П.И.Рудаков, К.Г.Финогенов

Программируем на языке ассемблера IBM PC

2-е издание

П.И.Рудаков, К.Г.Финогенов

Программируем на языке ассемблера IBM PC

Обнинск 1997 год
Издательство "Принтер"

ББК 32.973.1
Р83
УДК 681.3
Рудаков П.И., Финогенов К.Г.
Р83 Программируем на языке ассемблера IBM PC - Изд. 2-е. - Обнинск: Издательство "Принтер", 1997.- 584 с., илл.

Книга является простым и доступным для широкого круга пользователей пособием по программированию на языке ассемблера для персональных компьютеров IBM PC. Книга состоит из трех частей. Первая часть посвящена основам программирования на языке ассемблера, моделям памяти, управлению аппаратными средствами компьютера, алгоритмам преобразования данных, правилам написания резидентных программ и обработчиков прерываний, использованию средств DOS и BIOS. Часть вторая описывает особенности работы и программирования микропроцессора в защищенном режиме, часть третья - методику программирования арифметического сопроцессора.

Для читателей, не являющихся профессионалами-программистами, но имеющих дело с персональными компьютерами, а также студентов вузов.

ББК 32.973. 1

Учебное издание

Петр Иванович Рудаков
Кирилл Григорьевич Финогенов

ПРОГРАММИРУЕМ НА ЯЗЫКЕ АССЕМБЛЕРА IBM PC

Художник А.Косырев

Подписано к печати 10.06.97г. Формат 60x84 /16
Печ.л.36.5 Тираж 1020 экз. Заказ 611
Издательство "Принтер" ИК Н 89 (03)
249020 г.Обнинск, ул.Королева,6
Отпечатано на фабрике офсетной печати

Содержание

Часть 2. Программирование защищенного режима

Часть 3. Программирование арифметического сопроцессора

Приложения

Введение

Книга рассчитана на пользователей персональных компьютеров типа IBM PC, которые хотели бы познакомиться с архитектурными особенностями этих машин, системой команд и режимами работы используемых в них микропроцессоров, организацией ввода-вывода и прерываний, управлением аппаратными средствами компьютера, функциями базовой системы ввода-вывода BIOS и операционной системы MS-DOS. Все эти знания естественным образом приобретаются при изучении языка ассемблера - языка низкого уровня, приближенного к аппаратным средствам компьютера и его "натуральным" возможностям. В книге будут последовательно рассмотрены основные средства языка ассемблера микропроцессоров Intel 8086 - i486 и его применение при программировании задач разного рода: вычислительных, логических, по управлению аппаратурой и др.

Как известно, программы, написанные на языке ассемблера (если, конечно, они написаны грамотно), отличаются высокой эффективностью, т.е. минимальным объемом и максимальным быстродействием. Это обстоятельство обусловило широкое использование языка ассемблера в тех случаях, когда скорость работы программы или расходуемая ею память имеют решающее значение. Некоторые классы программ (например, программы устанавливаемых драйверов устройств, отличающиеся жесткой структурой) требуют для своего составления обязательного использования языка ассемблера. С другой стороны, поскольку современные системы программирования позволяют объединять в одну выполнимую программу фрагменты, написанные на разных языках, широко практикуется составление комбинированных программ, в которых основная часть программы написана на языке высокого уровня, а наиболее критические участки - на языке ассемблера. Может использоваться и обратный метод, когда в программу на языке ассемблера вставляют фрагменты для выполнения относительно сложных логических или математических преобразований, написанные на языке высокого уровня. Такой метод, в частности, применим при разработке устанавливаемых драйверов. Процедуры на языке Си, включаемые в текст драйвера, упрощают программирование и отладку драйвера и ускоряют процесс его разработки.

Однако, кроме потребительских качеств, язык ассемблера имеет еще значительную методическую ценность. Отражая архитектурные особен-

ности и режимы работы используемого в компьютере микропроцессора, язык ассемблера предоставляет уникальную возможность изучения машины на "низком уровне", освоения того, что и как умеет делать аппаратура компьютера и что вносит в его работу операционная система. Знакомство с внутренними возможностями компьютера чрезвычайно полезно, в частности, для программиста, работающего на языках Паскаль или Си, так как 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.

Язык ассемблера, как и любой другой язык программирования, имеет массу встроенных средств, позволяющих в ряде случаев ускорить и облегчить процесс программирования и расширить возможности создаваемых программ. Профессиональная работа на языке ассемблера, естественно, предполагает детальное знакомство со всеми этими средствами. Чем лучше пользователь владеет техникой программирования на языке ассемблера, тем более эффективными будут его программы. Однако не менее важной является и другая сторона вопроса - освоение особенностей применения языка для реализации аппаратных и программных возможностей компьютера. В этом плане вопросы, нашедшие отражение в настоящей книге, можно условно разбить на две группы. В первую группу входят сведения по основам языка и программирования на нем, в частности:

- способы адресации;
- система команд;
- типичные алгоритмы программ на языке ассемблера;
- выполнение арифметических и логических операций;
- преобразование данных;
- организация подпрограмм;
- макросредства ассемблера.

Другую группу составляют различные аспекты реализации в программах на языке ассемблера аппаратных и системных возможностей компьютера:

- программирование ввода-вывода;
- использование прерываний BIOS и функций DOS;
- программирование арифметического сопроцессора;
- работа в защищенном режиме;
- управление обычной и расширенной памятью.

Книга рассчитана на самостоятельную проработку ее читателем на персональном компьютере. Уже в первой статье книги дается простейшая программа на языке ассемблера, на основе которой рассматриваются наиболее общие архитектурные вопросы. В дальнейших статьях при-

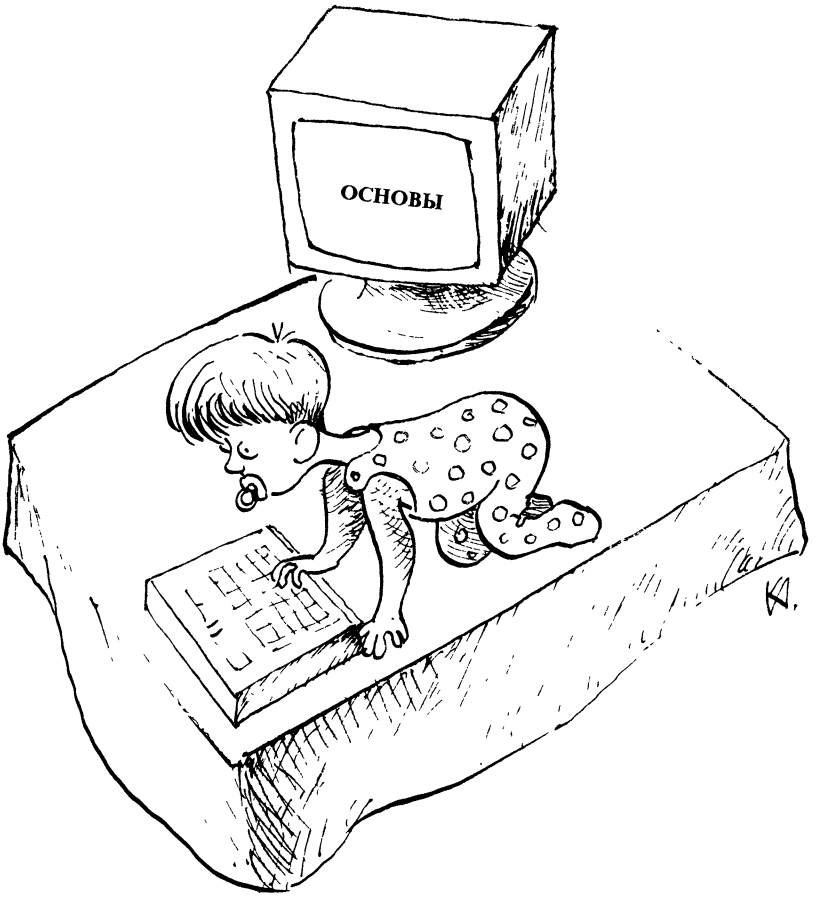
водимые примеры программ постепенно усложняются, обрастают все новыми деталями и дают возможность вводить в изложение новые понятия языка и архитектуры компьютера. Авторы сознательно отказались от традиционного последовательного изложения теоретического материала, заменив его рассмотрением большого количества тщательно подобранных (и, по возможности, простых) программных примеров, которые можно и нужно выполнять на компьютере по мере чтения соответствующих статей. Конечно, такая книга по сравнению с традиционными учебниками выглядит несколько легковесно, однако, как нам кажется, изучение материала на базе относительно простых программных примеров будет стимулировать самостоятельную работу читателей на компьютере и в конечном счете приведет к более глубоким и прочным знаниям.

В первом издании книга, отчасти по полиграфическим причинам, была разделена на четыре части. В настоящем издании две первые части, тесно связанные тематически, объединены в одну, и книга состоит, таким образом, из трех частей. В первой части даются начальные (но далеко не всегда элементарные) сведения по архитектуре процессора и языку ассемблера. Здесь же рассматриваются модели памяти, программирование ввода-вывода, написание обработчиков прерываний, разработка резидентных программ. Вторая часть посвящена защищенному режиму микропроцессора и таким смежным вопросам, как работа с расширенной памятью, многозадачность, защита и др. Наконец, третья часть описывает программирование арифметического сопроцессора.

При подготовке второго издания в книгу были внесены незначительные добавления, устранены повторы и исправлены замеченные опечатки.

Часть 1

Основы программирования



Статья 1

Простейшая программа на языке ассемблера

Начнем изучение языка ассемблера с рассмотрения простой, возможно, даже наипростейшей программы (пример 1.1), которая выводит на экран терминала строку с текстом. Вопросы ввода в компьютер текста программы, ее трансляции и компоновки мы рассмотрим в следующей статье, а пока сосредоточимся на структуре программы.

Пример 1.1. Простейшая программа.

```
text    segment 'code'    ;(1)Начало сегмента команд
        assume CS:text, DS:text;(2)Сегментные регистры CS и DS
                          ;будут указывать на сегмент команд
begin:  mov    AX,text    ;(3)Адрес сегмента команд загрузим
        mov    DS,AX      ;(4)сначала в AX, затем в DS
        mov    AH,09h     ;(5)Функция DOS 9h вывода на экран
        mov    DX,offset message;(6)Адрес выводимого сообщения
        int    21h        ;(7)Вызов DOS
        mov    AH,4Ch     ;(8)Функция 4Ch завершения программы
        mov    AL,00h     ;(9)Код 0 успешного завершения
        int    21h        ;(10)Вызов DOS
message db     'Наука умеет много гитик$';(11)Выводимый текст
text    ends              ;(12)Конец сегмента команд
        end    begin      ;(13)Конец текста программы
                          ;с указанием точки входа
```

Следует заметить, что при вводе исходного текста программы с клавиатуры можно использовать как прописные, так и строчные буквы: транслятор воспринимает, например, строки text segment и TEXT SEGMENT одинаково. Однако с помощью ключа /ML можно заставить транслятор различать прописные и строчные буквы в именах. Тогда строки text segment и TEXT segment уже не будут эквивалентны. Фактически они будут описывать два разных сегмента. Неэквивалентность прописных и строчных букв касается только имен; строки

```
        mov    ds,ax
        MOV    DS,AX
        mov    DS,AX
```

во всех случаях воспринимаются одинаково.

В настоящей книге в программах и их описаниях используются преимущественно строчные буквы. Прописными буквами выделены обозначения регистров и, иногда, имена программных и иных файлов.

Наша программа содержит 13 строк - предложений языка ассемблера. Первое предложение с помощью оператора segment открывает сегмент команд нашей программы. Сегменту дается произвольное имя text. Описатель 'code' (так называемый класс сегмента) говорит о том, что это сегмент команд (слово code обозначает и коды, и команды программы). В конце предложения после точки с запятой располагается комментарий. Таким образом, предложение языка ассемблера может состоять из четырех полей: имени, оператора, операндов и комментария, располагаемых в перечисленном порядке.

Любая программа должна обязательно состоять из сегментов - без сегментов программ не бывает. Обычно в программе задаются три сегмента: команд, данных и стека, но мы в нашей простой программе пока ограничились одним сегментом команд.

Во втором предложении мы с помощью оператора assume сообщаем ассемблеру (программе-транслятору), что сегментные регистры CS и DS будут указывать на один и тот же сегмент text. Сегментные регистры (а всего их в процессоре четыре) играют очень важную роль. Когда программа загружается в память и становится известно, по каким адресам памяти она располагается, в сегментные регистры заносятся начальные адреса закрепленных за ними сегментов. В дальнейшем любые обращения к ячейкам программы осуществляются путем указания сегмента, в котором находится интересующая нас ячейка, а также номера того байта внутри сегмента, к которому мы хотим обратиться. Этот номер носит название относительного адреса, или смещения. Поскольку в единственном сегменте нашей программы будут размещаться и команды, и данные, мы указываем ассемблеру оператором assume (assume - предположим), что и сегментный регистр команд CS, и сегментный регистр данных DS будут указывать на сегмент text. При этом в регистр CS адрес начала сегмента будет загружен автоматически, а регистр DS нам придется загружать (или, как говорят, инициализировать) "вручную".

Строго говоря, в приведенной программе, где нет прямых обращений к ячейкам сегмента данных, не было необходимости сопоставлять в операторе assume сегмент text с сегментным регистром DS (сопоставление сегмента команд с сегментным регистром команд CS обязательно во всех случаях). Учитывая, однако, что практически в любой разумной программе обращения к полям данных имеются, мы с самого начала написали оператор assume в том виде, в каком он используется в реальных программах.

Первые два предложения программы служат для передачи служебной информации программе ассемблера. Ассемблер воспринимает и запоминает эту информацию и пользуется ею в своей дальнейшей работе. Однако в состав выполнимой программы, состоящей из машинных кодов, эти строки не попадут, так как процессору, выполняющему про-

грамму, они не нужны. Другими словами, операторы segment и assume не транслируются в машинные коды, а используются лишь самим ассемблером на этапе трансляции программы. Такие нетранслируемые операторы иногда называют псевдооператорами, или директивами ассемблера в отличие от истинных операторов - команд языка.

Предложение 3, начинающееся с метки begin, является первой выполнимой строкой программы. Для того, чтобы процессор знал, с какой строки начать выполнять программу после ее загрузки в память, начальная метка программы указывается в качестве операнда самого последнего оператора программы end (см. предложение 13). Можно подумать, что указание точки входа в программу излишне: ведь как будто и так ясно, что программу надо начать выполнять с начала, а закончить, дойдя до конца. Однако в действительности для программ, написанных на языке ассемблера, это совсем не так! Текст программы может начинаться с описания вспомогательных подпрограмм или полей данных. В этом случае предложение программы, с которого нужно начать ее выполнение, может располагаться где-то в середине текста программы. И завершается выполнение программы совсем не обязательно в ее последних строках, а там, где стоят предложения вызова специальной программы операционной системы, предназначенной именно для завершения текущей программы и передаче управления системе (см. предложения 8...10). Однако начиная от точки входа программа выполняется строка за строкой точно в том порядке, в каком эти строки написаны программистом.

В предложениях 3 и 4 выполняется инициализация сегментного регистра DS. Сначала значение имени text (т.е. адрес сегмента text) загружается командой mov (от move, переместить) в регистр общего назначения процессора AX, а затем из регистра AX переносится в регистр DS. Такая двухступенчатая операция нужна потому, что процессор в силу некоторых особенностей своей архитектуры не может выполнить команду непосредственной загрузки адреса в сегментный регистр. Приходится пользоваться регистром AX в качестве "перевалочного пункта". Кстати, обратите внимание на то, что операнды в командах языка ассемблера записываются в несколько неестественном для европейца порядке - действие команды осуществляется справа налево.

Предложения 5, 6 и 7 реализуют существо программы - вывод на экран строки текста. Делается это не непосредственно, а путем обращения к служебным программам операционной системы MS-DOS, которую мы для краткости будет в дальнейшем называть просто DOS. Дело в том, что в составе команд процессора и, соответственно, операторов языка ассемблера нет команд вывода данных на экран (как и команд ввода с клавиатуры, записи в файл на диске и т.д.). Вывод даже одного символа на экран в действительности представляет собой довольно

сложную операцию, для выполнения которой требуется длинная последовательность команд процессора. Конечно, эту последовательность команд можно было бы включить в нашу программу, однако гораздо проще обратиться за помощью к операционной системе. В состав DOS входит большое количество программ, осуществляющих стандартные и часто требуемые функции - вывод на экран и ввод с клавиатуры, запись в файл и чтение из файла, чтение или установка текущего времени, выделение или освобождение памяти и многие другие.

Для того, чтобы обратиться к DOS, надо загрузить в регистр общего назначения AH номер требуемой функции, в другие регистры - исходные данные для выполнения этой функции, после чего выполнить команду int 21h, (int - от interrupt, прерывание), которая передаст управление DOS. Вывод на экран строки текста можно осуществить функцией 09h, которая требует, чтобы в регистре DX содержался адрес выводимой строки. В предложении 6 адрес строки message загружается в регистр DX, а в предложении 7 осуществляется вызов DOS.

После того, как DOS выполнит затребованные действия, в данном случае выведет на экран текст "Наука умеет много гитик" (помните одноименный карточный фокус?), выполнение программы продолжится. Вообще-то нам вроде бы ничего больше делать не нужно. Однако на самом деле это не так. После окончания работы нашей программы DOS должна выполнить некоторые служебные действия. Надо освободить занимаемую нашей программой память, чтобы туда можно было загрузить следующую программу. Надо вызвать системную программу, которая выведет на экран запрос DOS и будет ждать следующей команды оператора. Все эти действия выполняет функция DOS с номером 4Ch. Эта функция предполагает, что в регистре AL находится код завершения нашей программы, который она передаст DOS. При желании код завершения только что закончившейся программы можно "выловить" в DOS и проанализировать, но сейчас мы этим заниматься не будем. Если программа завершилась успешно, код завершения должен быть равен 0, поэтому в предложении 9 мы загружаем 0 в регистр AL и вызываем DOS уже знакомой нам командой int 21h.

После последнего выполнимого предложения программы можно описывать используемые в ней данные. У нас в качестве данных выступает строка текста. Текстовые строки вводятся в программу с помощью директивы ассемблера db (от define byte, определить байт), и заключаются в апострофы. Для того, чтобы в программе можно было обращаться к данным, поля данных, как правило, предваряются именами. В нашем случае таким именем является вполне произвольное обозначение message, с которого начинается предложение 11.

Выше, в предложении 6, мы через регистр DX передали DOS адрес начала выводимой на экран строки текста. Но как DOS определит, где

эта строка закончилась? Хотя нам конец строки в программе отчетливо виден, однако в машинных кодах, из которых состоит выполнимая программа, он никак не отмечен, и DOS, выведя на экран слово "гитик", продолжит вывод байтов памяти, расположенных за нашей фразой. Поэтому DOS следует передать информацию о том, где кончается строка текста. Некоторые функции DOS требуют указания в одном из регистров длины выводимой строки, однако функция 09h работает иначе. Она выводит текст до символа $, которым мы и завершили нашу фразу.

Директива ends (end segment, конец сегмента) в предложении 12 указывает ассемблеру, что сегмент text закончился.

Последняя строка программы содержит директиву end, которая говорит программе ассемблера, что закончился вообще весь текст программы, и больше ничего транслировать не нужно. В качестве операнда этой директивы, как уже отмечалось, обычно указывается точка входа в программу, т.е. адрес первой выполнимой программной строки. В нашем случае это метка begin.

