- 07h - неправильная таблица параметров жесткого диска
- 0Ch - не найден тип носителя данных
- 0Dh - неправильное число секторов в формате на жестком

диске

- 10h - невозможная ошибка данных
- 11h - восстановленная ошибка данных
- 20h - неисправность контроллера
- 40h - ошибка позиционирования
- 80h - тайм-аут диска
- AAh - жесткий диск не готов
- BBh - неизвестная ошибка жесткого диска

INT 13h, функция 02h. Чтение секторов.

Читает один или группу секторов с физического (не логи-  
 ческого!) диска в память. Для начального сектора указываются  
 абсолютные координаты (цилиндр, сектор, головка). Секторы  
 физического диска нумеруются на каждой дорожке от 1, ци-  
 линдры нумеруются от 0, головки нумеруются от 0. Сначала  
 секторы 1...n цилиндра 0, головки (поверхности) 0, затем  
 секторы 1...n цилиндра 0, головки (поверхности) 1, далее сек-  
 торы 1 цилиндра 1, головки 0 и т.д. Таким образом, сектор  
 1 цилиндра 0 головки 0 относится к главной загрузочной запи-  
 си (Master boot).

При вызове:

AH=02h

AL=число читаемых секторов

CH=цилиндр

CL=начальный сектор (биты 0...5) и два  
старших бита номера цилиндра  
(биты 6-7)

DH=головка

DL=дисковод

00h...7Fh - гибкий диск

80h...ffh - жесткий диск

ES:BX=адрес буфера

При возврате:

CF=0

AH=0

AL=число переданных секторов

При ошибке: CF=1  
AH=состояние (см. функцию 00h)

INT 13h, функция 03h. Запись секторов.

Записывает один или группу секторов из памяти на физический (не логический!) диск. Для начального сектора указывается абсолютные координаты (цилиндр, сектор, головка). Секторы физического диска нумеруются на каждой дорожке от 1, цилиндры нумеруются от 0, головки нумеруются от 0. Сначала идут секторы 1...n цилиндра 0, головки (поверхности) 0, затем секторы 1...n цилиндра 1, головки 0 и т.д. Таким образом, сектор 1 цилиндра 0 головки 0 относится к главной загрузочной записи (Master boot).

При вызове: AH=03h  
AL=число читаемых секторов  
CH=цилиндр  
CL=начальный сектор (биты 0...5) и два старших бита номера цилиндра (биты 6-7)

DH=головка  
DL=дисковод  
00h...7Fh - гибкий диск  
80h...FFh - жесткий диск

ES:BX=адрес буфера

При возврате: CF=0  
AH=0  
AL=число переданных секторов

При ошибке: CF=1  
AH=состояние (см. функцию 00h)

INT 13h, функция 05h. Форматирование дорожки на гибком диске.

Форматирует указанную дорожку, нанося форматные метки.

При вызове: AH=05h  
AL=число формируемых секторов  
CH=цилиндр  
DH=головка  
DL=дисковод (00h...7Fh)  
ES:BX=адрес списка адресных полей

При ошибке: CF=1  
AH=состояние (см. функцию 00h)

#### Примечание

При форматировании дорожки дискеты следует составить список адресных полей. Каждое поле состоит из 4 байтов, в которых указываются: цилиндр, головка, сектор, код размера

Число адресных полей равно числу секторов на дорожке. Код размера сектора может принимать значения:  
0 - 128 байтов на сектор  
1 - 256 байтов на сектор  
2 - 512 байтов на сектор  
3 - 1024 байтов на сектор.  
На машинах PC/AT этой функции должна предшествовать функция 17h прерывания 13h.

INT 13h, функция 17h. Установка типа дискеты (PC/AT).  
Позволяет указать тип дискеты, установленной на конкретном дисковом.

При вызове: AH=17h  
AL=тип дискеты  
0 не используется  
1 дискета 360 Кбайт на дисковом  
360 Кбайт  
2 дискета 360 Кбайт на дисковом  
1,2 Мбайт  
3 дискета 1,2 Мбайт на дисковом  
1,2 Мбайт  
4 дискета 720 Кбайт на дисковом  
720 Кбайт  
DL=дисковод (00h...7Fh)

При ошибке: CF=1  
AH=состояние (см. функцию 00h)

INT 16h, функция 00h. Чтение символа с клавиатуры.  
Читает из кольцевого буфера ввода символ и скен-код. Если буфер пуст, ожидает ввода.

При вызове: AH=00h  
При возврате: AH=скен-код  
AL=символ ASCII

INT 16h, функция 01h. Получение состояния клавиатуры.  
Определяет, имеются ли в кольцевом буфере ожидающие ввода символы; возвращает флаг ожидания и сам символ при его наличии. Однако символ и скен-код не извлекаются из кольцевого буфера и будут снова получены при вызове функции 00h (Int16h).

При вызове: AH=01h  
При возврате: Если символ ожидает:  
ZF=0  
AH=скен-код  
AL=символ  
Если ожидающих символов нет:  
ZF=1



INT 16h. Функция 02h. Получение флагов клавиатуры.  
Возвращает байт флагов клавиатуры, записывающий состояние управляющих клавиш клавиатуры (байт в области данных BIOS по адресу 0000h:0417h).

При вызове: AH=02h

При возврате: AL=флаги. Биты байта имеют следующие значения:

- 0 - Нажата правая клавиша Shift
- 1 - Нажата левая клавиша Shift
- 2 - Нажата клавиша Ctrl
- 3 - Нажата клавиша Alt
- 4 - Включен режим Scroll Lock
- 5 - Включен режим Num Lock
- 6 - Включен режим Caps Lock
- 7 - Включен режим Insert

## Приложение 4. Команды процессора

**AAA ASCII-коррекция регистра AX после сложения**  
Команда AAA используется вслед за операцией сложения в регистре AL двух распакованных двоично-десятичных чисел. Она преобразует результат сложения в неупакованное двоично-десятичное число, младший десятичный разряд которого находится в AL. Если результат превышает 9, выполняется инкремент содержимого регистра AH.

Пример

```
mov    AX,0605h
add    AL,09h
aaa
```

;Двоично-десятичное 85  
;Двоично-десятичное 9, AX=060Eh  
;AX=0704h, двоично-десятичное 74

**AAD ASCII-коррекция регистра AX перед делением**  
Команда AAD используется перед операцией деления неупакованного двоично-десятичного числа в регистре AX на другое двоично-десятичное число. Команда преобразует делимое в регистре AX в беззнаковое двоичное число, чтобы в результате деления получились правильные неупакованные двоично-десятичные числа (частное в AL, остаток в AH).

Пример

```
mov    AX,0207h
mov    DL,06h
aad
div    DL
```

;Двоично-десятичное 27  
;Двоично-десятичное 6  
;AX=001B=27  
;AX=0304, т.е. 4 и 3 в остатке

**AAM ASCII-коррекция регистра AX после умножения**  
Команда AAM используется вслед за операцией умножения двух неупакованных двоично-десятичных чисел. Она преобразует результат умножения, являющийся двоичным числом, в правильное неупакованное двоично-десятичное число, младший разряд которого помещается в AL, а старший - в AH.

Пример

```
mov    AL,08h
mov    CL,07h
mul    CL
aam
```

;Двоично-десятичное 8  
;Двоично-десятичное 7  
;AX=0038h=56  
;AX=0506h, двоично-десятичное 56

**AAS ASCII-коррекция регистра AL после вычитания**

Команда AAS используется вслед за операцией вычитания одного неупакованного двоично-десятичного числа из другого в AL. Она преобразует результат вычитания в неупакованное двоично-десятичное число. Если результат вычитания оказывается меньше 0, выполняется декремент содержимого регистра AH.

**Пример**

mov	AX, 0708h	; Двоично-десятичное 78
mov	CL, 09h	; Двоично-десятичное 9
sub	AL, CL	; AX=07FFh, CF=1
aas		; AX=0609h, двоично-десятичное 69

**ADC Целочисленное сложение с переносом**

Команда ADC осуществляет сложение первого и второго операндов, прибавляя к результату значение флага переноса CF. Исходное значение первого операнда (приемника) теряется, замещаясь результатом сложения. Второй операнд не изменяется. В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака. Команда ADC обычно используется для сложения 32-разрядных чисел.

**Пример 1**

mov	AX, 1125h	
adc	AX, 2C25h	; AX=3D48h, если CF был = 1

**Пример 2**

В полях данных:

numlow	dw	0FFFFh	; Младшая часть 2-го слагаемого
numhigh	dw	0005h	; Старшая часть 2-го слагаемого
			; Число 5FFFFh=393215

В программном сегменте:

mov	AX, 0005h	; Младшая часть 1-го слагаемого
mov	BX, 0002h	; Старшая часть 1-го слагаемого
		; Число 20005h=131077
add	AX, numlow	; Сложение младших частей. AX=4,
		; CF=1
adc	BX, numhigh	; Сложение старших частей с
		; переносом.
		; BX:AX=0008:0004h. Число 80004h=
		; 524292

**ADD Целочисленное сложение**

Команда ADD осуществляет сложение первого и второго операндов. Исходное значение первого операнда (приемника)

теряется, замещаясь результатом сложения. Второй операнд не изменяется. В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака.

**Пример**

mov	BX, 1FFh
mov	CX, 3
add	BX, CX

; BX=2001h

**AND Логическое И**

Команда AND осуществляет логическое (побитовое) умножение первого операнда на второй. Исходное значение первого операнда (приемника) теряется, замещаясь результатом умножения. В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами.

**Пример**

mov	AX, 0FFh
and	AX, 5555h

; AX=0554h

**CALL Вызов процедуры**

Команда CALL передает управление процедуре (подпрограмме), сохранив перед этим в стеке адрес возврата. Команда RET, которой обычно заканчивается процедура, забирает из стека адрес возврата и возвращает управление на команду, следующую за командой CALL.

Команда CALL имеет четыре модификации:

- вызов прямой ближний (в пределах текущего программного сегмента);
- вызов прямой дальний (вызов процедуры, расположенной в другом программном сегменте);
- вызов косвенный ближний;
- вызов косвенный дальний.

Команда CALL прямого ближнего вызова заносит в стек относительный адрес точки возврата в текущем программном сегменте и модифицирует IP так, чтобы в нем содержался относительный адрес точки перехода в том же программном сегменте. Необходимая для вычисления этого адреса величина смещения от точки возврата до точки перехода содержится в коде

команды, который занимает 3 байта (код операции E8h и смещение к точке перехода).