Статья 2

Подготовка программы к выполнению

Процесс подготовки и отладки программы на языке ассемблера включает этапы подготовки исходного текста, трансляции, компоновки и отладки.

Подготовка исходного текста программы выполняется с помощью любого текстового редактора. Файл с исходным текстом должен иметь расширение .ASM. 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 о формате (размер страниц, тип шрифта и др.). 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, Norton Editor, редактор EDIT.COM, входящий в состав операционной системы MS-DOS и др. Поскольку при интенсивном программировании часто приходится переносить фрагменты текста из одной про-

граммы в другую, желательно, чтобы редактор имел средство деления экрана на независимые окна. Таким свойством обладают программы Лексикон (10 окон) и Norton Editor (2 окна, что в большинстве случаев вполне достаточно) но не обладает редактор EDIT.COM. При работе в операционной среде какой-либо системы программирования, например, Borland C, можно воспользоваться редактором, встроенным в эту среду.

Трансляция исходного текста программы состоит в преобразовании строк исходного языка в коды машинных команд и выполняется с помощью транслятора с языка ассемблера (т.е. с помощью программы ассемблера). Можно воспользоваться макроассемблером фирмы IBM, программой TASM фирмы Borland или транслятором MASM фирмы Microsoft. Трансляторы различных фирм имеют незначительные различия, в основном, в части описания макросредств. Однако, входной язык (т.е. мнемоника машинных команд и других операторов и правила написания предложений ассемблера) для всех ассемблеров одинаков. После трансляции образуется объектный файл с расширением OBJ.

Компоновка объектного файла выполняется с помощью программы компоновщика (редактора связей). Эта программа получила такое название потому, что ее основное назначение - подсоединение к файлу с основной программой файлов с подпрограммами и настройка связей между ними. Однако компоновать необходимо даже простейшие программы, не содержащие подпрограмм. Дело в том, что у компоновщика есть и вторая функция - изменение формата объектного файла и преобразование его в выполнимый файл, который может быть загружен в оперативную память и выполнен. Файл с программой компоновщика обычно имеет имя LINK.EXE, хотя это может быть и не так. Например, компоновщик фирмы Borland назван TLINK.EXE. Компоновщик желательно брать из одного пакета с ассемблером. В результате компоновки образуется загрузочный, или выполнимый файл с расширением .EXE.

Известно, что программы, выполняемые под управлением MS-DOS, могут принадлежать к одному из двух типов, которым соответствуют расширения имен выполнимых 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. В результате 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 команд и данных. При разработке программных продуктов чаще используется формат .EXE, как обладающий большей универсальностью. Однако некоторые специфические программы (обработчики аппаратных прерываний, резидентные программы) обычно пишутся в формате .COM.

Если исходная программа написана в формате .COM, то после трансляции и компоновки обычным образом ее надо преобразовать в файл типа .COM, для чего используется внешняя команда DOS EXE2BIN. Позже этот вопрос будет рассмотрен подробнее.

Отладка готовой программы может выполняться разными методами, выбор которых определяется структурой и функциями отлаживаемой программы. Свою специфику отладки имеют, например, резидентные программы, обработчики аппаратных прерываний, драйверы устройств и другие классы программ. В целом наиболее удобно отлаживать программы с помощью какого-либо интерактивного отладчика, который позволяет выполнять отлаживаемую 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 контролировать выполнение программы.

При использовании пакета фирмы Borland следует взять "турбо-дебаггер" TD.EXE, при трансляции и компоновке программы с помощью пакета фирмы Microsoft - отладчик Codeview (файл CV.EXE). Можно воспользоваться и отладчиком DEBUG.EXE, входящим в состав операционной системы MS-DOS, хотя с ним не очень удобно работать, так эта программа не обеспечивает привычный для сегодняшнего пользователя полноэкранный интерфейс.

Следует заметить, что хотя турбо-отладчик, входящий в пакет фирмы Borland, имеет весьма богатый набор возможностей, он довольно сложен в освоении. Начинающему программисту удобнее воспользоваться отладчиком Codeview фирмы Microsoft. В настоящей книге будет предполагаться, что читатель выполняет предложенные примеры с помощью пакета фирмы Microsoft (транслятор MASM.EXE, компоновщик LINK.EXE, отладчик CV.EXE).

Если файл с исходным текстом программы назван P.ASM, то строка вызова ассемблера может иметь следующий вид:

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

(Еще раз напоминаем, что как в тексте программы на языке ассемблера, так и при вводе с клавиатуры командных строк можно с равным успехом использовать и прописные, и строчные буквы.)

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 CV.

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 файлов: исходного (P.ASM), объектного (P.OBJ) и листинга (P.LST). Точка с запятой подавляет формирование файла P.CRF с перекрестными ссылками, который нам не нужен.

Строка вызова компоновщика может иметь следующий вид:

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

Ключ /CO передает в загрузочный файл символьную 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). Символ точки с запятой подавляет формирование файла с листингом компоновки (P.MAP) и использование библиотечного файла с объектными модулями подпрограмм.

Как уже отмечалось, компоновщик создает загрузочный, готовый к выполнению, модуль в формате .EXE. Запуск подготовленной программы P.EXE осуществляется командой

```
P.EXE
```

или просто

```
P
```

Если программа не работает должным образом, необходимо прибегнуть к помощи интерактивного отладчика. Отладчик CodeView запускается командой

```
CV P
```

где P - имя файла с отлаживаемой программой. По умолчанию отладчик загружает файл с расширением .EXE. В процессе работы отладчик использует также файл с исходным модулем P.ASM, поэтому перед отладкой не следует переименовывать ни исходный, ни выполнимый файлы. Правила работы с отладчиком CodeView Microsoft и его основные команды будут описаны в одной из следующих статей.

Поскольку по мере изучения этой книги вам придется написать и отладить несколько десятков программ, целесообразно создать командный файл, автоматизирующий выполнение однотипных операций трансляции и компоновки. Текст командного файла (для подготовки программ с помощью транслятора и компоновщика фирмы Microsoft) может быть таким:

```
@echo off
masm/z/zi/n p,p,p;
if errorlevel 1 goto err
link/co p,p;
goto end
:err
echo Ошибка трансляции!
goto fin
:end
echo Конец сеанса
:fin
echo .
```

Если трансляция прошла успешно, на экран выводится сообщение "Конец сеанса" и создаются файлы P.OBJ, P.EXE и P.LST; при наличии ошибок в программе на экран будут выведены строки листинга с ошибками, а за ними сообщение "Ошибка трансляции!". Поскольку в приведенном командном файле имя программы указано в явной форме, текущий вариант программы всегда должен иметь одно и то же имя (в приведенном примере - P.ASM). Конечно, можно составить командный файл, воспринимающий текущее имя программы в качестве параметра при его запуске, однако практика показывает, что такая методика замедляет работу и иногда приводит к драматическим ошибкам. Удобнее отлаживать программу всегда под одним и тем же именем, а после отладки записывать в специально созданный каталог архива под уникальным именем (например, под именем, соответствующим номеру примера в книге: 01-01.asm).

Создайте файл с программой из примера 1.1. Подготовьте программу к выполнению. Не смущайтесь, увидев на экране сообщение

```
LINK : warning L4021: no stack segment
```

Это компоновщик сообщает, что не обнаружил в программе сегмента стека. Его там действительно нет, и это не очень хорошо, однако работать программа будет. Позже мы усовершенствуем программу, дополнив ее стеком.

Изучите листинг трансляции (файл P.LST). На рис. 2.1 приведен несколько сокращенный текст листинга трансляции примера 1.1, из которого были удалены комментарии. Обратите внимание на следующие моменты.

Команды программы имеют различную длину и располагаются в памяти вплотную друг к другу. Так, первая команда mov AX,text, начинающаяся с байта 0000 сегмента, занимает 3 байт. Соответственно, вторая команда начинается с байта 0003. Вторая команда имеет длину 2 байт, поэтому третья команда начинается с байта 0005 и т.д.

```
Относительные адреса команд относительно начала сегмента
Машинные коды команд      Исходный текст программы

0000                      text    segment 'code'
                          assume  CS:text,DS:text
0000  B8 ---- R           begin:  mov  AX,text
0003  8E D8                       mov  DS,AX
0005  B4 09                       mov  AH,09h
0007  BA 0011 R                   mov  DX,offset message
000A  CD 21                       int  21h
000C  B4 4C                       mov  AH,4Ch
000E  B0 00                       mov  AL,00h
0010  CD 21                       int  21h
                Коды символов, образующих наше сообщение

0012  8D A0 E3 AA A0 20 E3  message db 'Наука умеет много гитик$'
      AC A5 A5 E2 20 AC AD
      AE A3 AE 20 A3 A8 E2
      A8 AA 24
002A                      text    ends
                          end     begin
  Размер сегмента в байтах
```

Рис. 2.1. Листинг трансляции примера 1.1.

Предложения программы с операторами segment, assume, end не транслируются в какие-либо машинные коды и не находят отражения в памяти. Они нужны лишь для передачи транслятору служебной информации.

Транслятор не смог полностью определить код команды mov AX,text. В этой команде в регистр AX засылается адрес сегмента text. Однако этот адрес станет известен лишь в процессе загрузки выполнимого файла программы в память. Поэтому в листинге на месте этого адреса стоит прочерк.

Данные, введенные нами в программу, также оттранслировались: вместо символов текста в загрузочный файл попадут коды ASCII этих символов. Подробнее о кодах ASCII будет рассказано в статье 7.

Из листинга трансляции легко определить размер отдельных составляющих программы и всей программы в целом. В нашем случае длина сегмента команд (в который вошли и данные) равна всего 2Ah=42 байт (символ h обозначает, что число записано в шестнадцатеричной системе счисления).

Выполните пробный прогон программы и убедитесь, что она работает, как ожидалось: на экран выводится текст "Наука умеет много гитик". Если этого не произойдет, значит, в программе присутствует какая-то скрытая ошибка, и программу придется отлаживать с помощью

отладчика. Работа с интерактивным отладчиком требует некоторых навыков; статья 4 поможет вам освоить эту интересное и полезное инструментальное средство.

Статья 3

Регистры процессора

В статье 1 при описании приведенной там программы упоминались регистры процессора, в частности, сегментные и общего назначения. Регистры процессора являются важнейшим инструментом программиста (между прочим, не только на языке ассеблера, но и на языках высокого уровня), и необходимо иметь четкое представление о составе регистров процессора, их названиях и применении. В примере 1.1 в явной форме использовались лишь три регистра процессора - AX (и его старшая половина AH), DS и DX; по мере чтения книги вы столкнетесь с примерами использования остальных регистров. Однако для того, чтобы читатель мог получить общее представление об этом важном средстве, ниже дан краткий обзор всей системы регистров процессора. Если в этом обзоре вам встретятся незнакомые понятия или неясные места - не огорчайтесь; в последующих статьях они будут описаны более подробно.

Строго говоря, приведенное ниже описание относится только к процессорам 8086 и 80286, для которых характерны 16-разрядные регистры. В процессорах 80386 и i486 почти все регистры 32-разрядные, что существенно увеличивает возможности этих процессоров и в некоторых случаях упрощает программирование (хотя очень часто наоборот, усложняет). Однако младшие половины регистров этих процессоров совпадают и по названиям, и по назначению с 16-разрядными регистрами процессора 8086. Поэтому программы, написанные для 16-раз-рядного процессора, прекрасно работают и на 32-разрядном, хотя и не используют все его возможности. Поначалу мы ограничимся 16-разрядной архитектурой; особенности современных 32-разрядных процессоров будут описаны позже.

Процессор содержит двенадцать 16-разрядных программно-адресуемых регистров, которые принято объединять в три группы: регистры общего назначения (или регистры данных), регистры-указатели

и сегментные регистры. Кроме того, в состав процессора входят счетчик команд и регистр флагов (рис. 3.1).

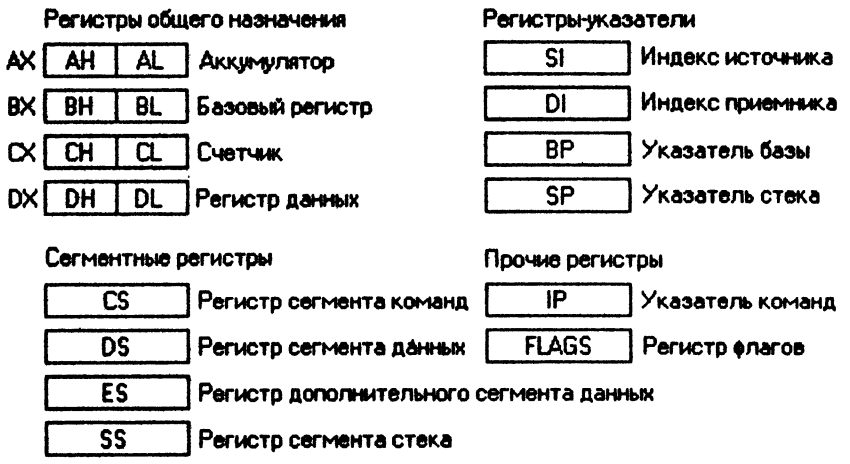


Рис. 3.1. Регистры процессора.

В группу регистров общего назначения 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
```

пересылает старший байт регистра AX в младший байт регистра BX, не затрагивая при этом вторых байтов этих регистров. Еще раз отметим, что сначала указывается операнд-приемник, а после запятой - операнд-источник, т.е. команда выполняется как бы справа налево. Во многих случаях регистры общего назначения вполне эквивалентны, однако предпочтительнее в первую очередь использовать AX, поскольку многие команды занимают в памяти меньше места и выполняются быстрее, если их операндом является регистр AX (или его половины AL или AH).

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 обычно находится в одном из базовых регистров (BX или BP). В дальнейшем такие примеры будут приведены в изобилии.

Регистр BP служит указателем базы при работе с данными в стековых структурах, о чем будет речь впереди, но может использоваться и произвольным образом в большинстве арифметических и логических операций или просто для временного хранения каких-либо данных.

Последний из группы регистров-указателей, указатель стека SP, стоит особняком от других в том отношении, что используется исключительно как указатель вершины стека - специальной структуры, которая будет рассмотрена позже.

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

Четыре сегментных регистра CS, DS, ES и SS хранят начальные адреса сегментов программы и, тем самым, обеспечивают возможность обращаться к этим сегментам.

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 двум сегментам данных, основному и дополнительному), а регистр SS - к сегменту стека. 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.

Регистр флагов, эквивалентный регистру состояния процессора других вычислительных систем, содержит информацию о текущем состоянии процессора (рис. 3.2). Он включает 6 флагов состояния и 3 бита управления состоянием процессора, которые, впрочем, тоже обычно называются флагами.

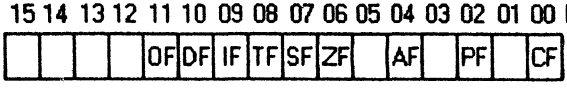


Рис. 3.2. Регистр флагов.

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 младшие 8 бит результата операции содержат 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 нулю.

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

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

Флаги состояния автоматически устанавливаются процессором после выполнения каждой команды. Так, если в регистре AX содержится число 1, то после выполнения команды декремента (уменьшения на 1)

```
dec AX
```

содержимое AX станет равно 0, и процессор сразу отметит этот факт, установив в регистре флагов бит ZF (флаг нуля). Если попытаться сложить два больших числа, например, 58000 и 61000, то установится флаг переноса CF, так как число 119000, получающееся в результате сложения, должно занять больше двоичных разрядов, чем помещается в регистрах или ячейках памяти, и возникает "перенос" старшего бита этого числа в бит CF регистра флагов.

Индицирующие флаги процессора дают возможность проанализировать, если это нужно, результат последней операции и осуществить "разветвление" программы: например, в случае нулевого результата перейти на выполнение одного фрагмента программы, а в случае ненулевого - на выполнение другого фрагмента. Такие разветвления осуществляются с помощью команд условных переходов, которые в процессе своего выполнения анализируют состояние регистра флагов. Так, команда

```
jz zero
```

осуществляет переход на метку zero, если результат выполнения предыдущей команды окажется равен 0 (т.е. флаг ZF установлен), а команда

```
jnc okey
```

выполнит переход на метку okey, если предыдущая команда сбросила флаг переноса CF (или оставила его в сброшенном состоянии).

Управляющий флаг трассировки TF (Trace Flag) используется в отладчиках для осуществления пошагового выполнения программы. Если TF=1, то после выполнения каждой команды процессор реализует процедуру прерывания 1 (через вектор прерывания с номером 1).

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 особой группой команд, предназначенных для обработки строк. Если DF=0, строка обрабатывается в прямом направлении, от меньших адресов к большим; если DF=1, обработка строки идет в обратном направлении.

Таким образом, в отличие от битов состояния, управляющие флаги устанавливает или сбрасывает программист, если он хочет изменить настройку системы (например, запретить на какое-то время аппаратные прерывания или изменить направление обработки строк).

Вернемся к примеру 1.1. Для того, чтобы инициализировать сегментный регистр DS сегментным адресом text нашего (единственного) сегмента, значение text загружается сначала в регистр общего назначения AX, а из него - в сегментный регистр DS. В принципе в качестве "перевалочного пункта" вместо регистра AX можно взять любой другой (например, BX или SI), однако некоторым трансляторам это может не понравиться, так что лучше все-таки использовать AX.

В предложении 5 в регистр AH заносится номер функции DOS, реализующей вывод на экран строки текста. DOS, получив управление с помощью команды int 21h, определяет номер требуемой функции именно по содержимому регистра AH, поэтому никаким другим регистром здесь воспользоваться нельзя. Функция DOS вывода строки извлекает адрес выводимой строки из регистра DX, поэтому в строке 6 использование регистра DX также предопределено. В действительности дело обстоит сложнее. Функция 09h предполагает, что строка с выводимым текстом находится в сегменте, на который указывает вполне определенный сегментный регистр, именно регистр DS. Поэтому перед вызовом функции 09h необходимо настроить этот регистр, что мы и сделали в предыдущих предложениях программы. Сведения о том, какие регистры требуется настроить для выполнения той или иной функции DOS, можно почерпнуть из справочника по функциям DOS, без которого, таким образом, практически невозможно писать программы с обращениями к системным средствам.

В предложениях 8...10 осуществляется вызов системной функции 4Ch, служащей для завершения текущей программы. По-прежнему номер функции заносится в регистр AH; кроме этого, в AL можно поместить код завершения программы, который в нашем примере равен 0.

Статья 4

Интерактивный отладчик CodeView Microsoft

Часто бывает так, что программа, успешно пройдя этапы трансляции и компоновки, все же работает не так, как ожидалось или вообще не работает. Это значит, что формально, с точки зрения правил языка программирования, программа написана правильно (в ней нет синтаксических ошибок), однако алгоритм ее в чем-то неверен. Для отладки такой программы следует воспользоваться услугами интерактивного отладчика. Интерактивным он называется потому, что вся работа с ним осуществляется в непрерывном диалоге с пользователем.

Познакомимся с отладчиком CodeView фирмы Microsoft, воспользовавшись программой из примера 1.1, которую мы еще раз приводим здесь в слегка видоизмененном виде. В программе сокращены комментарии и предусмотрен вывод на экран латинских, а не русских букв (отладчик CV не во всех случаях удовлетворительно работает с русскими буквами).

Пример 4.1. Программа для изучения отладчика CodeView.

```
text     segment 'code'
         assume CS:text,DS:text
myproc   proc
         mov    AX,text     ;Инициализация
         mov    DS,AX       ;регистра DS
         mov    AH,09h      ;Номер функции DOS
         mov    DX,offset mes;Адрес выводимой строки
         int    21h         ;Вызов DOS
outprog: mov    AH,4Ch      ;Вызов функции DOS 4Ch
         mov    AL,0h       ;завершения
         int    21h         ;текущей программы
myproc   endp
mes      db     'Program started$'
text     ends
end      myproc
```

Как уже отмечалось, для полного использования возможностей отладчика следует при трансляции программы указать в числе других ключ /ZI, а при компоновке - ключ /CO:

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

Кроме того, следует убедиться, что в вашем рабочем каталоге имеется и загрузочный (P.EXE), и исходный (P.ASM) файлы, поскольку отладчик в своей работе использует оба этих файла. Вызовем отладчик командой

```
CV P
```

На экране появится информационный кадр отладчика (рис. 4.1).

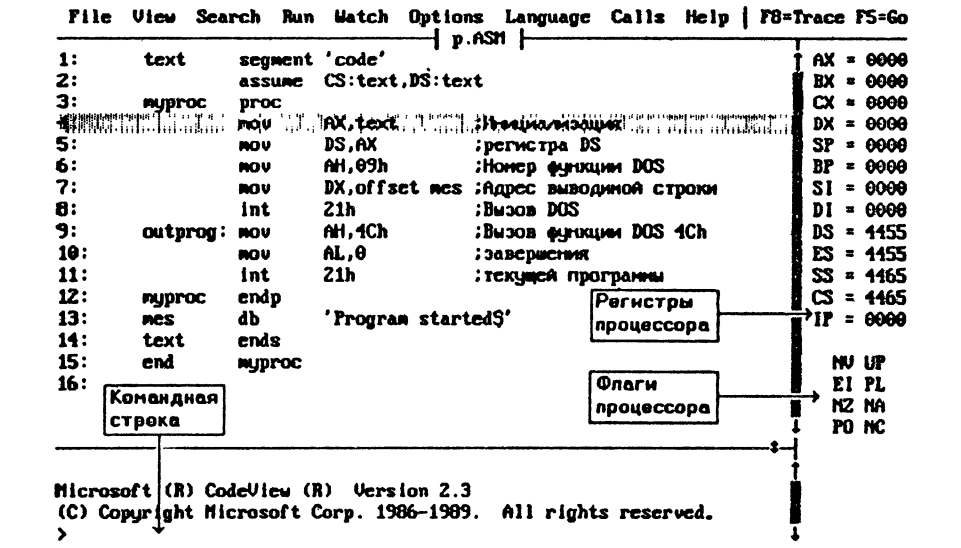


Рис. 4.1. Начальный информационный кадр отладчика.

Как видно из рисунка, информационный кадр отладчика содержит несколько полей. Основное поле в центре кадра занято текстом отлаживаемой программы. Верхняя строка - это главное меню отладчика. Для перехода в меню необходимо нажать клавишу <Alt>, после этого подсвечивается активный пункт меню. Смена активного пункта выполняется с помощью клавиш <стрелка вправо> и <стрелка влево>. Если после выделения требуемого пункта нажать клавишу Enter, то под его именем выведется меню данного пункта. Для выбора необходимого действия надо с помощью клавиш <стрелка вверх> и <стрелка вниз> выделить нужный пункт и нажать клавишу Enter.

Управлять процессом отладки можно тремя способами: с помощью меню, путем использования функциональных и управляющих клавиш, а также с помощью команд отладчика. Часто одно и то же действие можно реализовать несколькими путями. На первых порах проще всего

пользоваться меню. Так, например, для завершения работ ладчиком надо войти в главное меню, выбрать пункт File, крывшемся подменю выбрать пункт Exit (т.е. переместить на светку и нажать клавишу <Enter>). На рис. 4.2. приведен ин онный кадр отладчика с открытым меню File перед последн тием клавиши Enter.

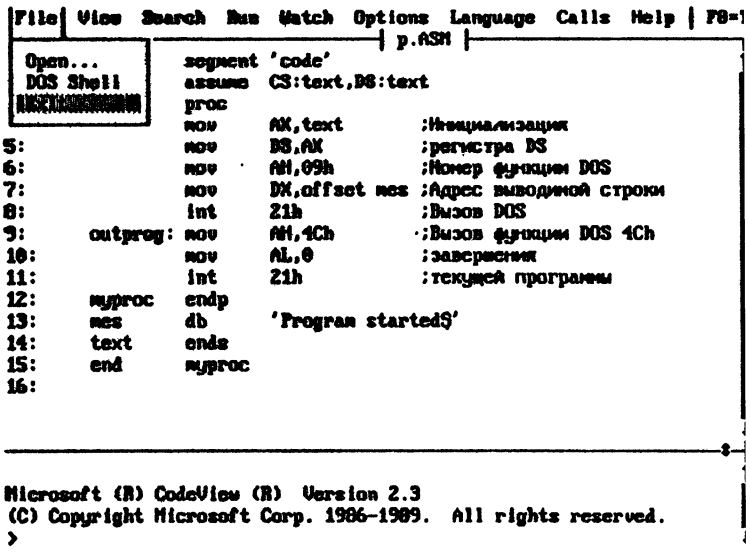


Рис. 4.2. Информационный кадр отладчика с активизированным меню пунк

Второй способ (использование "горячих клавиш") предпо поминание некоторых комбинаций управляющих клавиш. В пункте каждого меню одна буква выделена другим цветом. Ве виатуры этой буквы (при открытом меню) активизирует соот щий пункт меню. Поскольку в пункте File выделена буква F, а Exit - буква е, для завершения работы с отладчиком достаточ комбинацию клавиш <Alt>/f/x (можно нажать и отпустить <A тем нажать сначала "f", а затем "x"; можно, не отпуская <Alt сначала "f", а затем "x").

Вернемся к рис. 4.1. Первое выполнимое предложение п выделено синим цветом. По мере последовательного выполне программы отладчик всегда выделяет то предложение, кото выполнено в следующий момент. Правую сторону информа кадра занимает поле, в которое выводится содержимое регис цессора непосредственно перед выполнением выделенного пре

программы. В нижней части этого поля в двух столбцах индицируются значения флагов процессора. Используемые отладчиком условные обозначения состояния флагов приведены в табл. 4.1.

Таблица 4.1. Перечень и значения флагов, выводимых в отладчике

Название флага	Выводимые значения	
	Флаг установлен	Флаг не установлен
Левый столбец		
Флаг переполнения OF	OV	NV
Флаг прерывания IF	EI	DI
Флаг нуля ZF	ZR	NZ
Флаг паритета PF	PE	PO
Правый столбец		
Флаг направления DF	DN	UP
Флаг знака ZF	NG	PL
Флаг вспомогательного переноса AF	AC	NA
Флаг переноса CF	CY	NC

Наконец, нижнее поле информационного кадра служит для ввода команд оператора (на самой нижней строке) и получения выходных сообщений отладчика, в частности, вывода содержимого заданного участка памяти ("дампа" памяти). В исходном состоянии отладчика нижнее поле является активным: в нем находится мерцающий курсор. Если нужно сделать активным основное поле (с текстом программы), следует нажать клавишу <F6>. Курсор будет находиться под одной из строк программы (под ее номером); перемещая курсор по программе, можно расставлять точки останова и управлять пошаговым выполнением программы.

Рассмотрим типичные действия, к которым приходится прибегать по ходу отладки программы.

После нажатия клавиши <F10> выполняется одно предложение программы. Можно выполнить сразу и целый фрагмент программы (несколько предложений). Для этого надо нажатием клавиши <F6> перевести курсор в основное поле, поместить его перед тем предложением, на котором требуется сделать остановку, и нажать клавишу <F7>. Выполнятся все строки программы до той, на которой установлен курсор; подсветка переместится на эту строку. Далее можно опять выполнять программу построчно, нажимая на клавишу <F10> или, установив в требуемом месте курсор, выполнить следующий фрагмент, нажав <F7>.

Для повторного выполнения программы ее следует "рестартовать". Для этого надо выбрать в главном меню пункт Run, а в меню этого пункта - пункт Restart. То же достигается вводом команды <Alt>/r/r.

По ходу выполнения строк программы в правое поле выводится текущее состояние регистров процессора и его флагов. Контроль содержимого регистров и флагов в процессе пошагового выполнения программы - основной метод ее отладки.

Не выходя из отладчика, можно увидеть результат работы программы, если она выводит что-либо на экран. Для этого надо выбрать пункт меню View, а в нем подпункт Output. То же действие выполняется нажатием клавиши <F4>. Для возврата в окно отладчика надо нажать любую клавишу.

По ходу пошагового выполнения программы можно изменять содержимое регистров. Это дает возможность исправлять обнаруженные ошибки (если выяснилось, что в какой-то строке программы заполняется не тот регистр или не тем содержимым), а также динамически модифицировать программу с целью изучения ее работы.

Выполните программу 4.1 до предложения int 21h и просмотрите содержимое регистров процессора. Вы увидите, что в старшей половине регистра AX находится число 09 - номер вызываемой функции DOS. Младшая половина регистра AX заполнена "мусором" - остатком от выполнения последней операции с регистром AX. В регистре DX будет, скорее всего, число 12h - относительный адрес первого байта строки message в сегменте команд. Изменим этот относительный адрес. Для этого надо ввести команду

```
r dx=n
```

где n - значение (16-ричное), которое требуется записать в регистр. Введем команду

```
R DX=16
```

(или r dx=16). Выполним следующе предложение программы (клавиша <F10>) и посмотрим содержимое экрана DOS (клавиша <F4>). Будет выведена строка

```
ram started
```

Действительно, увеличив содержимое регистра DX на 4, мы поместили в него относительный адрес пятого символа нашей строки, с которого теперь и выводится текст.

Отладчик позволяет также изменять содержимое ячеек памяти. Введем команду рестарта программы (<Alt>/R/R) и выполним ее опять до предложения int 21h. Теперь изменим содержимое ячеек памяти, в которых хранится выводимая на экран строка. Для этого надо воспользоваться командой E, например, в таком варианте:

```
E DX+4 '-' '0' '1'
```

В байты памяти, начиная с адреса, на 4 превышающего содержимое DX, т.е. начиная с пятого байта нашего сообщения, запишутся коды символов "-", "0" и "1". Если теперь выполнить очередную команду (int 21h), на экран будет выведена строка

```
Prog-01 started
```

Еще одной полезной операцией является вывод содержимого области памяти, начиная с заданного адреса ("дамп" памяти). Выведем содержимое поля message. Для этого выполним, как и раньше, программу до предложения int 21h (чтобы в DX попало смещение message) и введем команду

```
DB DX
```

Здесь D - команда выполнить дамп памяти, а буква B указывает, что дамп надо выполнять по байтам (а не по словам). В нижнее поле кадра будет выведено содержимое сегмента данных, начиная с адреса, содержащегося в регистре DX. В левой части поля указываются 16-ричные коды, а в правой - соответствующие им символы. Для того, чтобы увидеть весь вывод отладчика, следует нажатием клавиши <F2> убрать на время изображение регистров и флагов. Вторичным нажатием клавиши <F2> восстанавливается исходный режим. На рис. 4.3 приведен информационный кадр отладчика после выполнения указанной операции.

```
File  View  Search  Run  Watch  Options  Language  Calls  Help | F8=Trace F5=Go
                                  p.ASM
1:      text    segment 'code'
2:              assume  CS:text,DS:text
3:      myproc  proc
4:              mov     AX,text         ;Инициализация
5:              mov     DS,AX           ;регистра DS
6:              mov     AH,09h          ;Номер функции DOS
7:              mov     DX,offset mes ;Адрес выводимой строки
8:              int     21h             ;Вызов DOS
9:      outprog: mov    AH,4Ch          ;Вызов функции DOS 4Ch
10:             mov     AL,0            ;завершения
11:             int     21h             ;текущей программы
12:     myproc  endp
13:     mes     db      'Program startedS'
14:     text    ends
15:     end     myproc
16:

>db dx
4165:0010       50 72 6F 67 72 61-6D 20 73 74 61 72 74 65    Program starte
4165:0020  64 24 4E 42 30 30 7E 00-00 00 00 00 00 00 22 00  d$NB00~.......".
>
```

Рис. 4.3. Информационный кадр отладчика после ввода команды получения дампа памяти.

Естественно, что возможности отладчика не ограничиваются описанными. Перечень наиболее употребительных команд отладчика приведен в Приложении 2 в конце книги; кроме того, в отладчике имеется встроенный справочник (составленный, к сожалению, довольно путанно), с помощью которого можно получить дополнительную информацию о возможностях и командах программы CodeView. Справочник вызывается нажатием комбинации клавиш <Alt>/h.

Статья 5

Сегментная адресация и сегментная структура программ

Почему программа должна обязательно состоять из сегментов? Причина этого кроется в архитектурных особенностях микропроцессоров фирмы Intel, которые нам придется здесь коротко рассмотреть. Важнейшей характеристикой любого микропроцессора (МП) является разрядность его внутренних регистров, а также внешних шин адресов и данных. МП Intel 8086 имеет 16-разрядную внутреннюю архитектуру и такой же разрядности шину данных. 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. Физический 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 иногда называют относительным адресом (рис. 5.1). Сегментный адрес без четырех младших битов, т.е. деленный на 16, хранится в одном из сегментных регистров. При вычислении физического адреса процессор умножает содержимое сегментного регистра на 16 и прибавляет к полученному 20-разрядному адресу относительный адрес. Умножение базового адреса на 16 увеличивает диапазон адресуемых ячеек до величины 64 Кбайт * 16 = 1 Мбайт.

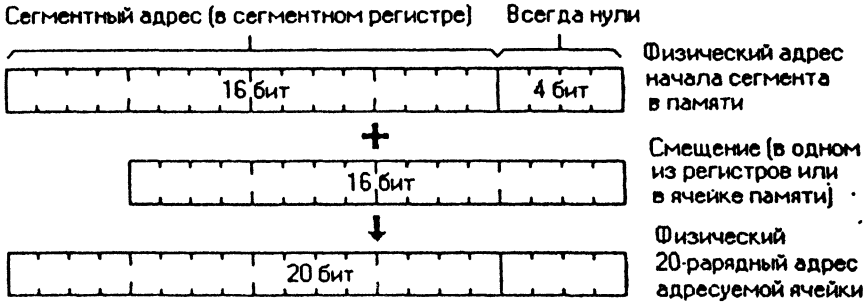


Рис. 5.1. Образование физического адреса из сегментного адреса и смещения.

МП 80286, использовавшийся как центральный процессор компьютеров IBM PC/AT, явился 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 и i486) используются два режима работы: реального адреса (реальный режим) и виртуального защищенного адреса (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 байт, из которых 3 байта (24 двоичных разряда) отводятся под сегментный адрес. Тем самым обеспечивается полное использование 24-разрядного адресного пространства.

МП 80386 и i486 являются высокопроизводительными процессорами с 32-разрядными шинами данных и адресов и 32-разрядной внутренней архитектурой. Они, как и МП 80286, могут работать в реальном и защищенном режимах. В последнем случае микропроцессор позволяет адресовать до 2^{32}=4 Гбайт физической памяти. Программирование защищенного режима будет подробно рассмотрено в следующей части этой книги.

Итак, обращение к любым участкам памяти осуществляется исключительно посредством сегментов - логических образований, накладываемых на требуемые 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 (как правило, программистом с помощью соответствующих программных строк) в один из сегментных регистров. При обращении к памяти процессор извлекает из сегментного регистра этот базовый адрес, умножает его на 16 сдвигом влево на 4 двоичных разряда и складывает с заданным каким-либо образом смещением, получая 20-разрядный физический адрес адресуемой ячейки памяти (слова или байта). Этот процесс проиллюстрирован на рис. 5.2 на конкретном примере команды inc mem1.

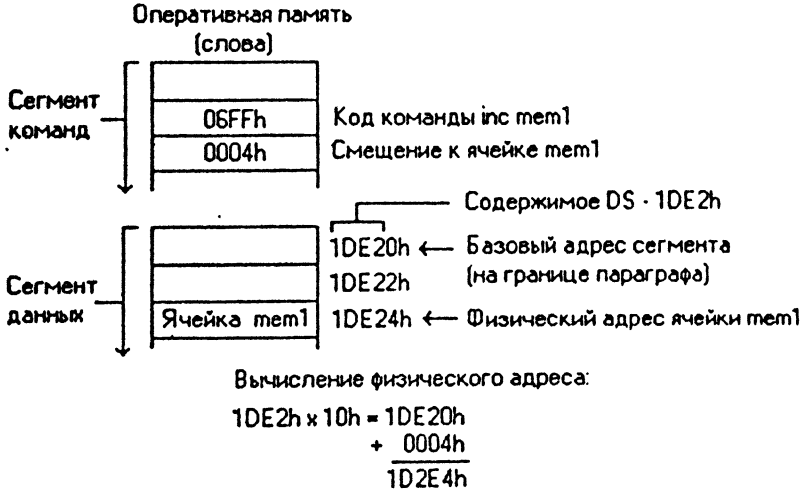


Рис. 5.2. Формирование физического адреса.

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 соответствующего ему физического адреса.

Наличие в микропроцессоре четырех сегментных регистров определяет структуру программы. В типичной, не слишком сложной программе имеются сегмент команд, сегмент данных и сегмент стека, которые

адресуются с помощью сегментных регистров CS, DS и SS соответственно. Дополнительный сегментный регистр ES часто используется для обращения к полям данных, не входящим в программу, например к видеобуферу или системным ячейкам. Однако при необходимости его можно настроить и на один из сегментов программы. В частности, если программа работает с большим объемом данных, для них можно предусмотреть два сегмента и обращаться к одному из них через регистр DS, а к другому - через ES.

Наша первая программа из статьи 1 содержала лишь один сегмент, в котором располагались и команды, и данные. Такая конструкция программы вполне законна, но не очень наглядна. Кроме того, предусмотрев в программе лишь один сегмент, мы ограничили суммарный объем команд и данных величиной 64K. Разумнее разнести команды и данные по отдельным сегментам, что продемонстрировано в примере 5.1.

Пример 5.1 Программа с двумя сегментами.