Команда CALL прямого дальнего вызова заносит в стек два слова - сначала сегментный адрес текущего программного сегмента, а затем (выше, в слово с меньшим адресом) относительный адрес точки возврата в текущем программном сегменте. Далее модифицируются регистры IP и CS: в IP помещается относительный адрес точки перехода в том сегменте, куда осуществляется переход, а в CS - сегментный адрес этого сегмента. Обе эти величины берутся из кода команды, который занимает 5 байтов (код операции 9Ah, относительный адрес вызываемой процедуры и ее сегментный адрес).

Косвенные вызовы отличаются тем, что адрес перехода извлекается не из кода команды, а из ячеек памяти или регистров; в коде команды содержится информация о том, где находится адрес перехода. Поэтому длина кода команды зависит от используемого способа адресации.

#### Примеры прямых ближних вызовов

```
call near ptr sub1 ;Прямой ближний вызов процедуры
                    ;sub1
                    ;То же самое
call sub1
```

#### Примеры косвенных ближних вызовов

```
call BX ;Косвенный ближний вызов
        ;процедуры, адрес которой
        ;находится в BX
call DS:sub1addr ;Косвенный ближний вызов
                 ;процедуры, адрес которой
                 ;находится в ячейке
                 ;sub1addr, объявленной
                 ;с помощью оператора dw
call word ptr [SI] ;Косвенный ближний вызов
                  ;процедуры, адрес которой
                  ;находится в ячейке, на
                  ;которую указывает регистр SI
call word ptr points[BX] ;Косвенный ближний вызов
                         ;процедуры, адрес которой
                         ;находится в массиве
                         ;адресов points. Смещением
                         ;к адресу процедуры служит
                         ;содержимое регистра BX
call word ptr ES:[BX][SI] ;Косвенный ближний вызов
                          ;процедуры, адрес которой
                          ;находится в дополнительном
                          ;сегменте данных, адресуемом через
                          ;ES. Адрес ячейки с адресом
                          ;процедуры определяется
                          ;суммированием содержимого
                          ;регистров BX и SI
```

#### Примеры прямых дальних вызовов

```
call far ptr sub2
```

;Прямой дальний вызов процедуры  
;sub2, расположенной в другом  
;программном сегменте  
;То же самое, но процедура sub2  
;должна быть объявлена дальней и  
;ее сегмент должен располагаться  
;в программе перед сегментом с  
;командой call

#### Примеры косвенных дальних вызовов

```
call DS:sub2addr
```

;Косвенный дальний вызов  
;процедуры, двухсловный адрес  
;которой находится в ячейке  
;sub2addr, объявленной

```
call dword ptr [BX]
```

;с помощью оператора dd  
;Косвенный дальний вызов  
;процедуры, двухсловный адрес

```
call dword ptr ES:list[DI]
```

;которой находится в ячейке,  
;на которую указывает регистр BX  
;Косвенный дальний вызов  
;процедуры, двухсловный адрес  
;которой находится в массиве list,  
;расположенном в дополнительном  
;сегменте данных, адресуемом  
;через ES. Смещение в массиве к  
;двухсловной ячейке с адресом  
;процедуры находится в регистре DI

#### CBW Преобразование байта в слово

Команда CBW заполняет регистр AH знаковым битом числа, находящегося в регистре AL, что дает возможность выполнять арифметические операции над исходным операндом-байтом как над словом в регистре AX.

#### Примеры

```
mov AL,5
cbw
mov AL,-2 ;AX=0005h
cbw       ;AL=FEh=-2 (байт)
          ;AX=FFFEh=-2 (слово)
```

#### CLC Сброс флага переноса

Команда CLC сбрасывает флаг переноса CF в регистре флагов.

#### Пример

```
clc ;флаг переноса сбрасывается
```

#### CLD Сброс флага направления

Команда CLD сбрасывает флаг направления DF в регистре флагов, устанавливая прямое (в порядке возрастания адресов) направление выполнения операций со строками (цепочками).



**Пример** `cld` ;Флаг направления сбрасывается

**CLI** Сброс флага прерываний  
Команда CLI сбрасывает флаг разрешения прерываний IF в регистре флагов, запрещая (до его установки) все аппаратные прерывания (от таймера, клавиатуры, дисков и т.д.).

**Пример** `cld` ;Запрет аппаратных прерываний

**CMC** Инвертирование флага переноса  
Команда CMC изменяет значение флага переноса CF в регистре флагов на обратное.

**Пример** `cmc` ;Состояние флага CF изменяется на обратное

### CMPS Сравнение

Команда CMP выполняет вычитание второго операнда из первого. В соответствии с результатом вычитания устанавливаются состояния флагов CF, PF, AF, ZF, SF и OF. Сами операнды не изменяются. Таким образом, если команду сравнения записать в общем виде

`cmp операнд_1, операнд_2`  
то ее действие можно условно изобразить следующим образом:  
`операнд_1 - операнд_2 -> флаги процессора`

В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака. Обычно вслед за командой CMP стоит одна из команд условных переходов, анализирующих состояние флагов процессора.

### Пример

```
cmp     AX, 5620h
je      equal      ;Переход на метку equal,
                   ;если AX=5620h
ja      above      ;Переход на метку above,
                   ;если содержимое AX,
                   ;рассматриваемое как число без
                   ;знака, превышает беззнаковое
                   ;число 5620h
```

### CMPSB Сравнение строк

### CMPSB Сравнение строк по байтам

### CMPSW Сравнение строк по словам

Команды предназначены для операций над строками (строкой называется последовательность байтов или слов памяти с любым содержанием). Они сравнивают по одному элементу каждой строки, фактически осуществляют вычитание второго операнда из первого и устанавливая в соответствии с результатом вычитания флаги CF, PF, AF, ZF, SF и OF. Первый операнд адресуется через DS:SI, второй - через ES:DI. Таким образом, операцию сравнения можно условно изобразить следующим образом:

`(DS:SI) - (ES:DI) -> флаги процессора`  
После каждой операции сравнения регистры SI и DI получают положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1 или 2 в зависимости от размера сравниваемых элементов.

Вариант команды CMPS имеет формат

`cmps строка_1, строка_2`  
(что не избавляет от необходимости инициализировать регистры DS:SI и ES:DI адресами строк строка\_1 и строка\_2 соответственно). В этом формате возможна замена сегмента первой строки:

`cmps ES:строка_1, строка_2`  
Рассматриваемые команды могут предваряться префиксами повторения REPE/REPZ (повторять, пока элементы равны, т.е. до первого неравенства) и REPNE/REPZ (повторять, пока элементы не равны, т.е. до первого равенства). В любом случае выполняется не более CX операций над последовательными элементами.

После выполнения рассматриваемых команд регистры SI и DI указывают на ячейки памяти, находящиеся за теми (если DF=0) или перед теми (DF=1) элементами строк, на которых закончились операции сравнения.

### Пример 1

В полях данных основного сегмента данных, адресуемого через DS:

```
str1 db 128 dup (?) ;1-я строка
```

В полях данных дополнительного сегмента данных, адресуемого через ES:

```
str2 db 128 dup (?) ;2-я строка
```

В программном сегменте:

```
cld ;Сравнение вперед
lea SI, str1 ;DS:SI -> str1
```

```

lea    DI, str2      ;ES:DI -> str2
mov     CX, 128       ;Длина сравниваемых строк
cmpsb   ;Поиск различия в строках
je      equal        ;Переход, если строки совпадают

```

**Пример 2**

В полях данных основного сегмента данных, адресуемого через DS:

```

str1    db    11 dup (?) ;1-я строка
str2    db    11 dup (?) ;2-я строка

```

В программном сегменте:

```

cld      ;Сравнение вперед
push     DS ;Передача содержимого DS
pop      ES ;в ES
lea      SI, str1 ;ES:SI -> str1
lea      DI, str2 ;ES:DI -> str2
mov     CX, 11 ;Длина сравниваемых строк
cmpsb   ;Поиск первой пары одинаковых
        ;элементов
jne      notequ ;Переход, если таковой нет

```

**Пример 3**

В полях данных дополнительного сегмента данных, адресуемого через ES:

```

str1    db    1000 dup (?) ;1-я строка
str2    db    1000 dup (?) ;2-я строка

```

В программном сегменте:

```

cld      ;Сравнение вперед
lea      SI, str1 ;ES:SI -> str1
lea      DI, str2 ;ES:DI -> str2
mov     CX, 1000 ;Длина сравниваемых строк
cmps     ;Поиск различия в строках.
        ;Поскольку строки объявлены
        ;директивой DB, фактически
        ;выполняется команда CMPSB
        ;Переход, если строки совпадают
je      equal

```

**CWD Преобразование слова в двойное слово**

Команда CWD заполняет регистр DX знаковым битом содержимого регистра AX, преобразуя тем самым 16-разрядное число со знаком в 32-разрядное. Команду удобно использовать для преобразования двухбайтового делимого в четырехбайтовое (двойное слово) при делении на 16-разрядный операнд.

**Пример 1**

```

mov     AX, 32767 ;AX=7FFFh
cwd     ;AX=7FFFh, DX=0000h. DX:AX=32767

```

**Пример 2**

```

mov     AX, -32768 ;AX=8000h.
cwd     ;AX=8000h, DX=FFFFh. DX:AX=-32768

```

**DAA Десятичная коррекция в регистре AL** после сложения AL двух упакованных десятичных чисел (по одной цифре в каждом полубайте), чтобы получить пару правильных упакованных десятичных цифр. Команда используется вслед за операцией сложения упакованных десятичных чисел. Если результат сложения превышает 99, возникает перенос и устанавливается флаг CF.

**Пример**

```

mov     AL, 87h
add     AL, 04h
daa

```

;Упакованное десятичное 87  
;После сложения AL=8Bh  
;AL=91h, т.е. упакованное  
;десятичное 91

**DAS Десятичная коррекция в регистре AL** после вычитания AL двух упакованных десятичных чисел (по одной цифре в каждом полубайте), чтобы получить пару правильных упакованных десятичных цифр. Команда используется вслед за операцией вычитания упакованных десятичных чисел. Если для вычитания требовался заем, устанавливается флаг CF.