```
text    segment 'code'    ;(1)Начало сегмента команд
        assume CS:text,DS:data;(2)Регистр CS будет указывать на
                          ;сегмент команд, DS - на сегмент данных
begin:  mov    AX,data    ;(3)Адрес сегмента данных загрузим
        mov    DS,AX      ;(4) сначала в AX, затем в DS
        mov    AH,9       ;(5)Функция DOS вывода на экран
        mov    DX,offset message;(6)Адрес выводимого сообщения
        int    21h        ;(7)Вызов DOS
        mov    AX,4C00h   ;(8)Функция DOS завершения программы
                          ;вместе с кодом завершения 0
        int    21h        ;(9)Вызов DOS
text    ends              ;(10)Конец сегмента команд
data    segment           ;(11)Начало сегмента данных
message db 'Наука умеет много гитик$';(12)Выводимый текст
data    ends              ;(13)Конец сегмента данных
        end    begin      ;(14)Конец текста программы и точка
                          ;входа
```

Приведенная программа отличается от примера 1.1 лишь несколькими деталями (хотя ее структура, можно сказать, отличается радикально). Вслед за сегментом команд введен отдельный сегмент данных с произвольным именем data. Этот сегмент открывается в предложении 11 и закрывается в предложении 13. Изменилось предложение 2 - в нем указано, что сегментный регистр CS будет указывать на сегмент команд text, а регистр DS - на сегмент данных data. Соответственно изменилось и предложение 3. Теперь в регистр DS загружается адрес сегмента данных data, а не сегмента команд text, как это было в примере 1.1.

Оттранслировав, скомпоновав и выполнив программу примера 5.1, можно заметить, что результат работы этой программы в точности тот же, что и для первой, односегментной программы. Действительно, введение отдельного сегмента данных повысило наглядность программы и

дало нам возможность (которой мы пока не воспользовались) увеличить общий размер программы до 128 Кбайт (64 Кбайт команд и 64 Кбайт данных). Однако существо программы осталось без изменения.

Теперь у нас есть две простых программы, одна с одним сегментом, а другая - с двумя. Запустите первую программу (пример 1.1) под управлением отладчика, выясните, чему равен сегментный адрес сегмента команд и определите, в какое место оперативной памяти загружена программа. Проследите за процессом настройки сегментного регистра DS. Обратите внимание на то, что перед началом выполнения программы содержимое регистров DS и ES оказывается в точности на 10h меньше содержимого регистра CS. В дальнейшем этот важный факт получит свое объяснение. Запустите под управлением отладчика вторую программу (пример 5.1), изучите расположение сегментов программы в памяти, сопоставьте полученные результаты с размерами сегментов, полученными из листинга трансляции .

Статья 6

Стек

В предыдущих статьях вскользь упоминался стек. Рассмотрим это понятие более подробно.

Стеком называют область программы для временного хранения произвольных данных. Отличительной особенностью стека является своеобразный порядок выборки содержащихся в нем данных: в любой момент времени в стеке доступен только верхний элемент, т.е. элемент, загруженный в стек последним. Выгрузка из стека верхнего элемента делает доступным следующий элемент.

Элементы стека располагаются в области памяти, отведенной под стек, начиная со дна стека (т.е. с его максимального адреса) по последовательно уменьшающимся адресам. Адрес верхнего, доступного элемента хранится в регистре-указателе стека SP. Как и любая другая область памяти программы, стек должен входить в какой-то сегмент или образовывать отдельный сегмент. В любом случае сегментный адрес этого сегмента помещается в сегментный регистр стека SS. Таким образом, пара регистров SS:SP описывают адрес доступной ячейки стека: в SS хранится сегментный адрес стека, а в SP - относительный адрес до-

ступной (текущей) ячейки (рис. 6.1,а). Обратите внимание на то, что в исходном состоянии указатель стека SP указывает на ячейку, лежащую под дном стека и не входящую в него.

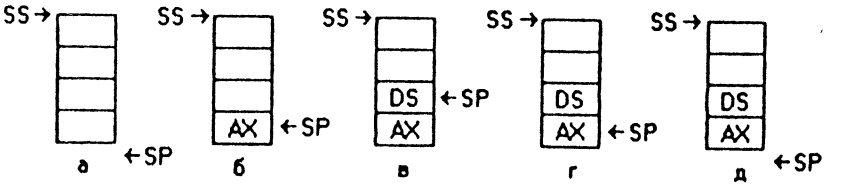


Рис. 6.1. Организация стека: а - исходное состояние, б - после загрузки одного элемента (в данном примере - содержимого регистра AX), в - после загрузки второго элемента (содержимого регистра DS), г - после выгрузки одного элемента, д - после выгрузки двух элементов и возврата в исходное состояние.

Загрузка в стек осуществляется специальной командой работы со стеком push (протолкнуть). Эта команда сначала уменьшает на 2 содержимое указателя стека, а затем помещает операнд по адресу в SP. Если, например, мы хотим временно сохранить в стеке содержимое регистра AX, следует выполнить команду

```
push   AX
```

Стек переходит в состояние, показанное на рис. 6.1,б. Видно, что указатель стека смещается на два байта вверх и по этому адресу записывается указанный в команде проталкивания операнд. Следующая команда загрузки в стек, например,

```
push   DS
```

переведет стек в состояние, показанное на рис. 6.1,в. В стеке будут теперь храниться два элемента, причем доступным будет только верхний, на который указывает указатель стека SP. Если спустя какое-то время нам понадобилось восстановить исходное содержимое сохраненных в стеке регистров, мы должны выполнить команды выгрузки из стека pop (выполкнуть):

```
pop    DS
pop    AX
```

Состояние стека после выполнения первой команды показано на рис. 6.1,г, а после второй - на рис. 6.1,д. Для правильного восстановления содержимого регистров выгрузка из стека должна выполняться в порядке, строго противоположном загрузке - сначала выгружается элемент, загруженный последним, затем предыдущий элемент и т.д.

Обратите внимание на то, что после выгрузки сохраненных в стеке данных они физически не стерлись, а остались в области стека на своих

местах. Правда, при "стандартной" работе со стеком они оказываются недоступными. Действительно, поскольку указатель стека SP указывает под дно стека, стек считается пустым; очередная команда push поместит новое данное на место сохраненного ранее содержимого AX, затерев его. Однако пока стек физически не затерт, сохраненными и уже выбранными из него данными можно пользоваться, если помнить, в каком порядке они расположены в стеке. Этот прием часто используется при работе с подпрограммами и в дальнейшем будет описан подробнее.

В примерах 1.1 и 5.1 мы не заботились о стеке, поскольку, на первый взгляд, нашей программе стек был не нужен. Однако на самом деле это не так. Стек автоматически используется системой в ряде случаев, в частности, при переходе на подпрограммы и при выполнении команд прерывания int. И в том, и в другом случае процессор заносит в стек адрес возврата, чтобы после завершения выполнения подпрограммы или программы обработки прерывания можно было вернуться в ту точку вызывающей программы, откуда произошел переход. Поскольку в нашей программе есть две команды int 21h, операционная система при выполнении программы дважды обращалась к стеку. Где же был стек программы, если мы его явным образом не создали? Чтобы разобраться в этом вопросе, изменим пример 4.1, введя в него строки работы со стеком.

Пример 6.1. Программа, работающая со стеком.

```
text    segment 'code'    ;(1)Начало сегмента команд
        assume CS:text,DS:data;(2)CS->сегмент команд,
                          ;DS->сегмент данных
begin:  mov   AX,data     ;(3)Адрес сегмента данных загрузим
        mov   DS,AX       ;(4)сначала в AX, затем в DS
        push  DS          ;(5)Загрузим в стек содержимое DS
        pop   ES          ;(6)Выгрузим его из стека в ES
        mov   AH,9        ;(7)Функция DOS вывода на экран
        mov   DX,offset message;(8)Адрес выводимого сообщения
        int   21h         ;(9)Вызов DOS
        mov   AX,4C00h    ;(10)Функция завершения программы
                          ;с указанием кода завершения 00h
        int   21h         ;(11)Вызов DOS
text    ends              ;(12)Конец сегмента команд
data    segment           ;(13)Начало сегмента данных
message db 'Наука умеет много гитик$';(14)Выводимый текст
data    ends              ;(15)Конец сегмента данных
        end   begin       ;(16)Конец текста программы с
                          ;указанием точки входа
```

В предложении 5 содержимое DS сохраняется в стеке, а в следующем предложении выгружается из стека в ES. После этой операции оба сегментных регистра, и DS, и ES, будут указывать на один и тот же сегмент данных. В нашей программе эти строки не имеют практического смысла, но вообще здесь продемонстрирован удобный прием переноса

содержимого одного сегментного регистра в другой. Выше уже отмечалось, что в силу особенностей архитектуры микропроцессора для сегментных регистров действуют некоторые ограничения. Так, в сегментный регистр нельзя непосредственно загрузить адрес сегмента; нельзя также перенести число из одного сегментного регистра в другой. При необходимости выполнить последнюю операцию в качестве "перевалочного пункта" часто используют стек.

Запустите под управлением отладчика программу 6.1. Посмотрите, чему равно содержимое регистров SS и SP. Вы увидите, что в SS находится тот же адрес памяти, что и в CS; отсюда можно сделать вывод, что сегменты команд и стека совпадают. Однако содержимое SP равно 0. Первая же команда PUSH уменьшит содержимое SP на 2, т.е. поместит в SP -2. Значит ли это, что стек будет расти, как ему и положено, вверх, но не внутри сегмента команд, а над ним, по адресам -2, -4, -6 и т.д. относительно верхней границы сегмента команд? Оказывается, это не так.

Если взять 16-разрядный двоичный счетчик, в котором записан 0, и послать в него два вычитающих импульса, то после первого импульса в нем окажется число FFFFh, а после второго - FFFEh. При желании мы можем рассматривать число FFFEh, как -2 (что и имеет место при работе со знаковыми числами, о которых будет идти речь позже), однако процессор при вычислении адресов рассматривает содержимое регистров, как целые числа без знака, и число FFFEh оказывается эквивалентным не -2, а 65534. В результате первая же команда занесения данного в стек поместит это данное не над сегментом команд, а в самый его конец, в последнее слово по адресу CS:FFFEh. При дальнейшем использовании стека его указатель будет смещаться в сторону меньших адресов, проходя значения FFFCh, FFFAh и т.д.

Таким образом, если в программе отсутствует явное объявление стека, система сама создает стек по умолчанию в конце сегмента команд.

Рассмотренное явление, когда при уменьшении адреса после адреса 0 у нас получился адрес FFFFh, т.е. от начала сегмента мы прыгнули сразу в его конец, носит название циклического возврата или оборачивания адреса. С этим явлением приходится сталкиваться довольно часто.

Расположение стека в конце сегмента команд не приводит к каким-либо неприятностям, пока размер программы далек от граничной величины 64K. В этом случае начало сегмента команд занимают коды команд, а конец - стек. Если, однако, размер программы приближается к 64K, то для стека остается все меньше места. При интенсивном использовании стека в программе может получиться, что по мере занесения в стек новых данных, стек дорастет до последних команд сегмента команд и начнет затирать эти команды. Очевидно, что этого нельзя допускать.

В то же время система не проверяет, что происходит со стеком и никак не реагирует на затирание команд или данных. Таким образом, оценка размеров собственно программы, данных и стека является важным этапом разработки программы.

Современные программы часто имеют значительный размер (даже не помещаясь в один сегмент команд), а стек иногда используется для хранения больших по объему массивов данных. Поэтому целесообразно ввести в программу отдельный сегмент стека, определив его размер, исходя из требований конкретной программы. Эти и сделано в следующем примере.

Выполняя программу 6.1 по шагам, пронаблюдайте, как команды push и pop изменяют содержимое регистров SP и ES. Выведите на экран дамп памяти начиная с адреса SS:FFF0h. Убедитесь, что содержимое DS действительно записалось в память по адресу SS:FFFEh, и так и осталось там после извлечения содержимого стека и восстановления его указателя.

Что еще имеется в нашей двухсегментной программе, кроме сегментов команд и данных? При загрузке программы в память она будет выглядеть так, как это показано на рис. 6.2.

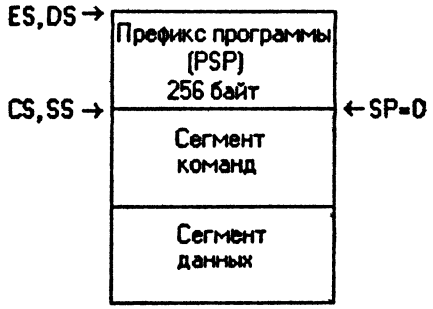


Рис. 6.2. Образ памяти программы .EXE со стеком по умолчанию.

Образ программы в памяти начинается с очень важной структуры данных, которую мы будем называть префиксом программы. В оригинальной литературе эта структура носит не очень удачное название Program Segment Prefics (или сокращенно PSP), т.е. "префикс программного сегмента". PSP образуется и заполняется системой в процессе загрузки программы в память; он всегда имеет размер 256 байт и содержит поля данных, используемые системой (а часто и самой программой) в процессе выполнения программы. К составу полей PSP мы еще не раз будем возвращаться в этой книге.

Вслед за PSP располагаются сегменты программы. Поскольку объявления сегментов сделаны нами наипростейшим образом (операторы segment не сопровождаются операндами-описателями), порядок размещения сегментов в памяти совпадает с порядком их объявления в программе, что упрощает исследование и отладку программы. Для большинства программ не имеет значения, в каком порядке вы будете объявлять сегменты, хотя встречаются программы, для которых порядок

сегментов существен. Для таких программ предложения с операторами segment будут выглядеть сложнее.

В процессе загрузки программы в память 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 команд. SS, как мы экспериментально убедились, также в нашем случае указывает на начало сегмента команд. Как мы увидим позже, верхняя половина PSP занята важной для системы и самой программы информацией, а нижняя половина (128 байт) практически свободна.

Поскольку после загрузки программы в память оба сегментных регистра данных указывают на PSP, сегмент данных программы оказывается не адресуемым. Не забывайте об этом! Если вы позабудете инициализировать регистр DS так, как это сделано в предложениях 3-4 нашей программы, вы не сможете обращаться к своим данным. При этом транслятор не выдаст никаких ошибок, но программа будет выполняться неправильно. Поставьте поучительный эксперимент: уберите из текста программы 6.1 строки инициализации регистра DS (проще всего не стирать эти строки, а поставить в их начале знак комментария - символ ";"). Оттранслируйте, скомпонуйте и выполните такой вариант программы. Ничего ужасного не произойдет, но на экран будет выведена какая-то ерунда. Возможно, в конце этой ерунды будет и строка "Наука умеет много гитик". Почему так получилось? Когда начинает выполняться функция DOS 09h, она предполагает, что полный двухсловный адрес выводимой на экран строки находится в регистрах DS:DX (в DS - сегментный адрес, в DX - относительный). У нас же сегментный регистр DS указывает на PSP. В результате на экран будет выводиться содержимое PSP, который заполнен адресами, кодами команд и другой числовой (а не символьной) информацией.

Рассмотрим теперь программу с тремя сегментами: команд, данных и стека. Такая структура широко используется для относительно несложных программ.

Пример 6.2. Программа с тремя сегментами.

```
text    segment 'code'    ;(1)Начало сегмента команд
        assume CS:text,DS:data;(2)CS->сегмент команд,
                          ;DS->сегмент данных
begin:  mov   AX,data     ;(3)Адрес сегмента данных загрузим
        mov   DS,AX       ;(4)сначала в AX, затем в DS
        push  DS          ;(5)Загрузим в стек содержимое DS
        pop   ES          ;(6)Выгрузим его из стека в ES
        mov   AH,9        ;(7)Функция DOS вывода на экран
        mov   DX,offset message;(8)Адрес выводимого сообщения
        int   21h         ;(9)Вызов DOS
        mov   AX,4C00h    ;(10)Функция завершения программы
        int   21h         ;(11)Вызов DOS
```

```
text    ends                ;(12)Конец сегмента команд
data    segment             ;(13)Начало сегмента данных
message db 'Наука умеет много гитик$';(14)Выводимый текст
data    ends                ;(15)Конец сегмента данных
stk     segment stack 'stack';(16)Начало сегмента стека
        dw    128 dup (0);(17)Под стек отведено 128 слов
stk     ends                ;(18)Конец сегмента стека
        end   begin         ;(19)Конец текста программы
```

В программе 6.2 вслед за сегментом данных объявлен еще один сегмент, которому мы дали имя stk. Так же, как и другие сегменты, сегмент стека можно назвать как угодно. Строка описания сегмента стека (предложение 16) должна содержать так называемый тип объединения, в данном случае описатель stack. Тип объединения указывает компоновщику, каким образом должны объединяться одноименные сегменты разных программных модулей, и используется главным образом в тех случаях, когда отдельные части программы располагаются в разных исходных файлах (например, пишутся несколькими программистами) и объединяются на этапе компоновки. Хотя для одномодульных программ тип объединения обычно не имеет значения, для сегмента стека обязательно указание типа stack, поскольку в этом случае при загрузке программы выполняется автоматическая инициализация регистров SS (сегментным адресом стека) и SP (смещением конца сегмента стека).

В предложении 16, объявляющем сегмент стека, имеется еще один описатель. Слово 'stack', стоящее в апострофах после оператора segment, определяет класс сегмента (в принципе имена классов можно выбирать произвольно). Классы сегментов анализируются компоновщиком и используются им при компоновке загрузочного модуля: сегменты, принадлежащие одному классу, загружаются в память рядом. Для простых программ, входящих в единственный файл с исходным текстом и включающих по одному сегменту команд, данных и стека, указание класса не обязательно, однако для правильной работы компоновщиков и отладчиков желательно, а в некоторых случаях и необходимо указание классов сегментов: 'code' для сегмента команд и 'stack' для сегмента стека. Так, отладчик CodeView не сможет реализовать некоторые из своих режимов вывода на экран текста программы без указания класса 'code', а компоновщик TLINK не будет инициализировать сегмент стека, если не объявлен его класс 'stack'.

В приведенном примере для стека зарезервировано 128 слов памяти, что более чем достаточно для несложной программы.

Заметим, что получившаяся у нас программа является типичной и аналогичная структура будет использоваться в большинстве последующих примеров.

Подготовьте программу 6.2 к выполнению. Запустите ее под управлением отладчика, изучите расположение сегментов программы в

памяти, обратив особое внимание на содержимое регистров SS и SP. Убедитесь, что в программе образовался отдельный сегмент стека размером 100h байт (128 слов = 256 байт). Поинтересуйтесь, где сохраняется значение DS при выполнении предложения 5.

Статья 7

Вывод на экран символьной информации

Представим себе, что мы создаем обучающую программу и нам понадобилось вывести на экран несколько строк из введения к этой книге. Соответствующая программа приведена в примере 7.1.

Пример 7.1. Вывод на экран информационного сообщения.

```
text    segment             ;Начало сегмента команд
        assume CS:text,DS:data
begin:  mov    AX,data      ;Инициализация сегментного
        mov    DS,AX        ;регистра DS
        mov    AH,9         ;Функция DOS вывода на экран
        mov    DX,offset message;Адрес выводимого сообщения
        int    21h          ;Вызов DOS
        mov    AX,4C00h     ;Завершение программы
        int    21h
text    ends                ;Конец сегмента команд
data    segment             ;Начало сегмента данных
message db'Другую группу составляют различные аспекты реализации'
        db 'в программах на языке ассемблера аппаратных и '
        db 'системных возможностей персонального компьютера:'
        db '- программирование ввода-вывода;'
        db '- использование прерываний BIOS и функций DOS;'
        db '- программирование арифметического сопроцессора;'
        db '- работа в защищенном режиме;'
        db '- управление обычной и расширенной памятью.$'
data    ends                ;Конец сегмента данных
stk     segment stack 'stack';Начало сегмента стека
        dw     128 dup(0) ;Стек
stk     ends                ;Конец сегмента стека
        end    begin        ;Конец текста программы
```

Заметим, что эта программа написана стандартным образом: в ней имеются сегменты команд, данных и стека. Так же будут составляться и программы последующих примеров этой книги.

Текст, предназначенный для вывода на экран, включен в программу в помощью директивы db (определить байт). Поскольку длина текста

превышает ширину экрана, он разбит на произвольные по длине участки, каждый из которых формально составляет отдельное предложение языка ассемблера. В конце текста включен знак "$", характеризующий конец строки для функции DOS 09h.

Подготовив и выполнив эту программу, мы увидим, что весь текст вывелся на экран сплошной последовательностью символов без разбивки на строки и абзацы. Неприятно также и то, что наш текст вывелся на "грязный" экран с остатками предыдущих выводов. Для получения более удобочитаемого вывода в символьную строку следует включить коды управления курсором:

09 - табуляция;

10 - перевод строки;

13 - "возврат каретки", т.е. возврат курсора в начало строки экрана.

С очисткой экрана дело обстоит сложнее. Единственное, что пока можно сделать - это вывести на экран столько строк пробелов, чтобы, перемещаясь по мере вывода вверх по экрану, они очистили всю его верхнюю часть.

В примере 7.2 представлено описание модифицированной строки message (все остальные предложения программы не изменяются).

Пример 7.2. Вывод на экран форматированного текста.

```
message db 80*18 dup (' ')
db 9,'Другую группу составляют различные аспекты реализации в'
db ' программах на',10,13,' языке ассемблера аппаратных и'
db ' системных возможностей персонального компьютера:',10,13
db 9,'- программирование ввода-вывода;',10,13
db 9,'- использование прерываний BIOS и функций DOS;',10,13
db 9,'- программирование арифметического сопроцессора;',10,13
db 9,'- работа в защищенном режиме;',10,13
db 9,'- управление обычной и расширенной памятью.$'
```

Пробелы (18 строк по 80 пробелов в строке) описаны с помощью оператора dup (duplicate, дублировать). Перед словом dup указывается коэффициент повторения (в котором можно использовать арифметические выражения), а после оператора dup в скобках - повторяемая строка (состоящая не обязательно из одного символа). Коды табуляции создадут отступы красных строк. Для того, чтобы текст выглядел на экране аккуратно, мы в середину первого абзаца включили коды 10 и 13 перехода на следующую строку.

Кроме обычных алфавитно-цифровых символов и других знаков, имеющихся на клавиатуре компьютера (например, знаков < > [ ] { } | и др.) на экран можно выводить символы псевдографики. Таких символов всего 48; им соответствуют коды от 176 до 223. Так, например, для формирования на экране двойной рамки используются следующие коды:

Наконец, если в строку, выводимую на экран, включить код 7, прозвучит короткий звуковой сигнал.

В примере 7.3 приведена символьная строка, вывод которой с помощью функции 09h DOS приведет к выводу длительного звукового сигнала и изображению приблизительно в центре чистого экрана слова "Внимание!" в рамке.

Пример 7.3. Вывод на экран информационного кадра.

```
message db      25 dup (7)
        db      80*12 dup (' ')
        db      35 dup (' '),201,9 dup (205),187,10,13
        db      35 dup (' '),186,'Внимание!',186,10,13
        db      35 dup (' '),200,9 dup (205),188,10,13
        db      10*80 dup (' '),'$'
```

В приведенных выше примерах некоторые символы вводились в программу в виде их изображений (букв), а другие - с помощью их кодов. В действительности все символы, отображаемые на экране терминала или выводимые на печать, имеют закрепленные за ними коды, которые называются кодами ASCII. Каждый код ASCII занимает один байт и может принимать значение от 0 до 255. Совокупность всех 255 кодов вместе с изображениями символов составляют кодовую таблицу ASCII (рис.7.1).

Очевидно, что описывать в программе текстовые строки с помощью кодов ASCII по меньшей мере неудобно. В то же время для вывода на экран псевдографических изображений или других пиктограмм, для которых нет соответствующих клавиш, приходится пользоваться кодами ASCII. Однако иметь в виду соответствие символов и кодов ASCII приходится довольно часто. Например, упорядочивание символов или символьных строк по алфавиту фактически выполняется по кодам ASCII.

Для усвоения вышеизложенного выведите на экран строку, описанную в примере 7.4. В ней все символы, и алфавитные, и псевдографические, и записаны в виде кодов ASCII. В текст строки вкрались ошибки. Найдите их.

Пример 7.4. Вывод на экран текста, описанного кодами ASCII.

```
message db 25 dup (7), 80*12 dup (32)
db 35 dup (32),201,9 dup (205),187,10,13
db 35 dup (32),186,130,173,168,172,160,173,236,63,186,10,13
db 35 dup (32),200,9 dup (205),188,10,13, 10*80 dup (32),'$'
```

	00	01	02	03	04	05	06	07	08	09
000		☺	☻	♥	♦	♣	♠	•	◘	○
010	◙	♂	♀	♪	♫	☼	►	◄	↕	‼
020	¶	§	▬	↨	↑	↓	→	←	∟	↔
030	▲	▼		!	"	#	$	%	&	'
040	(	)	*	+	,	-	.	/	0	1
050	2	3	4	5	6	7	8	9	:	;
060	<	=	>	?	@	A	B	C	D	E
070	F	G	H	I	J	K	L	M	N	O
080	P	Q	R	S	T	U	V	W	X	Y
090	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	⌂	А	Б

	00	01	02	03	04	05	06	07	08	09
130	В	Г	Д	Е	Ж	З	И	Й	К	Л
140	М	Н	О	П	Р	С	Т	У	Ф	Х
150	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
160	а	б	в	г	д	е	ж	з	и	й
170	к	л	м	н	о	п	░	▒	▓	│
180	┤	╡	╢	╖	╕	╣	║	╗	╝	╜
190	╛	┐	└	┴	┬	├	─	┼	╞	╟
200	╚	╔	╩	╦	╠	═	╬	╧	╨	╤
210	╥	╙	╘	╒	╓	╫	╪	┘	┌	█
220	▄	▌	▐	▀	р	с	т	у	ф	х
230	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
240	Ё	ё	»	«	„	"	+	≈	•	·
250	·	√	№	¤	■					

а

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀	♪	♫	☼
10	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂
80	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
90	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
A0	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
B0	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
C0	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
D0	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
E0	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
F0	Ё	ё	»	«	„	"	+	≈	•	·	·	√	№	¤	■	

б

Рис. 7.1. Кодовая таблица для нашей страны: а - для кодов ASCII в десятичном представлении, б - в шестнадцатеричном.

Статья 8

Esc-последовательности

Из рассмотрения примеров статьи 6 видно, что возможности DOS по управлению экраном весьма скудны. С помощью функции 09h можно выводить на экран символьные строки, включая в них управляющие символы и символы псевдографики. Позиционирование курсора, как и очистку экрана приходится выполнять с помощью пробелов. В DOS нет никаких функций, с помощью которых можно было бы программно установить курсор в заданное место экрана; нельзя также изменить цвет выводимых символов (это-то при цветном мониторе!).

Включение в систему устанавливаемого 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, или 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.

Esc-последовательности можно использовать в прикладных программах для формирования на экране требуемого изображения. В этом случае они включаются в строки, выводимые на экран операторами того языка, на котором написана программа. Это дает возможность выводить строки текста в заданные места экрана, изменять их цвет, заставлять мерцать или выделяться яркостью и т.д.

Для управления экраном используются такие Esc-последовательности:

Esc[2J - очистка экрана и перемещение курсора в левый верхний угол;

Esc[K - очистка строки от курсора до конца строки;

Esc[*строка*;*позиция*H - установка позиции курсора. Параметр *строка* обозначает Y-координату курсора в пределах 1...25, параметр *позиция* - X - координату в пределах 1...80 (для видеорежима 80x25 символов);

Esc[*код_1*;*код_2*;*код_3*m - выбор атрибутов символов. Параметры *код_1, код_2* и *код_3* могут принимать значения:

0 - нормальное изображение (белые символы на черном поле);

1 - выделение яркостью;
5 - выделение мерцанием;
7 - инверсное изображение (черные символы на белом поле);

30 - черные символы;	40 - черный фон;
31 - красные символы;	41 - красный фон;
32 - зеленые символы;	42 - зеленый фон;
33 - коричневые символы;	43 - коричневый фон;
34 - синие символы;	44 - синий фон;
35 - фиолетовые символы;	45 - фиолетовый фон;
36 - бирюзовые символы;	46 - бирюзовый фон;
37 - белые символы;	47 - белый фон.

Esc[*число*A - перемещение курсора на *число* строк вверх;
Esc[*число*B - перемещение курсора на *число* строк вниз;
Esc[*число*C - перемещение курсора на *число* позиций вправо;
Esc[*число*D - перемещение курсора на *число* позиций влево;
Esc[s - сохранение текущих координат курсора в специальном буфере;

Esc[u - восстановление сохраненных в буфере координат курсора (используется вместе с предыдущей последовательностью для того, чтобы вывести что-то на экран в другом месте, а потом вернуть курсор в прежнюю позицию);

Помимо перечисленных, имеются Esc-последовательности, служащие для выбора видеорежима, переопределения клавиш клавиатуры и другие.

Модифицируем текст выводимой строки программы 7.3, включив в него Esc-последовательности управления экраном.

Пример 8.1. Управление экраном с помощью Esc-последовательностей.

```
message db 10 dup (7)      ;Звуковой сигнал
db 27,'[2J',27,'[31;5m'    ;Очистка экрана и задание цвета
db 27,'[12;35H',201,9 dup (205),187;Позиционирование и символы
db 27,'[13;35H',186,'Внимание!',186;Позиционирование и символы
db 27,'[14;35h',200,9 dup (205),188; Позиционирование и символы
db 27,'[0m,27,'[25;1H$'    ;Отмена цвета и позиционирование
```

Статья 9

Циклы

Циклы (т.е. выполнение некоторого участка программы заданное число раз) относятся к числу важнейших элементов программ на любых языках программирования. Для демонстрации техники организации циклов рассмотрим фрагмент программы, в котором создается и выводится на экран тестовый символьный массив, заполненный кодами алфавитно-цифровых и псевдографических символов. Эти символы имеют коды от 32 (пробел) до 254 (сплошной квадратик). Такой массив можно создать в полях данных программы вручную с помощью оператора db:

```
symbols db      32,33,34,35,36,37,38,39,...
```

однако проще заполнить его данными программным образом (пример 9.1). Для удобства читателей программа примера 9.1 приведена полностью, однако в дальнейшем в примерах будут приводиться только содержательные фрагменты программ. Весь "антураж" (объявления сегментов, инициализация сегментного регистра DS, завершение программы), который всегда остается практически неизменным, будет опускаться.

Пример 9.1. Циклы.

```
text    segment             ;(1)Начало сегмента команд
        assume CS:text,DS:data;(2)
begin:  mov    AX,data      ;(3)Инициализация сегментного
        mov    DS,AX        ;(4)регистра DS
;Подготовим все необходимое для организации цикла
        mov    CX,223       ;(5)Число шагов в цикле
        mov    SI,0         ;(6)Индекс адресуемого элемента
                            ;в заполняемом массиве
        mov    AL,32        ;(7)Код первого символа - пробела
;Теперь собственно цикл, в который входит 4 команды
fill:   mov    symbols[SI],AL;(8)Занесение очередного кода
                            ;в байт массива с индексом SI
        inc    AL           ;(9)Создадим код следующего символа
        inc    SI           ;(10)Сдвинемся в массиве на 1 байт
        loop   fill         ;(11)Команда цикла из CX шагов
;Выведем для контроля полученный символьный массив на экран
        mov    AH,40h       ;(12)Функция DOS вывода
        mov    BX,1         ;(13)Стандартный дескриптор экрана
        mov    CX,223       ;(14)Число выводимых байтов
        mov    DX,offset symbols;(15)Адрес выводимого сообщения
        int    21h          ;(16)Вызов DOS
```

```
;Завершим программу
        mov     AX,4C00h    ;(17)
        int     21h         ;(18)
text    ends                ;(19)Конец сегмента команд
data    segment             ;(20)Начало сегмента данных
;Поля данных программы
symbols db 223 dup ('*') ;(21)Заполняемый массив
data    ends                ;(22)Конец сегмента данных
stk     segment stack 'stack';(23)Начало сегмента стека
        dw    128 dup (0);(24)Стек
stk     ends                ;(25)Конец сегмента стека
        end   begin         ;(26)Конец текста программы
```

Рассмотрим содержательную часть программы. Счетчиком шагов цикла служит регистр CX. Поэтому в него надо занести требуемое число шагов цикла (предложение 5), равное длине заполняемого массива. Работа с массивом осуществляется, как правило, с помощью одного из индексных регистров (SI или DI), в которых хранится и наращивается индекс адресуемого элемента массива, т.е. номер байта массива, к которому осуществляется обращение в данном шаге цикла. Поскольку мы начинаем обрабатывать массив с самого начала, в предложении 6 в регистр SI заносится 0. Регистр AL выбран нами для хранения текущего кода символа, отправляемого в массив. С таким же успехом эту роль мог бы выполнить любой другой байтовый регистр - AH, BL, BH, DL или DH (регистры CL и CH, входящие в состав регистра CX, уже заняты). В предложении 7 в регистр AL заносится код первого символа - пробела. Подготовив все необходимые регистры, можно составить само тело цикла. В предложении 8 код из AL отправляется в элемент (байт) массива symbols, номер которого определяется содержимым индексного регистра SI. Это так называемая индексная адресация, у которой существует несколько разновидностей. В частности, в качестве индексного регистра с тем же успехом можно было использовать BX или DI. В первом шаге цикла заполнится элемент массива с индексом 0. В следующих двух предложениях выполняется инкремент (увеличение на 1) кода очередного символа и индекса в массиве. Наконец, команда loop (петля) (предложение 11) возвращает управление на метку fill, причем делает это ровно 223 раза, в соответствии с исходным содержимым CX.

Чтобы увидеть результаты работы программы, выведем полученный массив на экран. Для этого нельзя воспользоваться уже знакомой нам функцией 09h, потому что где-то в массиве будет содержаться код знака $, а он, как мы уже знаем, не может быть выведен на экран функцией 09h. Поэтому для вывода мы воспользуемся другой функцией DOS, с номером 40h, которая позволяет выводить информацию в файлы и на устройства, в частности, на экран. Приемник выводимой информации определяется числом, заносимым в регистр BX, и называемым дескриптором (или файловым индексом). На этапе инициализации DOS экрану

присваивается дескриптор 1, который носит специальное название "дескриптор стандартного вывода". Работа функции 40h с другими дескрипторами будет рассмотрена в дальнейшем. Кроме дескриптора, функция 40h требует, чтобы в регистре CX находилось число выводимых байтов, а в регистре DX (точнее, в паре регистров DS:DX) - адрес выводимой информации (см. строки 14 и 15). Наконец, команда int 21h передает управление DOS, которая и выполняет требуемую операцию.

В полях данных программы объявлен массив байтов (оператор db), который инициализирован кодами символа "*". Это очень удобный и распространенный прием, который облегчает отладку программы. Если в силу каких-то ошибок в программе массив будет заполнен не весь или не заполнен вообще, на экран будут выведены звездочки. Если же программа работает правильно, исходные звездочки затрутся заполняющими символами.

В заключение отметим еще одну особенность программы 9.1. Это первая программа, в которой выполняется прямое обращение к ячейкам сегмента данных (предложение 8, где заполняется массив с именем symbols). В предыдущих статьях подчеркивалось, что адрес любой ячейки памяти обязательно имеет два компонента: сегментный адрес, хранящийся в одном из сегментных регистров, и относительный адрес, или смещение, которое, частности, может указываться в команде в виде мнемонического обозначения ячейки. В предложении 8 имеется ссылка на ячейку symbols, т.е. указывается смещение. Каким сегментным регистром будет пользоваться процессор при выполнении этой команды? Если в команде не указан в явной форме сегментный регистр, по умолчанию используется DS. Собственно, именно в этом предположении мы в начале программы инициализировали регистр DS адресом сегмента данных, в котором расположен массив symbols. Однако для того, чтобы указанное умолчание действовало, необходимо с помощью оператора assume сопоставить DS именно с этим сегментом. Таким образом, определение DS:data в операторе assume стало необходимым только в этой программе, во всех предыдущих можно было обойтись без него.

Рассмотрим теперь вложенные циклы на примере организации программной задержки. Программные задержки широко используются в тех случаях, когда в какой-то точке программы надо приостановить ее выполнение на некоторое время. Такая необходимость часто возникает при программировании относительно медленной аппаратуры компьютера. Если в аппаратуру посылается последовательно несколько управляющих команд, то для того, чтобы дать аппаратуре время их выполнить, между ними включаются программные задержки (на время несколько единиц или десятков микросекунд). Программные задержки значительно большей величины, порядка нескольких секунд, удобно использовать при отладке программ, выводящих на экран некоторую

(возможно, отладочную) информацию. Задержка дает возможность программисту проследить результаты выполнения каждого шага программы. В примере 9.2 приведен фрагмент именно такой программы. В дальнейшем часто будут даваться не полные тексты программ, а лишь рассматриваемые фрагменты.

Пример 9.2. Программная задержка.

```
;Организуем демонстрационный цикл из 10 шагов, которые будут
;выполняться с задержкой порядка нескольких секунд
        mov   CX,10       ;(1)Число шагов в демо-цикле
cycle:  push  CX          ;(2)Сохраним этот счетчик в стеке
;Выведем на экран контрольную строку из трех символов
        mov   AH,09h      ;(3)
        mov   DX,offset string;(4)
        int   21h         ;(5)
;Организуем программную задержку
        mov   CX,100      ;(6)Счетчик внешнего цикла
outer:  push  CX          ;(7)Сохраним его в стеке
        mov   CX,65535    ;(8)Счетчик внутреннего цикла
inner:  loop  inner       ;(9)Повторм команду loop 65535 раз
        pop   CX          ;(10)Восстановим внешний счетчик
        loop  outer       ;(11)Повторим все это 100 раз
        pop   CX          ;(12)Восстановим счетчик демо-цикла
        loop  cycle       ;(13)Повторим демо-цикл CX=10 раз
;Поля данных (в сегменте данных)
string  db    '<> $'      ;(14)
```

В примере 9.2 с помощью функции DOS 09h (предложения 3..5) на экран выводится строка "<> " с периодом, определяемым программной задержкой. Задержка создается с помощью двух вложенных циклов. Внутренний представляет собой просто команду loop, повторяемую 65535 раз (предложение 9); внешний цикл служит для повторения внутреннего 100 раз. В результате команда loop выполняется 6553500 раз, что, в зависимости от скорости конкретного компьютера, дает задержку приблизительно от одной до нескольких десятков секунд.

Поскольку команда loop выполняется всегда CX раз, этот регистр приходится использовать в каждом цикле заново. Перед входом во внутренний, вложенный цикл текущее значение счетчика внешнего цикла (содержимое CX) сохраняется в стеке командой push, а перед командой loop внешнего цикла восстанавливается командой pop (пары строк 2,12 и 7,10). Разумеется, сохранить значение CX можно где угодно (в любом другом регистре или в ячейке памяти), однако команды сохранения в стеке и восстановления из стека эффективнее других в смысле времени выполнения и расходуемой памяти.

Подготовив к выполнению пример 9.2, поэкспериментируйте с длительностью задержки, изменяя число шагов внешнего цикла (предложение 6).

Статья 10

Ввод с клавиатуры символьной информации

Рассмотрим возможности ввода в программу данных с клавиатуры.

При нажатии на клавишу код ASCII нажатого символа (вместе с его скен-кодом, отражающим номер нажатой клавиши) поступает в буфер ввода с клавиатуры, находящийся в системной области оперативной памяти и доступный функциям DOS. Заполнение буфера клавиатуры происходит по мере нажатия клавиш и никак не связано с выполняемой программой. Если программе требуется ввести с клавиатуры определенную информацию, она ставит запрос к DOS на ввод с клавиатуры одного символа или целой строки. Запрошенная функция DOS обращается к буферу ввода (не непосредственно, а посредством системного драйвера клавиатуры CON и прерывания BIOS, но для нас это не очень существенно) и при наличии в нем символов передает первый из поступивших символов в программу. При этом символ изымается из буфера, освобождая там место для последующих символов.

Если к моменту вызова функции DOS буфер ввода оказывается пуст, DOS начинает непрерывно опрашивать его состояние, ожидая появления в буфере очередного кода, в результате чего программа останавливается до нажатия клавиши.

Таким образом, в программу при вводе с клавиатуры всегда поступают коды ASCII, закрепленные за нажимаемыми клавишами, т.е. символьные строки. Если в программу требуется ввести число, то поступающую символьную информацию следует преобразовать в числовую. Эта процедура будет рассмотрена в последующих статьях.

Рассмотрим любопытный пример использования в программе символьных данных, вводимых с клавиатуры. Вспомним, что цвет символов на экране можно задавать с помощью Esc-последовательностей, причем эти последовательности представляют собой символьные строки (за исключением кода Esc). Таким образом, вводя с клавиатуры параметры Esc-последовательностей и посылая их затем на экран с помощью подходящей функции DOS, можно с клавиатуры задавать цвет изображения на экране. Модифицируем программу примера 8.1, чтобы ею можно было управлять с клавиатуры терминала.