**Пример**

```

mov     AL, 55h
sub     AL, 19h
das

```

;Упакованное десятичное 55  
;После вычитания AL=3Ch  
;AL=36h, т.е. упакованное  
;десятичное 36

**DEC Декремент (уменьшение на 1)**

Команда DEC вычитает 1 из операнда, в качестве которого можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение. Операнд интерпретируется как число без знака.

**Пример 1**

```

mov     AX, 0FFFFh
dec     AX

```

;AX=FFFEh

**Пример 2**

```

mov     CX, 0
dec     CX

```

;CX=FFFFh=-1

**DIV Деление целых беззнаковых чисел**

Команда DIV выполняет деление целого числа без знака, находящегося в регистрах AX (в случае деления на байт) или DX:AX (в случае деления на слово), на операнд-источник (целое число без знака). Размер делимого в два раза больше размеров делителя и остатка.

Для однобайтовых операций делимое помещается в регистр AX; после выполнения операции частное записывается в регистр AL, а остаток - в регистр AH.

Для двухбайтовых операций делимое помещается в регистры DX:AX (в DX - старшая часть, в AX - младшая); после выполнения операции частное записывается в регистр AX, а остаток - в регистр DX.

В качестве операнда-делителя можно указывать регистр данных или ячейку памяти; не допускается деление на непосредственное значение. Если делитель равен 0, или если частное не помещается в назначенный регистр, возбуждается прерывание с вектором 0.

**Пример 1**

mov	AX, 506	; Делимое
mov	BL, 50	; Делитель
div	BL	; AL=0Ah (частное), AH=06h ; (остаток)

**Пример 2**

mov	DX, 1	; Старшая часть делимого 65537
mov	AX, 1	; Младшая часть делимого 65537
mov	CX, 256	; Делитель
div	CX	; AX=0100h (частное), DX=0001h ; (остаток)

### HLT Останов

Команда HLT прекращает выполнение программы и переводит процессор в состояние останова. Работа процессора возобновляется после операции запуска, а также в случае прихода немаскируемого или разрешенного маскируемого прерываний.

### IDIV Деление целых знаковых чисел

Команда IDIV выполняет деление целого числа со знаком, находящегося в регистрах AX (в случае деления на байт) или DX:AX (в случае деления на слово), на операнд-источник (целое число со знаком). Размер делимого в два раза больше размеров делителя и остатка. Оба результата рассматриваются как числа со знаком, причем знак остатка равен знаку делимого.

Для однобайтовых операций делимое помещается в регистр AX; после выполнения операции частное записывается в регистр AL, а остаток - в регистр AH.

Для двухбайтовых операций делимое помещается в регистры DX:AX (в DX - старшая часть, в AX - младшая); после выполнения операции частное записывается в регистр AX, а остаток - в регистр DX.

В качестве операнда-делителя можно указывать регистр данных или ячейку памяти; не допускается деление на непосредственное значение. Если делитель равен 0, или если частное

не помещается в назначенный регистр, возбуждается прерывание через вектор 0.

**Пример 1**

mov	AX, 506	
mov	BL, 50	
div		; Делимое ; Делитель ; AL=0Ah (частное), AH=06h ; (остаток)

**Пример 2**

mov	DX, 1	
mov	AX, 1	
mov	CX, 256	
div		; Старшая часть делимого 65537 ; Младшая часть делимого 65537 ; Делитель ; AX=0100h (частное), DX=0001h ; (остаток)

**Пример 3**

mov	AX, -506	
mov	BL, 50	
idiv		; AX=FEBh, Делимое ; Делитель ; AL=F6h (-10), AH=F6h (-6)

### IMUL Умножение целых знаковых чисел

Команда IMUL выполняет умножение целого знакового числа, находящегося в регистре AL (в случае деления на байт) или AX (в случае деления на слово), на операнд-источник (целое число со знаком). Размер произведения в два раза больше размера сомножителей.

Для однобайтовых операций один из сомножителей помещается в регистр AL; после выполнения операции произведение записывается в регистр AX.

Для двухбайтовых операций один из сомножителей помещается в регистр AX; после выполнения операции произведение записывается в регистры DX:AX (в DX - старшая часть, в AX - младшая).

В качестве операнда-сомножителя можно указывать регистр данных или ячейку памяти; не допускается умножение на непосредственное значение.

**Пример 1**

mov	AL, 5	; Первый сомножитель
mov	BL, 3	; Второй сомножитель
imul	BL	; AX=000Fh (произведение)

**Пример 2**

mov	AX, 256	; Первый сомножитель
mov	BX, 256	; Второй сомножитель
imul	BX	; DX=0001h, AX=0000h ; (число 65536)

**Пример 3**

mov	AL, -5	; AL=F6h
mov	BL, 3	; BL=03h
imul	BL	; AX=FFF1h (-15)



**IN Ввод из порта**

Команда IN вводит в регистр AL или AX соответственно байт или слово из порта, указываемого вторым операндом. Адрес порта помещается в регистр DX. Если адрес порта не превышает 255, он может быть указан непосредственным значением. Указание регистра-приемника (AL или AX) обязательно.

**Пример 1**  
 in AL, 60h ; Ввод байта из порта 60h

**Пример 2**  
 mov DX, 305h ; Адрес порта  
 in AX, DX ; Ввод слова из порта 305h

**INC Инкремент (увеличение на 1)**

Команда INC прибавляет 1 к операнду, в качестве которого можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение. Операнд интерпретируется как число без знака.

**Пример 1**  
 mov AX, 0563h  
 inc AX ; AX=0564h

**Пример 2**  
 mov BH, 15h  
 inc BH ; BH=16h

**Пример 3**  
 mov AX, A5FFh  
 inc AL ; AX=A500h  
 inc AH ; AX=A600h

**INT Программное прерывание**

Команда INT инициирует в процессоре процедуру прерывания, в результате которой управление передается на программу обработки прерывания с номером n, который указан в качестве операнда команды INT. В стек прерываемого процесса (текущей программы) заносится содержимое регистра флагов, сегментного регистра CS и указателя команд IP, после чего в регистры IP и CS передается содержимое двух слов из вектора прерывания типа n (расположенных по адресам 0:n\*4 и 0:n\*4+2).

**Пример 1**  
 int 60h ; Переход на пользовательскую  
 ; программу обработки прерывания  
 ; через вектор 60h

**Пример 2**  
 int 21h ; Переход в MS-DOS

**INTO Прерывание по переполнению**

Команда INTO, будучи установлена вслед за какой-либо арифметической, логической или строковой командой, возбуждает процедуру прерывания типа 4, если предшествующая команда установила флаг переполнения OF. Перед использованием команды INTO пользователь должен поместить в вектор прерывания по переполнению адрес своей программы обработки прерывания.

**Пример**

add AX, BX  
 into

; Произвольная команда  
 ; Переход на пользовательскую  
 ; процедуру в случае переполнения

**IRET Возврат из программы обработки прерывания**

Команда IRET возвращает управление прерыванному в результате аппаратного или программного прерывания процессу. Команда извлекает из стека три верхние слова и помещает их в регистры IP, CS и флагов (см. команду INT). Командой IRET должна завершаться любая программа обработки прерывания.

**Jcc Команды условных переходов**

Команды, обозначаемые (в книгах, не в программах!) Jcc, осуществляют переход по указанному адресу при выполнении условия, заданного мнемоникой команды. Если заданное условие не выполняется, переход не осуществляется, а выполняется команда, следующая за командой Jcc. Переход может осуществляться как вперед, так и назад в диапазоне +127...-128 байтов.

В составе команд процессора предусмотрены следующие команды условных переходов:

Команда	Перейти, если	Условия перехода
JA	выше	CF=0 и ZF=0
JAE	выше или равно	CF=0
JB	ниже	CF=1
JBE	ниже или равно	CF=1 или ZF=1
JC	перенос	CF=1
JCKZ	CX=0	CX=0
JE	равно	ZF=1
JG	больше	ZF=0 или SF=OF
JGE	больше или равно	SF=OF
JL	меньше	SF не равно OF
JLE	меньше или равно	ZF=1 или SF не равно OF
JNA	не выше	CF=1 или ZF=1
JNAE	не выше и не равно	CF=1
JNB	не ниже	CF=0
JNBE	не ниже и не равно	CF=0 и ZF=0
JNC	нет переноса	CF=0
JNE	не равно	ZF=0

Команда	Перейти, если	Условие перехода
JMG	не больше	ZF=1 или SF не равно OF
JMGE	не больше и не равно	SF не равно OF
JNL	не меньше	SF=OF
JNLE	не меньше и не равно	ZF=0 и SF=OF
JN	нет переполнения	OF=0
JNP	нет четности	PF=0
JNS	знаковый бит равен 0	SF=0
JNZ	не ноль	ZF=0
JO	переполнение	OF=1
JP	есть четность	PF=1
JPE	сумма битов четная	PF=0
JPO	сумма битов нечетная	SF=1
JS	знаковый бит равен 1	ZF=1
JZ	ноль	

**Пример 1**  
 cmp AX,0 ;AX=0?  
 je equal ;Если да, перейти на метку equal

**Пример 2**  
 int 21h ;Вызов системной функции  
 jc error ;Если CF=1 (ошибка), перейти на метку error

### JMP Безусловный переход

Команда JMP передает управление в указанную точку того же или другого программного сегмента. Адрес возврата не сохраняется.

Команда JMP имеет пять разновидностей:

- переход прямой короткий (в пределах -128...+127 байтов);
- переход прямой ближний (в пределах текущего программного сегмента);
- переход прямой дальний (в другой программный сегмент);
- переход косвенный ближний;
- переход косвенный дальний.

Все разновидности переходов имеют одну и ту же мнемонику JMP, хотя и различающиеся коды операций. В некоторых случаях транслятор может определить вид перехода по контексту, в тех же случаях, когда это невозможно, следует использовать атрибутные операторы:

short - прямой короткий переход;  
 near ptr - прямой ближний переход;  
 far ptr - прямой дальний переход;  
 word ptr - косвенный ближний переход;  
 dword ptr - косвенный дальний переход.