```
;Пример 10.1. Ввод с клавиатуры символьной информации.
;Очистим экран и выведем рамку с текстом без цвета
        mov     AH,9            ;(1)Функция DOS вывода на экран
        mov     DX,offset clear;(2)Адрес выводимого сообщения
        int     21h             ;(3)Вызов DOS
;Введем с клавиатуры параметр Esc-последовательности для задания
;цвета рамки
again:  mov     AH,01h          ;(4)Функция ввода с клавиатуры
        int     21h             ;(5)Вызов DOS
        mov     color,AL        ;(6)Первая цифра цвета
        mov     AH,01h          ;(7)Функция ввода с клавиатуры
        int     21h             ;(8)Вызов DOS
        mov     color+1,AL      ;(9)Вторая цифра цвета
;Выведем на экран цветную рамку
        mov     AH,9            ;(10)Функция DOS вывода на экран
        mov     DX,offset message;(11)Адрес выводимого сообщения
        int     21h             ;(12)Вызов DOS
        jmp     again           ;(13)Бесконечный цикл повторения
        mov     AX,4C00h        ;(14)Завершение программы
        int     21h             ;(15)Вызов DOS
;Поля данных
clear   db      27,'[2J'        ;(16)
message db      27,'['          ;(17)
color   db      '00'            ;(18)
        db      'm',27,'[12;35H',201,9 dup (205),187 ;(19)
        db      27,'[13;35H',186,'Внимание!',186 ;(20)
        db      27,'[14;35H',200,9 dup (205),188 ;(21)
        db      27,'[0m',27, '[25;1H$' ;(22)
```

Первыми строками приведенного фрагмента с помощью функции DOS 09h и Esc-последовательности с именем clear очищается экран (предложения 1...3).

Далее в предложениях 4...9 осуществляется ввод с клавиатуры кода цвета и, наконец, с помощью еще одного вызова функции DOS 09h (предложения 10...12) на экран выводится сообщение message с Esc-последовательностями задания цвета и позиционирования курсора, а также отображаемыми строками (коды рамки и слово "Внимание!"). Команда безусловного перехода jmp (предложение 13) возвращает управление в точку ввода нового значения кода цвета, организуя тем самым бесконечный цикл. Пользователь может многократно вводить код цвета и наблюдать результат этого на экране. Для завершения программы и выхода в DOS надо ввести с клавиатуры сочетание <Ctrl>/<C>.

Строка message для удобства программирования разбита на части. Те два байта Esc-последовательности, которые определяют код цвета, выделены в отдельное предложение ассемблера с именем color. Далее с помощью четырех операторов db описан остаток выводимой строки до завершающего вывод символа "$".

Для ввода с клавиатуры одного символа используется функция DOS 01h. Эта функция отображает вводимый символ на экране, что дает

возможность контролировать ввод. Код ASCII нажатой клавиши возвращается ею в регистре AL. В рассматриваемом примере функция 01h вызывается дважды. Результат первого вызова (первый введенный с клавиатуры символ) пересылается в байт с именем color. Второй введенный символ помещается по адресу color+1, т.е. в следующий байт. В результате в Esc-последовательности задания цвета исходный код цвета '00' (что обозначает исходную комбинацию: белые символы по черному полю) заменяется на коды ASCII двухразрядного десятичного числа, введенного с клавиатуры (например, 31 - красные символы при исходном цвете фона, 44 - синий фон при исходном цвете символов и т.д.). С таким же успехом можно было, не выделяя байты кода цвета, переслать первый символ по адресу message+2, а второй - по адресу message+3, однако приведенный вариант программы нагляднее.

В примере 10.1 сохранены строки завершения программы (предложения 14-15). Однако в действительности они в этой программе не нужны. В самом деле, программа представляет собой бесконечный цикл и сама по себе никогда не завершится. Если же мы завершаем работу программы нажатием сочетания <Ctrl>/<C>, то функция завершения программы вызывается изнутри DOS. Таким образом, строки программы после предложения 13 с командой безусловного перехода никогда выполнены не будут, и их можно удалить.

Если ввод в программу с клавиатуры осуществляется с помощью одной из функций DOS посимвольного ввода, например, 01h, то для ввода нескольких символов эту функцию приходится выполнять повторно. В тех случаях, когда в программу требуется ввести сразу группу символов без их промежуточного контроля, удобнее воспользоваться функцией 3Fh, которая позволяет ввести с клавиатуры символьную строку длиной до 126 символов. Количество символов лимитируется длиной системного буфера, куда эти символы сначала поступают. Системный буфер имеет длину 128 байт, но последние два байта резервируются под коды 10 и 13, которые посылаются в буфер при нажатии клавиши <Enter>. Функция 3Fh вводит символы до нажатия клавиши <Enter>. Получив код этой клавиши, системные программы переносят содержимое системного буфера с введенной строкой (и кодами 10 и 13) в буфер пользователя, адрес которого указывается в регистре DX. Рассмотрим пример использования этой функции.

Пример 10.2. Ввод с клавиатуры символьной строки

```
;Выведем на экран символ запроса
        mov     AH,02h      ;(1)Функция вывода символа
        mov     DL,'>'      ;(2)Выводимый символ
        int     21h         ;(3)
;Поставим запрос на ввод строки с клавиатуры
        mov     AH,3Fh      ;(4)Функция ввода
        mov     BX,0        ;(5)Дескриптор клавиатуры
```

```
        mov     CX,128      ;(6)Вводим не более 128 символов
        mov     DX,offset inbuf;(7)Адрес буфера ввода
        int     21h         ;(8)
        mov     actlen,AX   ;(9)Сохраним число фактически
                            ;введенных символов
;Выведем для контроля на экран введенную строку другим цветом
        mov     AH,40h      ;(10)Функция вывода
        mov     BX,1        ;(11)Дескриптор экрана
        mov     CX,actlen   ;(12)Столько символов ввели
        add     CX,esc-2    ;(13)Прибавим esc-CR-LF
        mov     DX,offset outbuf;(14)Адрес выводимой строки
        int     21h         ;(15)
;Перейдем назад в монохромный режим
        mov     AH,09h      ;(16)Функция вывода
        mov     DX,offset reset;(17)Адрес выводимой строки
        int     21h         ;(18)
;Поля данных
actlen  dw 0   ;(19)Ячейка для числа введенных символов
outbuf  db 27,'[34;46m' ;(20)Esc-последовательность задания цвета
esc=$-outbuf   ;(21)Длина этой Esc-последовательности
inbuf   db 128 dup ('*');(22)Буфер ввода
reset   db 27,'[0m$';(23)Esc-последовательность отмены цвета
```

В начале приведенного фрагмента с помощью функции DOS 02h на экран выводится символ-запрос ">" (предложения 1...3). Появление этого символа на экране удостоверяет, что программа запустилась правильно и ждет реакции пользователя. Далее ставится запрос на ввод с клавиатуры символьной строки. Функция 3Fh может вводить данные из устройства (клавиатуры, последовательного порта) или файла. Источник данных определяется дескриптором, засылаемым в регистр BX. При загрузке системы клавиатуре присваивается дескриптор 0, который носит название "дескриптор стандартного ввода". В регистр CX засылается число, на 2 большее максимально возможного числа символов, так как в буфер пользователя фактически поступают, кроме введенной строки, еще и коды 13 (возврат каретки, CR) и 10 (перевод строки, LF). Регистр DX содержит адрес буфера ввода. В процессе ввода строки DOS отображает вводимые символы на экране (эхо), а после завершения ввода возвращает в регистре AX число введенных символов плюс два. Для дальнейших операций с введенной строкой ее длину следует сохранить (предложение 9).

Для контроля введенной строки в примере 9.2 предусмотрен вывод ее на экран с помощью функции вывода 40h. Для большей наглядности вывод выполняется в цвете, для чего непосредственно перед буфером ввода (предложение 22) в полях данных записана Esc-последовательность задания цвета символов (предложение 20). Поскольку в качестве адреса буфера вывода указан адрес этой Esc-последовательности outbuf, на экран (точнее, в драйвер ANSI.SYS) поступает сначала команда задания цвета, а затем введенная строка. Число выводимых символов скорректировано командой add (сложение) на

длину Esc-последовательности (константа esc) и кодов CR и LF, которые, таким образом, на экран не поступают.

Для определения длины Esc-последовательности (предложение 21) использовано понятие счетчика текущего адреса, который имеет символическое обозначение $. Счетчик текущего адреса фактически представляет собой ячейку памяти в программе транслятора, в которую, по мере трансляции программы, заносится смещение транслируемой в данный момент строки программы (напомним, что смещение, или относительный адрес - это номер соответствующего байта от начала данного сегмента). По окончания трансляции предложения 20 значение счетчика текущего адреса рано номеру первого байта после нашей Esc-последовательности (т.е., между прочим, смещению начала буфера ввода inbuf). Имена же полей данных actlen, outbuf и др. получают значения, равные смещениям этих полей от начала сегмента. Поэтому величина esc=$-outbuf, если ее вычислить сразу вслед за предложением, описывающим буфер outbuf, окажется равной длине этого буфера. Такая методика определения длин строк текста позволяет легко модифицировать эти строки. Действительно, если изменить состав строки outbuf и перетранслировать программу, константа esc автоматически получит новое значение.

Если содержательную часть программы завершить в этой точке, то заданный в программе цвет символов останется "навечно". Чтобы этого не произошло, в конце программы с помощью функции 09h на экран выводится Esc-последовательность отмены цвета (предложения 16...18).

С помощью рассмотренных средств в программу можно вводить недлинные строки текста: имена файлов, дату или время, режимы работы программы и т.д. Естественно, в программе должны быть предусмотрены средства обработки этих данных.

Статья 11

Анализ данных и условные переходы

При выполнении программы из предыдущей статьи читатель мог обратить внимание на незащищенность программы от неправильного ввода. Если вместо цифры случайно ввести букву, то Esc-последовательность потеряет смысл и будет рассматриваться ANSI-драйвером, как

обычная символьная строка. Это приведет к сбою в работе программы. Очевидно, что при вводе с клавиатуры в программе следует предусматривать анализ вводимой информации и отбраковку ошибочной. Внесем соответствующие дополнения в программу примера 10.1.

Пример 11.1. Анализ символьной информации, вводимой с клавиатуры.

```
;Очистим экран и выведем рамку с текстом без цвета
          ...
;Введем с клавиатуры параметр Esc-последовательности с его
;анализом
again:    mov     AH,01h      ;(1)Функция ввода с клавиатуры
          int     21h         ;(2)Вызов DOS
          cmp     AL,'4'      ;(3)Введен символ > '4'?
          ja      again       ;(4)Да, повторим ввод
          cmp     Al,'3'      ;(5)Введен символ < '3'?
          jb      again       ;(6)Да, повторим ввод
          mov     color,AL    ;(7)Введен допустимый код, перешлем
                              ;его в строку
symb2:    mov     AH,01h      ;(8)Функция ввода с клавиатуры
          int     21h         ;(9)Вызов DOS
          cmp     AL,'7'      ;(10)Введен символ > '7'?
          ja      symb2       ;(11)Да, повторим ввод
          cmp     AL,'0'      ;(12)Введен символ < '0'?
          jb      symb2       ;(13)Да, повторим ввод
          mov     color+1,AL  ;(14)Введен допустимый код, перешлем
                              ;его в строку
          mov     AH,9        ;(15)Функция DOS вывода на экран
          mov     DX,offset message;(16)Адрес выводимого сообщения
          int     21h         ;(17)Вызов DOS
          jmp     again       ;(18)Бесконечный цикл повторения
;Поля данных
clear     db      27,'[2J'
message   db      27,'['
color     db      '00'
          db      'm',27,'[12;35H',201,9 dup (205),187
          db      27,'[13;35H',186,'Внимание!',186
          db      27,'[14;35H',200,9 dup (205),188
          db      27,'[0m',27, '[25;1H$'
```

Рассмотрим процедуру проверки правильности вводимых данных. После ввода первого кода (предложения 1-2) выполняется сравнение с помощью команды cmp (compare, сравнение) содержимого регистра AL с символьным значением '4'. Поскольку коды цвета могут принимать лишь значения в диапазонах от 30 до 37 и от 40 до 47, первый символ не должен быть больше '4'. Если это не так, командой ja (jump if above, переход, если выше) осуществляется возврат на метку again с целью повторного ввода символа. Если же проверка прошла успешно, содержимое AL сравнивается с нижней границей допустимого диапазона '3'. Если введенный код меньше '3', командой jb (jump if below, переход, если ниже) осуществляется возврат на ту же метку again. Таким образом, программа будет требовать повторного ввода до тех пор, пока не будут

нажаты клавиши <3> или <4>. После проверки введенный символ пересылается в байт с именем color (предложение 7).

Далее с клавиатуры вводится вторая цифра кода цвета. Эта цифра проверяется на вхождение в диапазон от 0 до 7 и в случае правильности переносится в байт с адресом color+1. При вводе неправильного символа функция ввода вызывается повторно.

Статья 12

Пароли и сравнение строк

Идея простейшей защиты программы от несанкционированного запуска заключается в том, что где-то в программе записывается ключевое слово-пароль, и программа, начав работать, требует ввода этого слова с клавиатуры. Если пользователь ввел пароль правильно, программа продолжает работать. Если пароль введен неверно, программа завершается. Таким образом, программа должна сравнить введенное слово с хранящимся в ней и фиксировать совпадение или наличие хотя бы незначительной разницы. Рассмотрим программу ввода и анализа пароля.

Вообще говоря, ввод пароля можно осуществить любыми функциями ввода с клавиатуры. Однако обычно пароль вводят одной из функций, не отображающих вводимые символы на экране. Таких функций две - 07h и 08h. Разница между ними заключается в том, что функция 08h, зафиксировав ввод пользователем сочетания <Ctrl>/<C>, аварийно завершает программу, функция же 07h к <Ctrl>/<C> нечувствительна. Поскольку обе эти функции вводят лишь один символ, для ввода пароля их надо использовать в цикле. Выход из цикла можно организовать по-разному: после ввода обусловленного числа символов, по нажатию клавиши <Enter>, либо как-то иначе. Рассмотрим простую программу ввода с клавиатуры и анализа введенного слова.

Пример 12.1. Ввод пароля и сравнение символьных строк.

```
;Выведем запрос prompt с помощью функции DOS 09h
        ...
;Введем пароль
        mov     BX,0        ;(1)Инициализируем индексный регистр
pass:   mov     AH,08h      ;(2)Функция ввода без эха
        int     21h         ;(3)Вызов DOS
        cmp     AL,13       ;(4)<Enter>?
```

```
        je      compare     ;(5)Да, на сравнение
        mov     string[BX],AL;(6)Нет, сохраним символ
        inc     BX          ;(7)Инкремент индекса
        jmp     pass        ;(8)Повторять
;Будем сравнивать строки
compare:push    DS          ;(9)Настроим ES на наш
        pop     ES          ;(10)сегмент данных
        mov     SI,offset string;(11)Смещение одной строки
        mov     DI,offset password;(12)Смещение другой строки
        cld                 ;(13)Направление вперед
        mov     CX,BX       ;(14)Инициализируем счетчик цикла
repe    cmpsb               ;(15)Сравнение
        jne     err         ;(16)Строки не равны
;Выведем сообщение ok, подтверждающее правильность пароля
        ...
exit:   mov     AX,4C00h    ;(17)Завершение программы
        int     21h         ;(18)и выход в DOS
err:    jmp     begin       ;(19)Повторить ввод пароля
;Поля данных
passworddb      'camel'     ;(20)Ожидаемый пароль
string  db      80 dup (?) ;(21)Поле для ввода пароля
prompt  db      '>>$'       ;(22)Запрос
ok      db      'Работаем!$';(23)Сообщение об успешном вводе пароля
```

Работа программы начинается с вывода на экран запроса программы. В данном случае запрос имеет вид ">>". Далее инициализируется регистр BX, который будет использоваться в качестве индексного, и ставится запрос к DOS на ввод символа без эха (предложения 2-3). Введенный код сравнивается с кодом клавиши <Enter> (предложение 4), и в случае равенства кодов командой je (jump if equal, переход, если равно) выполняется переход на участок сравнения строк. Если же нажата любая другая клавиша, ее код ASCII заносится в поле string со смещением относительно начала этого поля на число байтов, равное содержимому регистра BX. Выполняется инкремент регистра BX и командой безусловного перехода jmp управление передается в точку pass на ввод следующего символа.

При нажатии клавиши <Enter> выполняется настройка регистров для выполнения операции сравнения. Для сравнения последовательностей байтов или слов предусмотрены специальные команды cmpsb (compare string byte, сравнение строк по байтам) и cmpsw (compare string word, сравнение строк по словам). Строкой применительно к эти командам называется любая последовательность байтов или слов безотносительно к ее содержимому. Первая строка адресуется через пару регистров DS:SI, вторая - через регистры ES:DI. Однократное выполнение команды сравнивает лишь одну пару элементов строки (один байт или одно слово). После операции сравнения регистры SI и DI получают положительное или отрицательное приращение, величина которого составляет 1 или 2 в зависимости от размера сравниваемых элементов. Знак приращения зависит от состояния флага процессора DF. Если этот

флаг сброшен, что осуществляется с помощью команды cld (clear direction flag, сброс флага направления), то приращение имеет положительный знак, т.е. элементы строки в процессе операции сравнения просматриваются вперед, от меньших адресов к большим. Если флаг DF установлен (с помощью команды std - set direction flag, установка флага направления), то приращение отрицательно, и строки просматриваются в сторону меньших адресов, от конца строки к ее началу.

Таким образом, перед выполнением команды cmpsb(w) необходимо предварительно настроить регистры DS, SI, ES и DI, а также флаг DF. Регистр DS мы обычно настраиваем в самом начале программы. Поскольку обе сравниваемые строки находятся в одном и том же сегменте данных (а могли бы находиться и в разных сегментах), сегментный регистр ES следует настроить так же, как и DS. Передача содержимого DS в ES осуществляется через стек в предложениях 9-10.

В предложениях 11-12 регистры SI и DI заполняются относительными адресами сравниваемых строк, а в предложении 13 командой cld устанавливается направление сравнения - от начала строки к ее концу.

Для того, чтобы сравнить целую строку, команда cmpsb или cmpsw предваряется префиксом повторения. В примере используется префикс repe (repeat while equal, повторение, пока равно). В этом случае операция сравнения выполняется до тех пор, пока символы двух строк совпадают, но не более CX раз. Поэтому требуется еще настроить и регистр CX. В нашем случае в него заносится конечное содержимое регистра BX, т.е. длина введенной с клавиатуры строки.

Выход из цикла повторения команды cmpsb происходит либо после обнаружения несовпадающих байтов (и тогда ясно, что строки не совпадают), либо по исчерпанию счетчика цикла CX. В последнем случае все байты строк, кроме последних, наверняка совпадают, однако результат сравнения последней пары байтов неясен. Таким образом, сама по себе команда cmpsb, выполняемая в цикле, не позволяет судить о результатах сравнения. После этой команды необходим анализ результата сравнения последней пары байтов одной из команд условного перехода. Использованная в программе команда jne (jump if not equal, переход, если не равно) в случае неравенства передает управление на метку err и весь цикл вывода запроса и ввода пароля повторяется. Если же строки полностью совпадают, выводится контрольное сообщение, после чего может начаться выполнение содержательной части программы.

Как уже отмечалось, для ввода пароля, наряду с функцией 08h, можно было использовать и функцию 07h, которая тоже осуществляет ввод с клавиатуры одного символа без эха, но при этом не реагирует на ввод комбинации <Ctrl>/<C>. В нашей программе, где ввод пароля выполняется в бесконечном цикле, использование функции 07h привело бы к тому, что пользователь, не знающий пароль и запустивший

программу, был бы вынужден для продолжения работы перезагружать компьютер. В системе MS-DOS не предусмотрено никаких средств выхода из зациклившейся программы, кроме перезагрузки системы. В реальных программах часто задают максимальное число попыток ввода пароля, после чего программа сама завершается.

Статья 13

Управление программой с клавиатуры и расширенные коды ASCII

Программы для персональных компьютеров носят, как правило, диалоговый характер. На определенных этапах выполнения программы пользователь должен иметь возможность ввести с клавиатуры указание о том, что должна делать программа дальше. При этом алфавитно-цифровые клавиши обычно используются для ввода в программу текстовой информации, а управление программой осуществляется с помощью функциональных клавиш <F1>...<F10>, <Home>, <PgUp>, <PgDn> и др. или комбинаций управляющих клавиш с функциональными и алфавитно-цифровыми, например, <Alt>/<1>, <Shift>/<F1>, <Ctrl>/<F1>, и т.д. Вся эта техника основана на использовании расширенных кодов ASCII.

Как было упомянуто в статье 10, при нажатии алфавитно-цифровой клавиши в буфер ввода с клавиатуры поступает двухбайтовый код, в котором старший байт соответствует скен-коду нажатой клавиши, а младший - коду ASCII закрепленного за ней символа. Программы DOS, принимающие данные с клавиатуры (функции 01h, 07h, 08h, 3Fh и др.), передают в программу только код ASCII, оставляя скен-код без внимания. Однако это относится только к тем клавишам, которые закреплены за символами, отображаемыми на экране (буквы, цифры, знаки препинания и др.). Кроме них, на клавиатуре персонального компьютера имеется ряд клавиш, которым не назначены какие-либо отображаемые на экране символы. Это, например, функциональные клавиши <F1>...<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 преобразуются в коды ASCII?

В таблице преобразования, с которой работает системная программа обслуживания клавиатуры, 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.

Все функции DOS, предназначенные для посимвольного ввода данных с клавиатуры, позволяют работать и с расширенными кодами ASCII. Для приема с клавиатуры расширенного кода ASCII программа должна вызвать функцию DOS дважды. Первый вызов возвращает младший байт расширенного кода ASCII, т.е. 0. При втором вызове возвращается старший, значащий байт. При этом возникает некоторая неоднозначность. При нажатии "обычных" клавиш каждый вызов функции DOS вводит код одной клавиши. Если же нажата функциональная клавиша, то на ввод кода нажатой клавиши требуется два вызова DOS. Поэтому программа, ожидающая поступления расширенных кодов ASCII, должна проверять каждый введенный код и при поступлении кода 0 выполнять ввод вторично.

Широкое использование интерактивных средств потребовало расширения возможностей ввода с клавиатуры управляющей информации, 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>. Помимо этого, функциональные клавиши и клавиши управления курсором генерируют различные расширенные коды ASCII в зависимости от того, нажаты ли они в одиночку или в сочетании с клавишами <Alt>, <Ctrl> или <Shift>. В табл. 13.1...13.5 приведены значения информационных байтов расширенных кодов ASCII для наиболее употребительных клавиш и их сочетаний.

Таблица 13.1. Информационные байты расширенных кодов ASCII функциональных клавиш

Клавиша	Код		Клавиша	Код		Клавиша	Код	
	Dec	Hex		Dec	Hex		Dec	Hex
<F1>	59	3Bh	<F5>	63	3Fh	<F9>	67	43h
<F2>	60	3Ch	<F6>	64	40h	<F10>	68	44h
<F3>	61	3Dh	<F7>	65	41h	<F11>	133	85h
<F4>	62	3Eh	<F8>	66	42h	<F12>	134	86h

Таблица 13.2. Информационные байты расширенных кодов ASCII функциональных клавиш в сочетании с клавишей <Shift>

Клавиша	Код Dec	Код Hex	Клавиша	Код Dec	Код Hex	Клавиша	Код Dec	Код Hex
<F1>	84	54h	<F5>	88	58h	<F9>	92	5Ch
<F2>	85	55h	<F6>	89	59h	<F10>	93	5Dh
<F3>	86	56h	<F7>	90	5Ah	<F11>	135	87h
<F4>	87	57h	<F8>	91	5Bh	<F12>	136	88h

Таблица 13.3. Информационные байты расширенных кодов ASCII функциональных клавиш в сочетании с клавишей <Ctrl>

Клавиша	Код Dec	Код Hex	Клавиша	Код Dec	Код Hex	Клавиша	Код Dec	Код Hex
<F1>	94	5Eh	<F5>	98	62h	<F9>	102	66h
<F2>	95	5Fh	<F6>	99	63h	<F10>	103	67h
<F3>	96	60h	<F7>	100	64h	<F11>	137	89h
<F4>	97	61h	<F8>	101	65h	<F12>	138	8Ah

Таблица 13.4. Информационные байты расширенных кодов ASCII функциональных клавиш в сочетании с клавишей <Alt>

Клавиша	Код Dec	Код Hex	Клавиша	Код Dec	Код Hex	Клавиша	Код Dec	Код Hex
<F1>	104	68h	<F5>	108	6Ch	<F9>	112	70h
<F2>	105	69h	<F6>	109	6Dh	<F10>	113	71h
<F3>	106	6Ah	<F7>	110	6Eh	<F11>	139	8Bh
<F4>	107	6Bh	<F8>	111	6Fh	<F12>	140	8Ch

Таблица 13.5. Информационные байты расширенных кодов ASCII алфавитно-цифровых клавиш в сочетании с клавишей <Alt>

Клавиша	Код Dec	Код Hex	Клавиша	Код Dec	Код Hex	Клавиша	Код Dec	Код Hex	Клавиша	Код Dec	Код Hex
A	30	1Eh	J	36	24h	S	31	1Fh	1	120	78h
B	48	30h	K	37	25h	T	20	14h	2	121	79h
C	46	2Eh	L	38	26h	U	22	16h	3	122	7Ah
D	32	20h	M	50	32h	V	47	2Fh	4	123	7Bh
E	18	12h	N	49	31h	W	17	11h	5	124	7Ch
F	33	21h	O	24	18h	X	45	2Dh	6	125	7Dh
G	34	22h	P	25	19h	Y	21	15h	7	126	7Eh
H	35	23h	Q	16	10h	Z	44	2Ch	8	127	7Fh
I	23	17h	R	19	13h	0	129	81h	9	128	80h

Рассмотрим несколько примеров управления программами с клавиатуры. Пусть наша программа вводит с клавиатуры текстовую информацию, и мы хотим предусмотреть возможность аварийного завершения программы при нажатии на функциональную клавишу <F10>.

Пример 13.1. Завершение программы по <F10>

```
;Выведем на экран запрос программы prompt
        ...
        mov     BX,offset txt;Инициализация базового
        mov     SI,0        ;и индексного регистров
inpt:   mov     AH,01h      ;Функция ввода символа с эхом
        int     21h         ;Вызов DOS
        cmp     AL,0        ;Расширенный ASCII-код?
        je      ex_ascii    ;Да, на анализ
        mov     [BX][SI],AL;Нет, символ в буфер
        inc     SI          ;Инкремент индекса
        jmp     inpt        ;И бесконечно повторять
ex_ascii:mov    AH,08h      ;Ввод символа без эха
        int     21h         ;Вызов DOS
        cmp     AL,44h      ;<F10>?
        je      exit        ;Да, на выход
        jmp     inpt        ;Нет, на продолжение ввода
exit:   ...                 ;Завершение программы
;Поля данных
prompt  db      '>>$'
txt     db      80 dup (?)
```

Программа выводит на экран свой запрос ">>", инициализирует регистры BX и SI, которые будут использоваться при записи принимаемых символов в буфер программы, и ставит запрос к DOS на ввод символа с клавиатуры с отображением его на экране (функция 01h). Если код принятого символа равен 0, программа переходит на метку ex_ascii для приема второго байта расширенного кода ASCII и его анализа. Если же поступил код алфавитно-цифровой клавиши, он записывается в буфер txt. В программе используется эффективный способ записи элемента массива с помощью команды mov с базово-индексной адресацией. В этом случае адрес определяется как сумма содержимого обоих используемых в команде регистров. В одном из них (в данном случае BX) хранится базовый адрес массива, в другом (SI) - индекс, т.е. смещение от начала массива. Эффективность команды определяется тем, что исполнительный адрес, т.е. адрес, по которому происходит обращение, не надо извлекать из памяти: его составляющие уже хранятся в регистрах процессора. После занесения введенного символа в буфер txt выполняется инкремент индекса, после чего программа возвращается на ввод очередного символа.

В случае нажатия клавиши или сочетания клавиш, вырабатывающих расширенный код ASCII, программа переходит на метку ex_ascii с целью вторичного выполнения функции DOS ввода символа. Использо-

вание здесь функции 08h, которая работает без эха, позволяет избавиться от отображения на экране в виде бессмысленных символов вторых байтов расширенных кодов ASCII. Принятый функцией 08h код сравнивается с расширенным кодом ASCII клавиши <F10> (44h) и в случае равенства осуществляется выход из программы. Если нажата другая функциональная клавиша, ввод текста продолжается.

Приведенный пример не полон в том отношении, что при аварийном завершении программы результаты ее работы безвозвратно теряются. В реальной программе после ввода пользователем команды завершения (нажатие клавиши <F10>) следует предусмотреть сохранение на диске результатов ее работы, вывода на экран предупреждающих сообщений и проч.

Использованный в примере 13.1 метод управления программой характерен тем, что воздействовать на ход выполнения программы можно лишь в тех точках программы, где она ожидает ввода символа с клавиатуры. Во многих случаях желательно иметь средство управления программой, в частности, ее аварийного завершения, в то время, когда программа выполняет некоторые циклические действия, например, обрабатывает и выводит на экран графическое изображение. Если в такой цикл включить вызов функции DOS 01h или 08h, программа, вместо того, чтобы выводить на экран изображение, остановится в ожидании нажатия клавиши. Однако в DOS предусмотрены средства ввода команды с клавиатуры без остановки программы. Для этого используется функция 06h, которая может работать в двух режимах: ввода одиночных символов с клавиатуры и вывода одиночных символов на экран. Эта функция позволяет программе "заглянуть" в буфер ввода с клавиатуры и при наличии в нем символа ввести его в программу, а при отсутствии такового просто продолжить выполнение. Индикатором наличия или отсутствия ожидающего символа является флаг нуля ZF. Если функция обнаружила ожидающий ввода символ, флаг ZF сбрасывается. Если символа нет, флаг ZF устанавливается (обнаружено 0 символов).

В примере 13.2 продемонстрировано использование функции DOS 06h для управления программой в цикле. Управление, как и в предыдущем примере, заключается в аварийном завершении программы при нажатии определенной клавиши (конкретно сочетания <Alt>/<Q>).

Пример 13.2. Завершение циклической программы по <Alt>/Q

```
contrl: mov   AH,09h      ;Будем в цикле
        mov   DX,offset string;выводить на экран
        int   21h         ;строку string
        mov   CX,0        ;Программная
qqq:    loop  qqq         ;задержка
        mov   AH,06h      ;Функция прямого ввода
        mov   DL,0FFh     ;Режим ввода
        int   21h         ;Вызов DOS
```

```
        jnz     symb        ;Если символ есть, переход
        jmp     contrl      ;Символа нет, продолжить цикл
symb:   cmp     AL,0        ;Расширенный код ASCII?
        jne     contrl      ;Нет, продолжить цикл
        mov     AH,06h      ;Да, надо ввести второй байт
        mov     DL,0FFh     ;Режим ввода
        int     21h         ;Вызов DOS
        cmp     AL,10h      ;Нажато <Alt>/<Q>?
        je      exit        ;Да, на выход
        jmp     contrl      ;Нет, продолжить цикл
exit:   ...                 ;Завершим программы
;Поля данных
string  db      'Выполняется цикл  $'
```

В качестве бесконечного цикла, из которого осуществляется аварийный выход по нажатию <Alt>/<Q>, использован периодический (с небольшой задержкой) вывод на экран строки текста.

Для осуществления аварийного выхода из программы в цикл включен вызов функции DOS 06h. Эта функция может работать в двух режимах - ввода с клавиатуры и вывода на экран. Режим определяется содержимым регистра DL в момент вызова функции. Любой код в DL, кроме FFh, приводит в выводу на экран соответствующего этому коду символа. Код FFh (в таблице кодов ASCII ему отвечает "пустой" символ) включает режим ввода с клавиатуры. Команда jnz (jump if not zero, переход, если не нуль) анализирует результат выполнения функции 06h. Если за время прохождения текущего шага цикла не было нажатий клавиш, эта команда "не срабатывает", и следующей командой jmp contrl осуществляется переход на начало цикла. Если же за время шага цикла была нажата какая-либо клавиша, функция 06h сбрасывает флаг ZF, и команда jnz передает управление в точку symb. Здесь прежде всего следует проверить, какой код ASCII ожидает ввода в программу - обычный или расширенный. Если код обычный (не равный 0), он игнорируется и цикл продолжается. Между прочим, функция DOS 06h работает без эха, и введенный символ не искажает изображение на экране. Если введен расширенный код ASCII, функция 06h вызывается повторно для ввода второго, информационного байта. Далее командой cmp выполняется анализ этого байта. Код 10h - это значащая часть расширенного кода ASCII сочетания <Alt>/<Q>. Естественно, в качестве "клавиши завершения" можно было выбрать любое другое сочетание. Если нажато <Alt>/<Q>, программа завершается; в противном случае цикл продолжается. В результате программный цикл не воспринимает никакие нажатия клавиш, кроме <Alt>/<Q>.

Статья 14

Вывод простейших графических изображений

В предыдущих статьях было показано, что для осуществления ввода с клавиатуры и вывода на экран символьной информации приходится прибегать к функциям DOS. При этом выяснилось, что возможности DOS довольно скромны. DOS не поддерживает ни позиционирование курсора, ни смену цвета выводимых символов. В текстовом режиме расширить возможности DOS можно с помощью драйвера ANSI.SYS. С графическими изображениями дело обстоит хуже, так как в DOS нет никаких графических функций. Нет их также и в драйвере ANSI.SYS, за исключением возможности перевода видеоадаптера в графический режим (с помощью Esc-последовательности Esc[=*режим*h). Для того, чтобы вывести на экран графическое изображение, необходимо воспользоваться нижним уровнем операционной системы - базовой системой ввода-вывода (Basic In-Out System, BIOS). Программы BIOS находятся в постоянном запоминающем устройстве (ПЗУ) BIOS, которым в обязательном порядке комплектуется любой компьютер. В отличие от DOS, ко всем функциям которой мы обращаемся с помощью прерывания 21h, в BIOS за каждым устройством компьютера закреплено свое прерывание. Так, программирование диска осуществляется с помощью прерывания int 13, клавиатуры - int 16h, экрана - int 10h. Прерывание int 10h обеспечивает все функции видеоадаптера: смену видеорежима, вывод символьной и текстовой информации, смену шрифтов, настройку цветовой палитры, работу с графическим изображением и т.д. Воспользуемся прерыванием int 10h для перехода в графический режим и вывода простейшего графического изображения.

Пример 14.1. Вывод на экран горизонтальной прямой.

```
;Установим графический режим EGA
        mov     AH,00h      ;(1)Функция задания режима
        mov     AL,10h      ;(2)Графический режим EGA
        int     10h         ;(3)Вызов BIOS
;Нарисуем прямую линию в цикле по X
        mov     SI,150      ;(4)Начальная X-координата
        mov     CX,300      ;(5)Число точек по горизонтали
line:   push    CX          ;(6)Сохраним его в стеке
        mov     AH,0Ch      ;(7)Функция вывода пиксела
        mov     AL,4        ;(8)Цвет красный
```

```
        mov     BH,0        ;(9)Видеостраница
        mov     CX,SI       ;(10)X-координата (переменная)
        mov     DX,175      ;(11)Y-координата (константа)
        int     10h         ;(12)Вызов BIOS
        inc     SI          ;(13)Инкремент X-координаты
        pop     CX          ;(14)Восстановим счетчик шагов
        loop    line        ;(15)Цикл из CX шагов
;Остановим программу для наблюдения результата ее работы
        mov     AH,08h      ;(16)Функция ввода с клавиатуры без эха
        int     21h         ;(17)Вызов DOS
;Переключим видеоадаптер назад в текстовый режим
        mov     AH,00h      ;(18)Функция задания режима
        mov     AL,03h      ;(19)Текстовый режим
        int     10h         ;(20)Вызов BIOS
```

В предложениях 1...3 с помощью функции 00h прерывания BIOS 10h осуществляется переключение видеоадаптера в графический режим. Поскольку номер режима заносится в байтовый регистр AL, всего может существовать 256 различных текстовых и графических режимов, из которых на сегодняшний день используются (аппаратурой различных фирм) около ста. Режим 10h обеспечивает вывод графического изображения 16 цветами с разрешением 640x350 точек и широко используется с видеоадаптерами EGA и VGA.

Изображение рисуется по точкам (в BIOS не предусмотрено программных средств вывода каких-либо геометрических фигур или хотя бы линий, как нет и средств закрашивания областей экрана). Для вывода на экран цветной точки (пиксела) используется функция 0Ch прерывания 10h. Эта функция требует занесения в регистр AL кода цвета, в BH - номера видеостраницы, в CX - X-координаты выводимой точки в диапазоне 0...349, а в DX - Y-координаты точки в диапазоне 0...639. Поскольку регистр CX используется, как счетчик шагов в цикле, для хранения X-координаты зарезервирован регистр SI.

Прямая горизонтальная линия в примере 14.1 рисуется путем вызова функции 0Ch в цикле, в каждом шаге которого значение Y-координаты остается неизменным (175 в примере), а значение X-координаты увеличивается на 1 (предложение 13). После завершения цикла формирования изображения в программе предусмотрена остановка (предложения 16-17) для того, чтобы пользователь мог, оставаясь в графическом режиме, проанализировать результаты работы программы. Для остановки программы используется функция DOS 08h ввода одного символа с клавиатуры. Функция 08h, как уже отмечалось, не отображает введенный символ на экране и, тем самым, не искажает графическое изображение. Нажатие любой клавиши (кроме управляющих - Ctrl, Alt, Shift и др.) возобновляет выполнение программы.

В конце рассматриваемого фрагмента предусмотрено переключение видеоадаптера в стандартный текстовый режим с номером 03h (предложения 18...20). Если такое переключение не выполнить, видеоа-

даптер останется в графическом режиме, что может помешать правильному выполнению прикладных программ.

Рассмотрим кратко параметры вызова функции 0Ch прерывания 10h. В регистр BH заносится номер видеостраницы, на которую выводится данная точка. Графический адаптер EGA обеспечивает хранение и отображение двух графических страниц. По умолчанию видимой (активной) делается страница 0, однако рисовать изображение можно как на видимой, так и на невидимой странице. Для переключения страниц предусмотрена функция 05h прерывания 10h.

В регистр AL заносится код цвета точки. 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, коды цветов которой приведены в табл. 14.1.

Таблица 14.1. Коды цветов стандартной цветовой палитры EGA

Код Hex	Код Dec	Цвет	Код Hex	Код Dec	Цвет
0	0h	Черный	8	8h	Серый
1	1h	Синий	9	9h	Голубой
2	2h	Зеленый	10	Ah	Салатовый
3	3h	Бирюзовый	11	Bh	Светло-бирюзовый
4	4h	Красный	12	Ch	Розовый
5	5h	Фиолетовый	13	Dh	Светло-фиолетовый
6	6h	Коричневый	14	Eh	Желтый
7	7h	Белый	15	Fh	Ярко-белый

Теперь вы умеете выводить на экран точку и можете при желании сформировать любой рисунок. Режим EGA можно использовать практически с любым современным видеоадаптером. Исключение составляют лишь монохромные адаптеры, имеющие специальное применение, а также устаревшие и почти не встречающиеся адаптеры CGA.

Однако большинство из выпускающихся сейчас видеоадаптеров относятся к классу SVGA (Super VGA). Хотя они и допускают использование режимов EGA и VGA, однако позволяют выводить графические изображения со значительно лучшими характеристиками. В таблице 14.2 приведены некоторые режимы видеоадаптеров SVGA.

К сожалению, SVGA не является стандартом, как EGA или VGA. Хотя стандарт для SVGA был предложен ассоциацией по стандартизации в видеоэлектронике (Video Electronics Standards Assotiation, или сокращенно VESA), он поддерживается большинством, но не всеми изготовителями видеоадаптеров.

Если стандарт VESA поддерживается, то определенные им функции записываются производителями видеоадаптеров в ПЗУ самого адаптера. Они называются расширением прерывания BIOS 10h - VESA BIOS Extention или VBE. Для вызова функции VBE в регистр AH необходимо записать 4Fh, а в регистр AL номер функции. При этом функция может не выполниться по нескольким причинам. Она может отсутствовать в вашей версии VBE. Может быть и такая ситуация, при которой в VBE эта функция есть, но ее не поддерживает апаратура видеоадаптера. Если VBE поддерживает эту функцию, то в регистре AL возвращается значение 4Fh. Иначе возвращается другое число. В случае успешного выполнения функции в AL возвращатся 0, а при ошибке - 1. Если в регистре AH возвращается 2h, это означает, что данную функцию не поддерживает аппаратура видеоадаптера.

Таблица 14.2. Некоторые режимы видеоадаптеров SVGA

Разрешение в пикселах	Количество цветов, из которых можно выбирать при выводе на экран, то есть палитра	Номер режима
800*600	256	103h
800*600	16777216	115h
1024*768	65536	117h
1280*1024	256	107h

Спектр характеристик видеоадаптеров SVGA очень широк, поэтому составление программы, формирующей графическое изображение и способной вывести его не на один конкретный видеоадаптер а на любой из заданной группы, особенно учитывая отсутствие стандарта, может представлять достаточно трудоемкую задачу. Функции VBE позволяют определить тип установленного видеоадаптера и разрешенные графические режимы. Для простоты предположим, что используемый нами режим поддерживается видеоадаптером и посмотрим как изменится приведенный выше фрагмент программы вывода на экран горизонтальной прямой. Выберем режим 103h.

Пример 14.2. Вывод на экран горизонтальной прямой в режиме 103h SVGA