### Примеры прямых коротких переходов

jmp short shpoint ;Переход на метку shpoint  
 ;в пределах +127...-128 байтов

jmp shpoint

Примеры прямых ближних переходов  
 jmp point ;То же самое, если shpoint  
 ;находится выше по тексту  
 ;программы

Примеры прямых дальних переходов  
 jmp near ptr point ;Переход на метку point  
 ;в пределах текущего сегмента  
 ;То же самое

Примеры косвенных ближних переходов  
 jmp far ptr farpoint ;Переход на метку farpoint в  
 ;другом программном сегменте  
 ;Переход на метку farpoint в  
 ;другом программном сегменте, если  
 ;farpoint объявлена дальней меткой  
 ;директивой farpoint label far

Примеры косвенных дальних переходов  
 jmp DS:addrnear ;Переход по адресу, хранящемуся  
 ;в ячейке addrnear  
 jmp BX ;Адрес точки перехода в BX  
 jmp word ptr [SI] ;Адрес слова памяти с адресом  
 ;точки перехода в регистре SI  
 jmp word ptr [SI+2] ;Адрес слова памяти с адресом  
 ;точки перехода равен содержимому  
 ;регистра SI плюс 2  
 jmp word ptr table[BX] ;Адрес слова памяти с адресом  
 ;точки перехода находится в  
 ;массиве table и смещен  
 ;относительно его начала на  
 ;величину в регистре BX

Примеры косвенных дальних переходов  
 jmp DS:addrfar ;Адрес точки перехода в  
 ;двухсловной ячейке памяти addrfar  
 jmp dword ptr [DI] ;Адрес двухсловной ячейки  
 ;памяти с полным адресом точки  
 ;перехода в регистре DI  
 jmp dword ptr tblfar[BX] ;Адрес двухсловной ячейки  
 ;памяти с полным адресом точки  
 ;перехода находится в массиве  
 ;tblfar и смещен относительно его  
 ;начала на величину в регистре BX

### LAHF Загрузка флагов в регистр AH

Команда LAHF копирует флаги SF, ZF, AF, PF и CF соответственно в разряды 7, 6, 4, 2 и 0 регистра AH. Значение битов 5, 3 и 1 не определено.

Команда LAHF (совместно с командой SAHF) дает возможность читать и изменять значение флагов процессора, в том числе флагов SF, ZF, AF и PF, которые нельзя изменить непосредственно.

**Пример** `lahf` ;Регистр AH отображает состояние регистра флагов

**LDS** Загрузка указателя с использованием регистра DS.

Команда LDS считывает из памяти по указанному адресу двойное слово (32 разряда), содержащее указатель (полный адрес некоторой ячейки), и загружает младшую половину указателя (т.е. относительный адрес) в указанный в команде регистр, а старшую половину указателя (т.е. сегментный адрес) в регистр DS. Таким образом, команда

эквивалентна следующей группе команд:

```
mov reg, word ptr mem
mov DS, word ptr mem+2
```

В качестве первого операнда команды LDS должен быть указан регистр общего назначения, в качестве второго - ячейка памяти.

**Пример 1**

В полях данных:

```
string db ...
addr dd string
```

В программном сегменте:

```
lds SI, addr ;DS:SI->string
```

**Пример 2**

В полях данных:

```
mem dw ...
memaddr dd mem
```

В программном сегменте:

```
lea BX, memaddr ;BX=адрес указателя ячейки mem
...
lds BX, [BX] ;BX=смещение ячейки mem,
;DS=сегментный адрес ячейки mem
```

**Пример 3**

```
lds DI, dptr[SI] ;Предполагается, что по адресу
; dptr начинается массив
; двухсловных указателей.
; Указатель, загружаемый в DS:DI,
; отстоит от начала массива на
; величину в SI (кратную 4)
```

**LEA** Загрузка исполнительного адреса

Команда LEA загружает в регистр, указанный в команде в качестве первого операнда, относительный адрес второго операнда (не значение операнда!). В качестве первого операнда следует указывать регистр общего назначения (не сегментный), в качестве второго - ячейку памяти. Команда

```
lea reg, mem
```

эквивалентна команде

```
mov reg, offset mem
```

но у первой команды больше возможностей описания адреса интересующей нас ячейки.

**Пример**

В полях данных:

```
message db ...
```

В программном сегменте:

```
lea SI, message
```

В команде LEA допустимы и другие способы адресации для описания адреса интересующей нас ячейки памяти:

**Примеры**

```
lea SI, [BX]
```

```
lea DI, table[BX]
```

;Эквивалентно `mov SI, BX`. В BX -  
; адрес требуемой ячейки памяти  
; В DI загружается относительный  
; адрес ячейки массива table  
; со смещением от начала массива,  
; равным содержимому BX

**LES** Загрузка указателя с использованием регистра ES

Команда LES считывает из памяти по указанному адресу двойное слово (32 разряда), содержащее указатель (полный адрес некоторой ячейки), и загружает младшую половину указателя (т.е. относительный адрес) в указанный в команде регистр, а старшую половину указателя (т.е. сегментный адрес) в регистр ES. Таким образом, команда

```
les reg, mem
```

эквивалентна следующей группе команд:

```
mov reg, word ptr mem
mov ES, word ptr mem+2
```

В качестве первого операнда команды должен быть указан регистр общего назначения, в качестве второго - ячейка памяти.

**Пример 1**

В полях данных:

```
string db ...
addr dd string
```

В программном сегменте:

```
les SI, addr ;ES:SI->string
```

**Пример 2**

В полях данных:

```
mem dw ...
memaddr dd mem
```

В программном сегменте:

```
lea BX, memaddr ;BX=адрес указателя ячейки mem
```



```

leax    bx,[bx]          ;BX=смещение ячейки mem,
                           ;ES=сегментный адрес ячейки mem

Пример 3
leax    di,dptr[SI]      ;Предполагается, что по адресу
                           ;dptr начинается массив
                           ;двухсловных указателей.
                           ;Указатель, загружаемый в ES:DI,
                           ;отстоит от начала массива
                           ;на величину в SI (кратную 4)

```

### LODS Загрузка строки

### LODSB Загрузка строки по байтам

### LODSW Загрузка строки по словам

Команды предназначены для операций над строками (строкой называется последовательность байт или слов памяти с любым содержанием). Они загружают в регистр AL (в случае операций над байтами) или AX (в случае операций над словами) содержимое ячейки памяти по адресу, находящемуся в паре регистров DS:SI. После операции загрузки регистр SI получает положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1 или 2 в зависимости от размера загружаемого элемента.

Вариант команды LODS имеет формат

```

lodsb   строка
(что не избавляет от необходимости инициализировать регистры
DS:SI адресом строки). В этом формате возможна замена сег-
мента строки строка:
lodsb   ES:строка

```

### Пример 1

В полях данных основного сегмента данных, адресуемого через DS:

```
str     db     'qwertyuiop'
```

В программном сегменте:

```

cld                     ;Двигаемся по строке вперед
mov     SI,offset str   ;Адрес строки
add     SI,BX            ;Добавим смещение (пусть BX=4)
lodsb                     ;AL='t', SI -> 'y'

```

### Пример 2

В полях данных дополнительного сегмента данных, адресуемого через ES:

```
str     db     'qwertyuiop'
```

В программном сегменте:

```

cld                     ;Двигаемся по строке вперед
mov     SI,offset str   ;Адрес строки
lodsb   ES:str          ;AL='q', ES:SI -> 'w'

```

**LOOP Циклическое выполнение, пока содержимое CX не равно нулю**  
 Команда LOOP выполняет декремент содержимого регистра CX и если оно не равно 0 осуществляет переход на указанную метку вперед или назад в том же программном сегменте в диапазоне -128...+127 байтов. Содержимое регистра CX рассматривается как целое число без знака, поэтому максимальное число повторений группы включенных в цикл команд составляет 65536 (если перед входом в цикл CX=0).

### Пример 1

В полях данных:

```
dw      4096 dup (?)
```

В программном сегменте:

```

lea     BX,maz
xor     SI,SI
mov     CX,4096
mov     AX,0
mov     [BX][SI],AX
inc     SI
inc     SI
loop    nulmas

```

;Массив из 4096 слов

;BX->maz

;SI=0

;Счетчик повторений

;Заполнитель

;Очистка элемента массива

;Сдвиг к следующему

;слову массива

;Повторить CX раз

### Пример 2

```

xor     CX,CX
begin:  loop    begin

```

;CX=0

;65536 повторений

**LOOPE/LOOPZ Цикл, пока равно/цикл, пока нуль**  
 Оба обозначения представляют синонимы и относятся к одной команде. Команда выполняет декремент содержимого регистра CX и если оно не равно 0 и флаг ZF установлен осуществляет переход на указанную метку вперед или назад в том же программном сегменте в диапазоне -128...+127 байтов. Содержимое регистра CX рассматривается как целое число без знака, поэтому максимальное число повторений группы включенных в цикл команд составляет 65536.

### Пример

В полях данных:

```
command db     80 dup (' ')
```

В программном сегменте:

```

lea     SI,command
cld
mov     CX,80
lodsb
cmp     AL,' '
loop   pass
dec     SI

```

;Настроим DS:SI

;Обработка вперед

;Обрабатывать не более 80 байтов

;Загрузим в AL очередной символ

;Пропустим все пробелы в

;начале строки

;DS:SI -> первый символ, отличный от пробела

**LOOPNE/LOOPNZ** Цикл пока не равно/цикл пока не нуль. Оба обозначения представляют синонимы и относятся к одной команде. Команда выполняет декремент содержимого регистра CX и если оно не равно 0 и флаг ZF сброшен осуществляет переход на указанную метку вперед или назад в том же диапазоне -128...+127 байтов. Содержимое регистра CX рассматривается как целое число без знака, поэтому максимальное число повторений группы включенных в цикл команд составляет 65536.

#### Пример

В полях данных:

```
command db 80 dup (0)
```

В программном сегменте:

```
lea SI,command
cld
mov CX,80
slash: lodsb
      cmp AL,'/'
      loopne slash
```

;DS:SI -> первый символ за знаком /

#### MOV Пересылка данных

Команда MOV замещает первый операнд (приемник) вторым (источником). При этом исходное значение первого операнда теряется. В зависимости от описания операндов пересылается слово или байт. Если операнды описаны по-разному или режим адресации не позволяет однозначно определить размер операнда, для уточнения размера передаваемых данных в команду следует включить один из атрибутивных операторов byte ptr или word ptr. В зависимости от использованных режимов адресации команда MOV может осуществлять пересылки следующих видов:

- из регистра общего назначения в регистр общего назначения;
- из ячейки памяти в регистр общего назначения;
- из регистра общего назначения в ячейку памяти;
- непосредственный операнд в регистр общего назначения;
- непосредственный операнд в ячейку памяти;
- из регистра общего назначения в сегментные регистры DS, ES и SS;
- из сегментного регистра в регистр общего назначения;
- из сегментного регистра в ячейку памяти.

Запрещены пересылки из ячейки памяти в ячейку памяти (для этого предусмотрена команда MOVS), а также загрузка сегментного регистра непосредственным значением, которое,

таким образом, приходится загружать через регистр общего назначения:

```
mov AX,seg mem
mov DS,AX
```

;Сегментный адрес ячейки mem

;Загрузка его в регистр DS

Нельзя также непосредственно переслать содержимое одного сегментного регистра в другой. Такого рода операции удобно выполнять с использованием стека:

```
push DS
pop ES
```

;Содержимое DS копируется в ES

#### Примеры

В полях данных:

```
memb db 5,6
memw dw 1000
memd dd memw
```

В программном сегменте:

```
mov AX,-1
```

;Двухсловный адрес ячейки memw

```
mov DI,AX
```

;Непосредственное значение в регистр

```
mov AL,memb
```

;Из регистра в регистр

```
mov memb,CL
```

;Из памяти в регистр (число 5)

```
mov CX,[SI][BX]+6
```

;Из регистра в память (на место числа 5)

```
mov DX,word ptr memb
```

;Из памяти в регистр

```
mov BX,word ptr memd
```

;Слово из памяти в регистр

```
mov ES,word ptr memd+2
```

;Слово из памяти в сегментный регистр (сегментный адрес ячейки memw)

```
mov byte ptr ES:[80],10
```

;Слово из памяти в сегментный регистр (сегментный адрес ячейки memw)

```
mov word ptr ES:[80],10
```

;Непосредственное значение (байт 0Ah) в память

```
mov DS,AX
```

;Непосредственное значение (слово 000Ah) в память

```
mov BX,ES
```

;Из регистра в сегментный регистр

```
mov SS,[SI]
```

;Из сегментного регистра в регистр

;Из памяти в сегментный регистр

#### MOVS Пересылка данных из строки в строку

#### MOVSB Пересылка байта данных из строки в строку

#### MOVSW Пересылка слова данных из строки в строку

Команды предназначены для операций над строками (строкой называется последовательность байтов или слов памяти с любым содержанием). Они пересылают по одному элементу строки, который может быть байтом или словом. Первый операнд (приемник) адресуется через ES:DI, второй (источник) - через DS:SI. Операцию пересылки можно условно изобразить следующим образом:

(DS:SI) -> (ES:DI)

После каждой операции пересылки регистры SI и DI получают положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1 или 2 в зависимости от размера пересылаемых элементов.

Вариант команды MOVS имеет формат:

movs строка\_1, строка\_2  
(что не избавляет от необходимости инициализировать регистры ES:DI и DS:SI адресами строк строка-1 и строка-2). В этом формате возможна замена сегмента второй строки:

movs строка\_1, ES:строка\_2  
Рассматриваемые команды могут предваряться префиксом повторения REP (повторять CX раз). После выполнения рассматриваемых команд регистры SI и DI указывают на ячейки памяти, находящиеся за теми (если DF=0) или перед теми (если DF=1) элементами строк, на которых закончились операции пересылки.

### Пример 1

В полях данных основного сегмента данных, адресуемого через DS:

```
DS:
txt db 'Страница ' ;Пересылаемая строка
txt_len equ $-txt ;Ее длина
```

В полях данных дополнительного сегмента данных, адресуемого через ES:

```
string db 80 dup (' ')
```

В программном сегменте:

```
lea SI,txt ;DS:SI -> txt
lea DI,string+10 ;ES:DI -> string+10
cld ;Движение по строке вперед
mov CX,txt_len ;Столько байтов переслать
rep movsb ;Пересылка
```

### Пример 2

В полях данных основного сегмента данных, адресуемого через DS:

```
DS:
txt db 'A',84h,'B',84h,'A',84h,'P',84h,'И',84h,'Я',84h,'I',84h
txt_len=$-txt
```

В программном сегменте:

```
;Наложим дополнительный сегмент данных на текстовый видеобuffer
;{адрес B8000h}
mov AX,B8000h ;Сегментный адрес видеобuffer
mov ES,AX ;Инициализируем ES

;Выведен на экран текст
mov DI,1672 ;Смещение к середине экрана
lea SI,txt ;DS:SI -> txt
cld ;Движение по строке вперед
mov CX,txt_len/2 ;Столько слов переслать
```

rep movsw  
красной

;Пересылка в середину экрана

;мерцающая (атрибут 84h) надпись  
;"АВАРИЯ!"

### Пример 3

В полях данных основного сегмента данных, адресуемого через DS:

```
cmdtail db 128 dup (?)
```

В программном сегменте:

;Заберем из префикса программного сегмента (PSP) хвост (параметры) команд. После загрузки программы в память ES указывает на PSP

```
push ES ;Перешлем через стек
push DS ;содержимое ES в DS,
pop ES ;а содержимое DS в ES
pop DS
```

;Теперь DS -> PSP, ES -> основной сегмент данных

```
lea DI,cmdtail ;ES:DI -> приемник хвоста
mov SI,80h ;Хвост начинается в PSP с байта
```

80h

```
mov AL,[SI]
```

;Первый байт - длина хвоста

;Преобразуем его в слово

;и отправим в CX

```
mov CX,AX
```

```
inc SI
```

;DS:SI -> начало хвоста

;Перешлем хвост в cmdtail для

;последующего анализа

;Восстановим нормальную

;адресуемость

;{ES -> PSP, DS -> основной

;сегмент данных}

```
push DS
```

```
push ES
```

```
pop DS
```

```
pop ES
```

### MUL Умножение целых беззнаковых чисел

Команда MUL выполняет умножение целого беззнакового числа, находящегося в регистре AL (в случае умножения на байт) или AX (в случае умножения на слово), на операнд-источник (целое число без знака). Размер произведения в два раза больше размера сомножителей.

Для однобайтовых операций один из сомножителей помещается в регистр AL; после выполнения операции произведение записывается в регистр AX.

Для двухбайтовых операций один из сомножителей помещается в регистр AX; после выполнения операции произведение записывается в регистры DX:AX (в DX - старшая часть, в AX - младшая).

В качестве операнда-сомножителя можно указывать регистр данных или ячейку памяти; не допускается умножение на непосредственное значение.

### Пример 1

```
mov AL,5
mov BL,3
```

;Первый сомножитель

;Второй сомножитель



Пример 2

mov	AX, 256	:Первый сомножитель
mov	BX, 256	:Второй сомножитель
mul	BX	:DX=0001h, AX=0000h
		: (число 65536)

Пример 3

mov	AL, 251	:AL=FBh
mov	BL, 3	:BL=03h
mul	BL	:AX=02F1h

### NEG Изменение знака, дополнение до 2

Команда NEG выполняет вычитание знакового целочисленного операнда из нуля, превращая положительное число в отрицательное и наоборот. В качестве операнда можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение.

#### Примеры

mov	AX, 0001	
neg	AX	:AX=FFFFh=-1
mov	BX, -2	:BX=FFFEh=-2
neg	BX	:BX=0002h

### NOP Холодная команда

По команде NOP процессор не выполняет никаких действий, кроме увеличения на 1 (поскольку команда NOP занимает 1 байт) содержимого указателя команд IP. Команда иногда используется в отладочных целях чтобы "забить" какие-то ненужные команды, не изменяя длину загрузочного модуля или, наоборот, оставить место в загрузочном модуле для последующей вставки команд. В ряде случаев команды NOP включаются в текст объектного модуля транслятором.

### NOT Инверсия, дополнение до 1

Команда NOT выполняет инверсию битов указанного операнда, заменяя 0 на 1 и наоборот. В качестве операнда можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение.

#### Примеры

mov	AX, 0FFFFh	
not	AX	:AX=0000h
mov	SI, 5551h	
not	SI	:SI=AAAEh

### OR Логическое ВКЛЮЧАЮЩЕЕ ИЛИ

Команда OR выполняет операцию логического (побитового) сложения двух операндов. Результат замещает первый операнд (приемник); второй операнд (источник) не изменяется. В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами.

#### Пример

mov	AX, 0FA01h
mov	BX, 1C15h
or	AX, BX

:AX=FE15h, BX=1C15h

### OUT Вывод в порт

Команда OUT выводит в порт, указываемый первым операндом, байт или слово соответственно из регистра AL или AX. Адрес порта помещается в регистр DX. Если адрес порта не превышает 255, он может быть указан непосредственным значением. Указание регистра-источника (AL или AH) обязательно.

#### Пример 1

out	61h, AL
-----	---------

:Вывод байта в порт 61h из AL

#### Пример 2

mov	DX, 3CEh
mov	AL, 5
out	DX, AL

:Адрес порта

:Данное

:Вывод байта из AL в порт 3CEh

### POP Извлечение слова из стека

Команда POP пересылает слово из вершины стека (на которую указывает регистр SP) по адресу операнда-приемника. Затем содержимое SP увеличивается на 2 и указывает на новую вершину стека.

В качестве операнда-приемника можно использовать любой 16-разрядный регистр (кроме CS) или ячейку памяти.

#### Пример

push	CS
pop	DS

:Теперь DS=CS

### POPF Восстановление из стека регистра флагов

Команда POPF пересылает определенные биты слова из вершины стека (на которую указывает регистр SP) в регистр флагов. Затем содержимое SP увеличивается на 2 и указывает на новую вершину стека. Команда воздействует на все флаги процессора.

## Пример

```
popf
```

;Регистр флагов загружается из стека

**PUSH** Занесение операнда в стек

Команда **PUSH** уменьшает на 2 содержимое указателя стека **SP** и заносит на эту новую вершину содержимое двухбайтового операнда-источника.

В качестве операнда-источника может использоваться любой 16-разрядный регистр (включая сегментные) или ячейка памяти.

## Пример

```
push AX
push CS
push ES:mem
```

```
;Сохранение AX
;Сохранение CS
;Сохранение содержимого слова
;памяти mem из дополнительного
;сегмента
```

**PUSHF** Занесение в стек содержимого регистра флагов

Команда **PUSHF** уменьшает на 2 содержимое указателя стека **SP** и заносит на эту новую вершину содержимое регистра флагов.

## Пример

```
pushf
```

```
;Содержимое регистра флагов
;сохраняется в стеке
```

**RCL** Циклический сдвиг влево через бит переноса

Команда **RCL** осуществляет сдвиг влево всех бит операнда. Если команда записана в формате

```
RCL операнд,1
```

сдвиг осуществляется на 1 бит. В младший бит операнда заносится значение флага **CF**; старший бит операнда загружается в **CF**. Если команда записана в формате

```
RCL операнд,CL
```

сдвиг осуществляется на число бит, указанное в регистре-счетчике **CL**, при этом в процессе последовательных сдвигов старшие биты операнда поступают сначала в **CF**, а оттуда - в младшие биты операнда.

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение.

## Пример 1

```
clic
mov AX,1
rci AX,1
```

;Сбросим CF

;AX=2; CF=0

## Пример 2

```
mov DL,8
rci DL,1
```

;DL=10h, CF=0

## Пример 3

```
mov BX,0FFFFh
rci BX,1
```

;BX=FFFFh, CF=1

## Пример 4

```
clic
mov DH,3
mov CL,4
rci DH,CL
```

;Сбросим CF

;Счетчик сдвигов  
;DH=30h, CF=0**RCR** Циклический сдвиг вправо через бит переноса

Команда **RCR** осуществляет сдвиг вправо всех бит операнда. Если команда записана в формате

```
RCR операнд,1
```

сдвиг осуществляется на 1 бит. В старший бит операнда заносится значение флага **CF**; младший бит операнда загружается в **CF**.

Если команда записана в формате

```
RCL операнд,CL
```

сдвиг осуществляется на число бит, указанное в регистре-счетчике **CL**, при этом в процессе последовательных сдвигов младшие биты операнда поступают сначала в **CF**, а оттуда - в старшие биты операнда.

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение.

## Пример 1

```
clic
mov AX,2
rci AX,1
```

;Сбросим флаг CF

;AX=1, CF=0

## Пример 2

```
mov DL,8
rci DL,1
```

;DL=4, CF=0

## Пример 3

```
mov BX,0Fh
rci BX,1
```

;BX=7, CF=1

## Пример 4

```
clic
mov DH,80h
mov CL,5
rci DH,CL
```

;Сбросим флаг CF

;Счетчик сдвигов  
;DH=4, CF=0

**RET** Возврат из процедуры

**RETN** Возврат из ближней процедуры

**RETF** Возврат из дальней процедуры

Команда **RET** извлекает из стека адрес возврата и передает управление назад в программу, первоначально вызвавшую процедуру. Если командой **RET** завершается ближняя процедура, объявленная с атрибутом **NEAR**, или используется модификация команды **RETN**, со стека снимается одно слово — относительный адрес точки возврата. Передача управления в этом случае осуществляется в пределах одного программного сегмента. Если командой **RET** завершается дальняя процедура, объявленная с атрибутом **FAR**, или используется модификация команды **RETF**, со стека снимаются два слова: относительный и сегментный адреса точки возврата. В этом случае передача управления может быть межсегментной.

В команду **RET** может быть включен необязательный операнд (кратный 2), который указывает, на сколько байтов дополнительно смещается указатель стека после возврата в вызывающую программу. Прибавляя эту константу к новому значению **SP**, команда **RET** обходит аргументы, помещенные в стек вызывающей программой (для передачи процедуре) перед выполнением команды **CALL**.

#### Пример 1

```

...      call     subr           ;Вызов подпрограммы
subr     proc     near
...
ret
subr     endp

```

#### Пример 2

```

...      push     AX             ;Параметр, передаваемый в
...                                     ;подпрограмму
...      push     SI             ;Адрес, передаваемый в
...                                     ;подпрограмму
...      call     subr           ;Вызов подпрограммы
subr     proc     near
...
;Извлечение из стека (без изменения содержимого SP) параметров
...      ret      4              ;Возврат в вызывающую программу и
...                                     ;снятие со стека двух слов с
...                                     ;параметрами
subr     endp

```

**ROL** Циклический сдвиг влево

Команда **ROL** осуществляет сдвиг влево всех бит операнда. Если команда записана в формате

**ROL** операнд,1  
сдвиг осуществляется на 1 бит. Старший бит операнда загружается в его младший разряд. Если команда записана в формате

**ROL** операнд,CL  
сдвиг осуществляется на число бит, указанное в регистре-счетчике **CL**, при этом в процессе последовательных сдвигов старшие биты операнда перемещаются в его младшие разряды.

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение.

#### Пример 1

```

mov     AX,1
rol     AX,1                     ;AX=0002h

```

#### Пример 2

```

mov     DL,8
rol     DL,1                     ;DL=10h

```

#### Пример 3

```

mov     BX,0FFFFh
rol     BX,1                     ;BX=FFFDh

```

#### Пример 4

```

mov     DH,0C0h
mov     CL,2
rol     DH,CL                   ;DH=3

```

**ROR** Циклический сдвиг вправо

Команда **ROR** осуществляет циклический сдвиг вправо всех бит операнда. Если команда записана в формате

**ROR** операнд,1  
сдвиг осуществляется на 1 бит. Младший бит операнда записывается в его старший разряд. Если команда записана в формате

**ROR** операнд,CL  
сдвиг осуществляется на число бит, указанное в регистре-счетчике **CL**, при этом в процессе последовательных сдвигов младшие биты операнда перемещаются в его старшие разряды.

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение.

#### Пример 1

```

mov     AX,2
ror     AX,1                     ;AX=1

```



Пример 2  
 mov DL, 8 ;DL=4  
 rot DL, 1

Пример 3  
 mov BX, 0Fh ;BX=0007h  
 rot BX, 1

Пример 4  
 mov DH, 0Ch ;DH=81h  
 mov CL, 3  
 rot DH, CL

**SAHF** Запись содержимого регистра AH в регистр флагов

Команда SAHF копирует биты 7, 6, 4, 2 и 0 регистра AH в младший байт регистра флагов, влияя на флаги SF, ZF, AF, PF и CF.

Команда SAHF (совместно с командой LAHF) дает возможность читать и изменять значение флагов процессора, в том числе флагов SF, ZF, AF и PF, которые нельзя изменить непосредственно.

Пример  
 mov AH, 5 ;Устанавливаются флаги PF и CF  
 sahf ;и сбрасываются флаги SF, ZF и AF

**SAL/SHL** Арифметический сдвиг влево/логический сдвиг влево

Команда SAL осуществляет сдвиг влево всех битов операнда. Старший бит операнда поступает в флаг CF. Если команда записана в формате

SAL операнд, 1  
 сдвиг осуществляется на 1 бит. В младший бит операнда загружается 0. Если команда записана в формате

SAL операнд, CL  
 сдвиг осуществляется на число битов, указанное в регистре-счетчике CL, при этом в процессе последовательных сдвигов старшие биты операнда, пройдя через флаг CF, теряются, а младшие заполняются нулями.

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение.

Каждый сдвиг влево эквивалентен умножению знакового числа на 2, поэтому команду SAL удобно использовать для возведения операнда в степень 2.

Поскольку логический сдвиг влево полностью эквивалентен арифметическому сдвигу влево, команда SHL является просто другим обозначением команды SAL.

Пример 1  
 mov AL, 7  
 sal AL, 1 ;AL=0Eh=7\*2

Пример 2  
 mov AX, 1FFh  
 mov CL, 2  
 sal AX, CL ;AX=07FCh=1FFh\*4

Пример 3  
 mov SI, -1  
 mov CL, 4  
 sal SI, CL ;SI=FFFFh  
 ;SI=FFFFh=-1\*16=-16

**SAR** Арифметический сдвиг вправо

Команда SAR осуществляет сдвиг вправо всех битов операнда. Младший бит операнда поступает в флаг CF. Если команда записана в формате

SAR операнд, 1  
 сдвиг осуществляется на 1 бит. Старший бит операнда сохраняет свое значение. Если команда записана в формате

SAR операнд, CL  
 сдвиг осуществляется на число бит, указанное в регистре-счетчике CL, при этом в процессе последовательных сдвигов младшие биты операнда, пройдя через флаг CF, теряются, а старший бит расширяется вправо.

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение.

Каждый сдвиг вправо эквивалентен делению знакового числа на 2, поэтому команду SAR удобно использовать для деления операнда на целые степени 2.

Пример 1  
 mov AL, 7  
 sar AL, 1 ;AL=3=7/2. Остаток потерян

Пример 2  
 mov AX, 1FF0h  
 mov CL, 4  
 sar AX, CL ;AX=01FFh=1FF0h/16

Пример 3  
 mov BX, -8  
 mov CL, 2  
 sar BX, CL ;BX=FFF8h  
 ;BX=FFF8h=-2=-8/4

**SBB** Целочисленное вычитание с займом

Команда SBB вычитает второй операнд (источник) из первого (приемника). Результат замещает первый операнд, предыдущее значение которого теряется. Если установлен флаг CF, из

результата вычитается еще 1. Таким образом, если команду вычитания записать в общем виде

то ее действие можно условно изобразить следующим образом:

В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака. Команда SBB обычно используется для вычитания 32-разрядных чисел.

#### Пример 1

```
mov     AX, 75A5h
sbb     AX, 76A3h
```

;AX=1, если CF был = 1  
;AX=2, если CF был = 0

#### Пример 2

В полях данных:

```
numlow  dw  000Ah
numhigh dw  0001h
```

;Младшая часть вычитаемого  
;Старшая часть вычитаемого  
;Число 1000Ah=65546

В программном сегменте:

```
mov     AX, 0
mov     DX, 0002
sub     AX, numlow
sbb     DX, numhigh
```

;Младшая часть уменьшаемого  
;Старшая часть уменьшаемого  
;Число 20000h=131072  
;Вычитание младших частей.  
;AX=FFF6h, CF=1  
;Вычитание старших частей с  
;займом.  
;DX:AX=0000:FFF6h=65526

SCAS Сканирование строки с целью сравнения

SCASB Сканирование строки байтов с целью сравнения

SCASW Сканирование строки слов с целью сравнения

Команды предназначены для операций над строками (строкой называется последовательность байтов или слов памяти с любым содержанием). Они сравнивают содержимое регистра AL (в случае операций над байтами) или AX (в случае операций над словами) с содержимым ячейки памяти по адресу, находящемуся в паре регистров ES:DI. Операция сравнения осуществляется путем вычитания содержимого ячейки памяти из содержимого AL или AX. Результат операции воздействует на регистр флагов, но не изменяет ни один из операндов. Таким образом, операцию сравнения можно условно изобразить следующим образом:

AX или AL - (ES:DI) -> флаги процессора

После каждой операции сравнения регистр DI получает положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1 или 2 в зависимости от размера сравниваемых элементов.

Вариант команды SCAS имеет формат

scas строка  
(что не избавляет от необходимости инициализировать регистры ES:DI адресом строки строка). Замена сегмента строки невозможна.

Рассматриваемые команды могут предваряться префиксами повторения REPE/REPZ (повторять, пока элементы равны, т.е. до первого неравенства) и REPNE/REPZ (повторять, пока элементы не равны, т.е. до первого равенства). В любом случае выполняется не более CX операций над последовательными элементами.

#### Пример 1

После выполнения рассматриваемых команд регистр DI указывает на ячейку памяти, находящуюся за тем (если DF=0) или перед тем (если DF=1) элементом строки, на котором закончилась операция сравнения.

В полях данных дополнительного сегмента данных, адресуемого через ES:

```
string db 80 dup (?)
```

В программном сегменте:

```
cld
lea     DI, string
mov     AL, ' '
mov     CX, 80
repb    scasb
je       blank
```

;Поиск вперед по строке  
;ES:DI -> string  
;Символ, который мы пропускаем  
;Длина строки  
;Поиск первого элемента, отличного от пробела  
;Элемент найден, ES:DI -> первый элемент за пробелом

#### Пример 2

В полях данных дополнительного сегмента данных, адресуемого через ES:

```
string db 80 dup (?)
```

В программном сегменте:

```
cld
lea     DI, string
mov     AL, ' '
mov     CX, 80
repne   scasb
jne     noblank
```

;Поиск вперед по строке  
;ES:DI -> string  
;Символ, который мы ищем  
;Длина строки  
;Поиск первого пробела в строке  
;Пробелов в строке нет  
;Пробел найден, ES:DI -> первый элемент за пробелом

**SHR** Логический сдвиг вправо

Команда SHR осуществляет сдвиг вправо всех бит операнда. Младший бит операнда поступает в флаг CF. Если команда записана в формате

SHR операнда, 1

сдвиг осуществляется на 1 бит. В старший бит операнда загружается 0, а младший теряется. Если команда записана в формате

SHR операнда, CL

сдвиг осуществляется на число бит, указанное в регистре-счетчике CL, при этом в процессе последовательных сдвигов старшие биты операнда заполняются нулями, а младшие, пройдя через флаг CF, теряются.

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение.

**Пример 1**

```
mov     AL, 7
shr     AL, 1           ;AL=3
```

**Пример 2**

```
mov     AX, 1FFFh
mov     CL, 4
shr     AX, CL          ;AX=01FFh
```

**Пример 3**

```
mov     DX, 0FFFFh
mov     CL, 4
shr     DX, CL          ;DX=0FFFh
```

**STC** Установка флага переноса

Команда STC устанавливает флаг переноса CF в регистре флагов.

**Пример**

```
stc                     ;флаг CF устанавливается
```

**STD** Установка флага направления

Команда STD устанавливает флаг направления DF в регистре флагов, определяя тем самым обратное направление выполнения строковых операций (в порядке убывания адресов элементов строки).

**Пример**

```
std                     ;флаг направления устанавливается
```

**STI** Установка флага прерывания

Команда STI устанавливает флаг разрешения прерываний IF в регистре флагов, разрешая все аппаратные прерывания (от таймера, клавиатуры, дисков и т.д.).

**Пример**

sti

;Разрешение аппаратных прерываний

**STOS** Запись в строку данных**STOSB** Запись байта в строку данных**STOSW** Запись слова в строку данных

Команды предназначены для операций над строками (строкой называется последовательность байтов или слов памяти с любым содержанием). Они копируют содержимое регистра AL (в случае операций над байтами) или AX (в случае операций над словами) в ячейку памяти соответствующего размера по адресу, определяемому содержимым пары регистров ES:DI. После операции копирования регистр DI получает положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1 или 2 в зависимости от размера копируемого элемента.

Вариант команды STOS имеет формат

stos строка

(что не избавляет от необходимости инициализировать регистры ES:DI адресом строки строка). Заменить сегментный регистр ES нельзя.

Рассматриваемые команды могут предваряться префиксом повторения REP. В этом случае они повторяются CX раз, задерживая последовательные ячейки памяти одним и тем же содержимым регистра AL или AX.

**Пример**

В полях данных дополнительного сегмента данных, адресуемого через ES:

```
line dw 80 dup (0)
```

В программном сегменте:

```
mov     AL, '>'
mov     AH, 14h
mov     CX, 5
cld
lea     DI, ES:line
rep     stos line

;Код ASCII знака >
;Атрибут (красный по синему)
;Заполнить 5 слов
;Движение по строке вперед
;ES:DI -> line
;Первые 5 слов строки line
;заполняются кодом ASCII знака >
;вместе с его атрибутом для
;последующего вывода на экран
```

**SUB** Вычитание целых чисел

Команда SUB вычитает второй операнд (источник) из первого (приемника) и помещает результат на место первого операнда. Исходное значение первого операнда (уменьшаемое) теряется. Таким образом, если команду вычитания записать в общем виде

sub операнда\_1, операнда\_2

то ее действие можно условно изобразить следующим образом:



операнда\_1 - операнда\_2 -> операнда\_1

В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака.

#### Примеры

```
sub    CX, 5           ;Выполняется действие CX-5->CX
sub    DH, DL          ;Выполняется действие DH-DL->DH
sub    AX, mem         ;Выполняется действие AX-(mem)->AX
sub    byte ptr mem, DL ;Выполняется действие
                        ;(mem)-DL->mem
sub    table[BX], '0'   ;Выполняется действие
                        ;[table[BX]]-30h -> table[BX]
```

#### TEST Логическое сравнение

Команда TEST выполняет операцию логического И над двумя операндами и в зависимости от результата устанавливает флаги SF, ZF и PF. Флаги OF и CF сбрасываются, а AF имеет неопределенное значение. Состояние флагов можно затем проанализировать командами условных переходов. Команда TEST не изменяет ни один из операндов.

В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака.

Флаг SF устанавливается в 1, если в результате выполнения команды образовалось число с установленным знаковым битом.

Флаг ZF устанавливается в 1, если в результате выполнения команды образовалось число, состоящее из одних двоичных нулей.

Флаг PF устанавливается в 1, если в результате выполнения команды образовалось число с четным количеством двоичных единиц в его битах.

#### Пример 1

```
test    AX, 1
jne     bityes         ;Переход, если бит 0 в AX
                        ;установлен
je      bitno          ;Переход, если бит 0 в AX сброшен
```

#### Пример 2

```
test    DX, 0FFFFh
jz      null           ;Переход, если DX=0
```

#### Пример 3

```
test    CX, 0F000h
jne     bityes
je      bitno
```

;Переход, если какие-либо из 4  
;старших битов CX установлены  
;Переход, если все 4 старших бита  
;CX сброшены

#### Пример 4

```
test    AX, AX
jz      zero
jnz     notzero
```

;Переход, если AX=0  
;Переход, если AX не равно 0

#### WAIT Ожидание

Команда WAIT переводит процессор в состояние ожидания до активизации линии TEST. Команда используется для обеспечения синхронизации с внешними устройствами и, в частности, с арифметическим сопроцессором.

#### XCHG Обмен данными между операндами

Команда XCHG пересылает значение первого операнда во второй, а второго - в первый. В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака.

#### Пример 1

```
mov     AX, 0FF01h
mov     SI, 1000
xchg    AX, SI
```

;AX=3E8h=1000, SI=FF01h

#### Пример 2

##### В полях данных:

```
mem     dw     0F0F0h
```

##### В программном сегменте

```
mov     CX, 1256h
xchg    CX, mem
```

;CX=F0F0h, (mem)=1256h

#### XLAT Табличная трансляция

Команда XLAT осуществляет выборку байта из таблицы. В регистре BX должен находиться относительный адрес таблицы, а в регистре AL - смещение в таблице к выбираемому байту (его индекс). Выбранный байт загружается в регистр AL, заменяя находившееся в нем смещение. Длина таблицы может достигать 256 байтов. Команда XLAT не имеет явных операндов, но требует предварительной настройки регистров BX и AL.

**Пример**

Пример демонстрирует преобразование схем-кодов клавиш верхнего ряда клавиатуры в коды ASCII соответствующих символов

В полях данных:

```
table db 0,27,'1234567890-='
```

В программном сегменте

```
lea bx,table
mov AL,5
xlat
```

;Схем-код 5 клавиши <4/\$>

;AL=34, код ASCII символа 4

**XOR Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ**

Команда XOR выполняет операцию логического (побитового) ИСКЛЮЧАЮЩЕГО ИЛИ над своими двумя операндами. Результат операции замещает первый операнд. Каждый бит результата устанавливается в 1, если соответствующие биты операндов различны, и сбрасывается в 0, если соответствующие биты операндов совпадают.

В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами.

**Пример 1**

```
mov AX,0Fh
xor AX,0FFFFh ;AX=FFFFh
```

**Пример 2**

```
xor BX,BX ;Обнуление BX
```

**Пример 3**

```
mov SI,0AAAAh
mov BX,5555h
xor SI,BX ;SI=FFFFh,BX=5555h
```

**Приложение 5. Основные команды отладчика CodeView Microsoft****Управляющие клавиши**

<Alt> - активизация строки меню в верхней части экрана.  
 <Alt>-F-X - выход из CodeView.  
 <Alt>-R-R - рестарт программы (возвращение ее в исходное состояние для повторного пуска с самого начала).  
 <Ctrl>/G - увеличение размера (каждый раз на одну строку) информационного окна с курсором.  
 <Ctrl>/T - уменьшение размера (каждый раз на одну строку) информационного окна с курсором.  
 <Esc> - возврат из меню.

**Функциональные клавиши**

<F1> - вывод справочника.  
 <F2> - вывод на экран информационного поля с содержанием регистров процессора.  
 <F3> - переключение вида основного информационного кадра (только машинные команды, только исходный текст, и то, и другое).  
 <F4> - переключение на экран DOS и обратно.  
 <F5> - выполнение программы до конца или до точки останова.  
 <F6> - перевод курсора на информационное поле программы или на поле командной строки.  
 <F7> - выполнение программы до курсора или до точки останова.  
 <F8> - выполнение 1 команды; подпрограммы и циклы выполняются команда за командой.  
 <F9> - установка или снятие точки останова в положении курсора.  
 <F10> - выполнение 1 команды; подпрограммы и циклы выполняются, как одна команда (если в них нет точек останова).

**Команды командной строки**

G seg:addr - выполнение программы до адреса seg:addr, точки останова или конца программы. В качестве параметра seg

может использоваться обозначение сегментного регистра или число. По умолчанию `seg=CS`.

`P n` - выполнение `n` команд с выполнением подпрограмм и циклов, как одной команды. По умолчанию `n=1`.

`T n` - выполнение `n` команд со входом в подпрограммы и циклы. По умолчанию `n=1`.

`E` - медленное (команда за командой) выполнение программы до нажатия любой клавиши.

`Dtype seg:addr L nmb` - дамп `nmb` байтов памяти в формате `type` начиная с адреса `seg:addr`. В качестве параметра `seg` может использоваться обозначение сегментного регистра или число. По умолчанию `seg=DS`. Если нет параметров `L` и `nmb`, выводится 2 строки (до 32 байтов). Параметр `type` (тип) может принимать значения: `A` - только коды ASCII, `B` - байты и коды ASCII, `W` - слова. После выполнения одной команды `D` указанный тип остается установленным. Между командой и типом не должно быть пробела.

`Dtype seg:addr1 addr` - дамп памяти от адреса `seg:addr1` до адреса `seg:addr2`.

`Dtype label L nmb` - дамп `nmb` байтов памяти в формате `type` начиная с символического имени `label`. Если нет параметров `L` и `nmb`, выводится 2 строки (до 32 байтов).

`R reg` - вывод содержимого регистра `reg` и запрос на его изменение.

`R reg=n` - занесение в регистр `reg` значения `n`.

`Etype seg:addr n1 n2 ...` - занесение в память начиная с адреса `seg:addr` значений `n1, n2 ...` в формате `type`. По умолчанию `seg=DS`. Между командой и типом не должно быть пробела.

`Nrdx` - изменение системы счисления в параметрах командной строки. Параметр `rdx` может принимать значения 16 и 10. Команда `N` без параметра выводит действующее значение системы счисления.

`BP seg:addr` - установка очередной точки останова по адресу `seg:addr`. По умолчанию `seg=CS`.

`BP seg:addr step` - установка точки останова по адресу `seg:addr` с пропуском `ee` при выполнении первые `step` раз. Команда удобна для отладки многошаговых циклов.

`BL` - вывод списка точек останова с их адресами.

`BC n` - снятие точки останова с номером `n`.

`BC *` - снятие всех точек останова.

`BD n` - выключение (но не снятие) точки останова с номером `n`.

`BE n` - включение (но не установка новой точки останова) точки останова с номером `n`.

## Литература

1. Лю Ю-Чжен, Гибсон Г. Микропроцессоры семейства 8086/88. Архитектура, программирование и проектирование микропроцессорных систем: Пер. с англ. - М.: Радио и связь, 1987. - 512 с.
2. Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера: Пер. с англ. - 2-е изд., стереотип. - М.: Радио и связь, 1991. - 336 с.
3. Дао Л. Программирование микропроцессора 8088: Пер. с англ. - М.: Мир, 1988. - 357 с.
4. Джордейн Р. Справочник программиста персональных компьютеров типа IBM PC, XT и AT: Пер. с англ. - Финансы и статистика, 1992. - 544 с.
5. Данкан Р. Профессиональная работа в MS-DOS: Пер. с англ. - М.: Мир, 1992. - 509 с.
6. Фролов А.В., Фролов Г.В. Операционная система MS-DOS: В 3 кн. - М.: "ДИАЛОГ-МИФИ", 1991-1992.
7. Касаткин А.И. Профессиональное программирование на языке Си. Управление ресурсами: Справочное пособие. - Минск: Выш. шк., 1992. - 432 с.
8. Касаткин А.И. Профессиональное программирование на языке Си. Системное программирование. - Минск: Выш. шк., 1993. - 301 с.
9. Браун Р, Кайл Дж. Справочник по прерываниям для IBM PC: В 2-х томах: Т.1. Пер. с англ. - М.: Мир, 1994. - 558 с. Т.2. Пер. с англ. - М.: Мир, 1994. - 480 с.
10. Финогенов К.Г. MS-DOS 5.0: Издание 2, переработанное и дополненное. - М.: МП "Мадип", 1993. 302 с.
11. Финогенов К.Г., Черных В.И. MS-DOS 6. - М.: ABF, 1993. - 448 с.
12. Финогенов К.Г. MS-DOS 6.2. - М.: ABF, 1994. - 320 с.



## Содержание

Введение .....	3
1. Архитектурные особенности IBM PC .....	6
1.1. Краткий обзор семейства микропроцессоров фирмы Intel .....	6
1.2. Распределение адресного пространства .....	8
1.3. Регистры процессора .....	13
2. Модели памяти и структуры программ .....	19
2.1. Структура и образ памяти программы .EXE .....	19
2.2. Структура и образ памяти программы .COM .....	22
2.3. Задачи по моделям памяти и структурам программ .....	25
3. Основы языка ассемблера .....	28
3.1. Основные определения данных .....	28
3.2. Режимы адресации .....	29
3.3. Основы программирования на языке ассемблера .....	33
3.4. Задачи по программированию на языке ассемблера .....	47
3.5. Обращение к системным средствам из прикладной программы .....	51
4. Работа с файлами, каталогами и дисками .....	56
4.1. Основные характеристики файловой системы MS-DOS .....	56
4.2. Задачи по программированию операций над файлами, каталогами и дисками .....	65
4.3. Системные средства обслуживания дисков и файлов .....	71
4.4. Защита программных продуктов от копирования и несанкционированного использования .....	82
4.5. Задачи по защите программ от копирования и несанкционированного использования .....	87

## Содержание

5. Ввод информации с клавиатуры терминала .....	99
5.1. Системные процедуры обработки прерываний от клавиатуры .....	99
5.2. Системные средства ввода данных с клавиатуры .....	111
5.3. Задачи по программированию ввода с клавиатуры .....	116
6. Вывод текстовой информации на экран терминала .....	123
6.1. Видеосистема компьютеров типа IBM PC .....	123
6.2. Вывод на экран средствами DOS .....	124
6.3. Управление экраном через ANSI-драйвер .....	126
6.4. Логическая организация текстового видеобuffers .....	130
6.5. Вывод на экран средствами BIOS .....	132
6.6. Задачи по программированию вывода на экран .....	135
6.7. Системные средства управления шрифтами .....	142
6.8. Задачи по программной смене шрифтов .....	145
7. Вывод графической информации на экран терминала .....	148
7.1. Графические возможности видеодрайвера BIOS .....	148
7.2. Задачи по программированию графического режима .....	151
7.3. Адаптер EGA и его прямое программное управление .....	159
7.4. Задачи на прямое программирование адаптера EGA в графическом режиме .....	164
8. Управление памятью и процессами .....	169
8.1. Системные средства распределения памяти .....	169
8.2. Организация дочерних процессов .....	180
8.3. Задачи по управлению процессами .....	190
9. Обработка прерываний .....	200
9.1. Контроллер прерываний и его программирование .....	200
9.2. Задачи на программирование контроллера прерываний .....	209
9.3. Взаимодействие прикладных и системных обработчиков прерываний .....	213
9.4. Обработка прерываний от таймера и будильника .....	217
9.5. Системные стеки и обработчики прерываний .....	220
9.6. Обработка прерываний по <Ctrl>/C и <Ctrl>/<Break> .....	223
9.7. Задачи на обработчики прерываний .....	227

10. Резидентные программы.....	238
10.1. Основы организации резидентных программ.....	238
10.2. Связь с резидентной программой.....	241
10.3. Проверка на повторную установку.....	244
10.4. Задачи на резидентные программы.....	247
11. Некоторые проблемы разработки резидентных программ и обработчиков прерываний.....	257
11.1. Обзор проблем.....	257
11.2. Использование средств BIOS в обработчиках аппаратных прерываний.....	261
11.3. Использование средств DOS в обработчиках аппаратных прерываний.....	263
11.4. Асинхронная активизация резидентных программ командами с клавиатуры.....	278
11.5. Работа с файлами в резидентном обработчике аппаратных прерываний.....	282
11.6. Выгрузка из памяти резидентных программ.....	289
11.7. Свинг области текущих данных DOS.....	300
Приложение 1. Справочные данные по функциям DOS.....	305
Приложение 2. Коды ошибок при выполнении функций DOS.....	325
Приложение 3. Справочные данные по функциям BIOS.....	327
Приложение 4. Команды процессора.....	339
Приложение 5. Основные команды отладчика CodeView Microsoft.....	377
Литература .....	379

МОСКОВСКИЙ ИНЖЕНЕРНО-ФИЗИЧЕСКИЙ ИНСТИТУТ  
Факультет повышения квалификации  
специалистов промышленности (ФПКСП)

и  
Международная ассоциация  
продолженного инженерного образования IASSEE

## IBM PC Краткосрочные курсы

8-10 лекций и 8-10 практических занятий на IBM PC

Хотите стать грамотным пользователем IBM PC? Поступайте в группу

Практическая работа на IBM PC

Вы освоите команды MS-DOS, поработаете с текстовыми процессорами, электронными таблицами, базами данных и инструментальными пакетами, научитесь конфигурировать MS-DOS для своих нужд, познакомитесь с организацией и обслуживанием файлов данных.

Звоните по телефону 324-34-45

Хотите глубоко изучить системные возможности MS-DOS? Поступайте в группу

Основы системного программирования

Вы познакомитесь с архитектурой IBM PC и языком ассемблера, научитесь использовать в своих программах весь арсенал системных средств, практически освоите методику создания и отладки программ управления периферийным оборудованием, обработчиков прерываний, резидентных программ, в том числе с использованием недокументированных средств операционной системы MS-DOS.

Звоните по телефону 324-34-45

Курсы разработаны и проводятся автором настоящей книги  
профессором МИФИ К.Г.Финогеновым