```
;Установим графический режим 103h SVGA
        mov     AH,4Fh      ;(1)Функция вызова Video BIOS Extention
        mov     AL,02h      ;(2)Подфункция установки графического
                            ;режима
        mov     BX,103h     ;(3)Графический режим SVGA
        int     10h         ;(4)Прерывание BIOS
        cmp     AH,0        ;(5)При ошибке
        jne     errmes1     ;(6)Переход на сообщение
;Установим в регистре 150 таблицы цветов зеленый цвет максимальной
;яркости
```

```
        mov     AH,10h      ;(13)Функция управления регистрами
                            ;палитры
        mov     AL,10h      ;(14)Подфункция установки регистра
                            ;цветов
        mov     BX,150      ;(15)Номер регистра таблицы цветов(0-
                            ;255)
        mov     DH,0        ;(16)Интенсивность красного цвета (6
                            ;бит)
        mov     CH,127      ;(17)Интенсивность зеленого цвета (6
                            ;бит)
        mov     CL,0        ;(18)Интенсивность синего цвета (6 бит)
        int     10h         ;(19)Прерывание BIOS
;Нарисуем прямую линию в цикле по x
        mov     SI,0        ;(20)Начальная X - координата
        mov     CX,800      ;(21)Число точек по горизонтали
line:   push    CX          ;(22)Сохраним его в стеке
        mov     AH,0Ch      ;(23)Функция вывода пиксела
        mov     AL,150      ;(24)Цвет зеленый
        mov     BH,0        ;(25)Видеостраница
        mov     CX,SI       ;(26)X - координата (переменная)
        mov     DX,300      ;(27)Y - координата (константа)
        int     10h         ;(28)Вызов BIOS
        inc     SI          ;(29)Инкремент X - координаты
        pop     CX          ;(30)Восстановление счетчика шагов
        loop    line        ;(31)Цикл из CX шагов
;Остановим программу для наблюдения результатов ее работы
        mov     AH,08h      ;(32)Функция ввода с клавиатуры без эха
        int     21h         ;(33)Вызов DOS
;Переключим видеоадаптер назад в текстовый режим
        mov     AX,3        ;(34)Установка текстового режима
        int     10h         ;(35)Вызов BIOS
        jmp     output      ;(36)
errmes1:
        mov     AH,09h      ;(37)Функция вывода сообщения
        mov     DX,offset message1;(38)Смещение сообщения
        int     21h         ;(39)Вызов DOS
output:
...
```

Сначала необходимо установить требуемый режим работы. Для этого воспользуемся функцией 02h. Загрузим в регистр AH - 4Fh, в регистр AL - 02h, а в регистр BX номер режима VESA и выполним прерывание BIOS 10h. Если графический режим установлен, то после выполнения прерывания в регистре AH возвращается 0. Поэтому в предложении 5 мы проверяем содержимое регистра AH, и в случае неудачи выводим сообщение об ошибке - предложения 37, 38, 39.

Перед выводом отрезка прямой нам требуется определить ее цвет. Поскольку мы установили режим 103h, в котором используется 256 цветов, то определим цвет с помощью функции 10h установки регистров палитры. Вызов этой функции выполняется в предложениях 13 - 19. В предложении 14 определяется номер подфункции, тоже 10h, которая позволяет непосредственно задать цвет для любого из 256 регистров

таблицы цветов. Интенсивности красной, зеленой и синей составляющих задаются в регистрах DH, CH и CL. Поскольку мы решили вывести линию зеленого цвета с максимальной яркостью, то в регистр CH заносим максимальное значение 2^6-1=127, а в остальные регистры нули. Номер регистра таблицы цветов определяем в регистре BX.

Остальные предложения практически идентичны соответствующим предложениям фрагмента, приведенного в примере 14.1, за исключением предложений 20, 21, 24 и 27. Поскольку линию будем рисовать с самого края и до конца экрана, в предложении 20 в регистр SI заносим 0, а в предложении 22 в регистр CX - 800. Чтобы расположить линию по середине экрана при выбранном разрешении в 600 точек по вертикали, в предложении 27 в регистр DX заносим 175. Для определения цвета в AL заносим номер регистра таблицы цветов, содержимое которого мы определили в предложениях 13...19, а именно 150.

Статья 15

Подпрограммы

В предыдущих примерах нам удалось построить простейшее изображение, включив фрагмент вывода на экран точки в цикл. Однако при построении более сложных изображений с не столь упорядоченным изменением координат программа окажется очень громоздкой. Существенного упрощения структуры программы можно достичь, используя в ней подпрограммы.

Модифицируем программу из примера 14.1, разбив ее на процедуры и организовав в цикле обращение к подпрограмме с передачей ей параметров. Поскольку введение процедур несколько изменяет структуру программы, пример 15.1 приведен не фрагментарно, а полностью, включая описания сегментов.

Пример 15.1. Вывод на экран горизонтальной прямой с помощью подпрограммы.

```
text    segment'code'     ;(1)Начало сегмента команд
        assume CS:text,DS:dat ;(2)
;Подпрограмма вывода одной точки. Параметры при вызове находятся в
;ячейках памяти: color - цвет точки, vpage - видеостраница,
;x - X-координата, y - Y-координата
draw    proc              ;(3)Объявление процедуры-
                          ;подпрограммы
```

```
        mov     AH,0Ch      ;(4)Функция вывода пиксела
        mov     AL,color    ;(5)Цвет
        mov     BH,vpage    ;(6)Видеостраница
        mov     CX,x        ;(7)X-координата
        mov     DX,y        ;(8)Y-координата
        int     10h         ;(9)Вызов BIOS
        ret                 ;(10)Команда выхода из подпрограммы
draw    endp                ;(11)Конец процедуры
;Главная процедура, с которой начинается выполнение программы
main    proc                ;(12)Объявление главной процедуры
        mov     AX,data     ;(13)Инициализация сегментного
        mov     DS,AX       ;(14)регистра DS
;Установим графический режим EGA
        mov     AH,00h      ;(15)Функция задания режима
        mov     AL,10h      ;(16)Графический режим EGA
        int     10h         ;(17)Вызов BIOS
;Нарисуем горизонтальную линию в цикле по X
        mov     CX,300      ;(18)Число точек по горизонтали
line:   push    CX          ;(19)Сохраним его в стеке
        call    draw        ;(20)Вызов подпрограммы
        inc     x           ;(21)Инкремент X-координаты
        pop     CX          ;(22)Восстановим счетчик шагов
        loop    line        ;(23)Цикл из CX шагов
;Остановим программу для наблюдения результата ее работы
        mov     AH,08h      ;(24)Функция ввода с клавиатуры
        int     21h         ;(25)Вызов DOS
;Переключим видеоадаптер назад в текстовый режим
        mov     AH,00h      ;(26)Функция задания режима
        mov     AL,03h      ;(27)Текстовый режим
        int     10h         ;(28)Вызов BIOS
        mov     AX,4C00h    ;(29)Завершение программы
        int     21h         ;(30)
main    endp                ;(31)Конец главной процедуры
text    ends                ;(32)Конец сегмента команд
data    segment             ;(33)Начало сегмента данных
x       dw      150         ;(34)Текущая X-координата
y       dw      175         ;(35)Текущая Y-координата
color   db      14          ;(36)Цвет точек
vpage   db      0           ;(37)Видеостраница
data    ends                ;(38)Конец сегмента данных
stack   segment stack       ;(39)Начало сегмента стека
        dw 128 dup (0)      ;(40)Стек
stack   ends                ;(41)Конец сегмента стека
        end     main        ;(42)Конец текста программы
```

Наша программа состоит теперь из двух процедур - главной с именем main и процедуры-подпрограммы с именем draw. Каждая процедура начинается оператором proc, перед которым указывается имя процедуры, а заканчивается оператором endp (end procedure, конец процедуры)(пары предложений 3, 11 и 12, 31). Порядок процедур в программе в большинстве случае не имеет значения, однако имя главной процедуры, с которой начинается выполнение программы, должно быть указано в качестве операнда директивы end, завершающей текст программы (предложение 42).

Подпрограммы вызываются оператором call (вызов); каждая подпрограмма должна заканчиваться командой ret (return, возврат), которая передает управление в точку возврата, т.е. на команду вызывающей программы, следующую за командой call.

Подпрограмма draw выводит на экран одну точку. В качестве входных параметров она должна получить две координаты точки, ее цвет, а также номер видеостраницы, на которую выводится изображение. В языке ассемблера нет установленных правил передачи параметров подпрограмме. Их можно передать через регистры общего назначения, стек или ячейки памяти. В примере 15.1 используется последний способ, не самый быстрый, но наиболее наглядный. Для хранения и модификации параметров в сегменте данных предусмотрены ячейки x, y, color и vpage. В данном примере вывода горизонтальной линии в трех ячейках хранятся константы, и лишь ячейка x модифицируется.

При использовании подпрограммы основной цикл упрощается. Фактически в нем лишь две содержательные строки: вызов подпрограммы draw и инкремент X-координаты в ячейке x. Однако сохранение в стеке и восстановление регистра CX является обязательным, потому что он используется в подпрограмме для задания X-координаты.

Выделив вывод точки в подпрограмму и существенно упростив цикл основной программы, мы облегчили задачу ее модификации. В примере 15.2 показано (уже фрагментарно), как можно, в дополнение к горизонтальной, вывести на экран наклонную и вертикальную линию.

Пример 15.2. Вывод на экран пучка линий.

```
;Нарисуем горизонтальную линию, идущую из начала координат
        mov     CX,640      ;(1)Число точек по горизонтали
line:   push    CX          ;(2)Сохраним его в стеке
        call    draw        ;(3)Вызов подпрограммы
        inc     x           ;(4)Инкремент X-координаты
        pop     CX          ;(5)Восстановим счетчик шагов
        loop    line        ;(6)Цикл из CX шагов
;Нарисуем наклонную линию
        mov     x,0         ;(7)Восстановим начальное значение X
        mov     CX,350      ;(8)Число точек
line1:  push    CX          ;(9)Сохраним его в стеке
        push    x           ;(10)Отправим значение X
        pop     y           ;(11)в ячейку Y
        call    draw        ;(12)Вызов подпрограммы
        inc     x           ;(13)Инкремент X-координаты
        pop     CX          ;(14)Восстановим счетчик шагов
        loop    line1       ;(15)Цикл из CX шагов
        mov     CX,640      ;(16)Число точек по горизонтали
;Нарисуем вертикальную линию
        mov     x,0         ;(17)Восстановим исходное значение X
        mov     y,0         ;(18)Восстановим исходное значение Y
        mov     CX,350      ;(19)Число точек
line2:  push    CX          ;(20)Сохраним его в стеке
        call    draw        ;(21)Вызов подпрограммы
```

```
        inc     y           ;(22)Инкремент Y-координаты
        pop     CX          ;(23)Восстановим счетчик шагов
        loop    line2       ;(24)Цикл из CX шагов
;Поля данных
x       dw      0           ;(25)Текущая X-координата
y       dw      0           ;(26)Текущая Y-координата
color   db      14          ;(27)Цвет точек
vpage   db      0           ;(28)Видеостраница
```

Рассмотрим некоторые важные детали программ 15.1 и 15.2.

В полях данных программ имеются 4 ячейки с параметрами подпрограммы. Две из них (color и vpage) описаны с помощью уже знакомой нам директивы db (определить байт); для двух других (x и y) использована директива dw (define word, определить слово). Почему директивы разные? Дело в том, что числа из ячеек color и vpage по ходу выполнения программы загружаются в байтовые регистры AL и BH; числа из ячеек x и y загружаются в 16-разрядные (словные) регистры CX и DX. 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. Поэтому при определении данных следует заранее подумать, в какие регистры эти данные будут загружаться.

В предложениях 10-11 примера 15.2 содержимое ячейки x копируется в ячейку y через стек. Вообще говоря, здесь надо было бы выполнить команду mov y,x. Однако в микропроцессорах 80x86 запрещена операция непосредственной пересылки операндов из одной ячейки памяти в другую. Такую пересылку можно осуществить только с помощью промежуточного регистра

```
        mov     AX,x
        mov     y,AX
```

или через стек, как это сделано примере 15.2 (можно было бы также воспользоваться командой пересылки строк movs, но это сложнее).

Программа вывода на экран трех линий состоит из трех схожих по структуре циклов. Начальные точки этих циклов обозначены схожими, но все же различающимися метками line, line1 и line2. В исходном тексте программы не должно быть нескольких одинаковых меток или имен ячеек памяти (даже если они относятся к разным процедурам или программным сегментам).

Каким должно быть взаимное расположение главной процедуры и подпрограмм? В обоих примерах главная процедура и процедура-подпрограмма располагались друг за другом, причем первой была описана процедура-подпрограмма. Такая структура не является обязательной. Процедуры могут располагаться в программе в любом порядке, при этом их можно вкладывать друг в друга. Например, возможен такой вариант структуры программы:

```
text    segment 'code'    ;Начало сегмента команд
main    proc              ;Объявление главной процедуры
        ...               ;Текст главной процедуры
        mov   AX,4C00h    ;Завершение программы
        int   21h         ;
draw    proc              ;Объявление процедуры-подпрограммы
        ...               ;Текст подпрограммы
        ret               ;Команда выхода из подпрограммы
draw    endp              ;Конец процедуры
main    endp              ;Конец главной процедуры
text    ends              ;Конец сегмента команд
        end   main
```

Здесь процедура-подпрограмма draw располагается внутри главной процедуры main после строк завершения программы (вызов функции DOS 4Ch). Последний момент является принципиально важным. Подпрограммы следует располагать таким образом, чтобы они не могли начать выполняться "сами по себе", без вызова командой call. Приведенный пример удовлетворяет этому требованию. Действительно, вызовом функции завершения 4Ch управление передается системе, в нашу программу уже никогда не вернется, и строки, стоящие после команды int 21h сами по себе выполняться не будут. В то же время приведенный ниже пример грубо неверен:

```
text    segment 'code'    ;Начало сегмента команд
main    proc              ;Объявление главной процедуры
draw    proc              ;Объявление процедуры-подпрограммы
        ...               ;Текст подпрограммы
        ret               ;Команда выхода из подпрограммы
draw    endp              ;Конец процедуры
        ...               ;Текст главной процедуры
        mov   AX,4C00h    ;Завершение программы
        int   21h         ;
main    endp              ;Конец главной процедуры
text    ends              ;Конец сегмента команд
        end   main
```

В такой программе после ее активизации сразу же начнется выполнение процедуры draw. Страшно здесь не то, что эта процедура выполнится "вне очереди", когда еще не установлен графический режим и не настроены параметры подпрограммы, а то, что она завершится выполнением команды ret, обратной по отношению к команде call. Однако команды call у нас не было и такая "непарная" команда ret приведет к зависанию системы. Для того, чтобы понять, что здесь происходит, надо разобраться в механизмах вызова подпрограмм и возврата из них. Этот вопрос будет рассмотрен в следующей статье.

Наконец, последнее замечание связано с ролью процедур вообще. Вспомним пример 1.1. В нем процедур не было, а в качестве точки входа в программу фигурировала метка begin. Процедуры не являются обязательным элементом программ на языке ассемблера, они лишь

повышают их наглядность. Пример 15.1 с подпрограммой тоже можно было построить без применения процедур, обозначив точки входа в главную программу и подпрограмму метками:

```
text    segment 'code'    ;Начало сегмента команд
draw:                     ;Точка входа в подпрограмму
        ...               ;Текст подпрограммы
        ret               ;Команда выхода из подпрограммы
main:                     ;Точка входа в главную программу
        ...               ;Текст главной процедуры
        call   draw       ;Вызов подпрограммы
        ...
        mov    AX,4C00h   ;Завершение программы
        int    21h        ;
text    ends              ;Конец сегмента команд
        end    main
```

Исходный текст программы получится даже короче, так как из него будут изъяты предложения proc и endp (загрузочный модуль программы от этого не изменится). Важно только соблюсти условие, о котором уже говорилось выше: подпрограммы должны размещаться таким образом, чтобы переход на их выполнение осуществлялся исключительно с помощью команд call. Попытка выполнить подпрограмму, как фрагмент главной программы, с большой вероятностью приведет к зависанию системы на команде ret.

Статья 16

Механизм вызова подпрограмм

Рассмотрим механизм выполнения конкретных команд call draw и ret из примера 15.1. На рис. 16.1 приведены фрагменты загрузочного модуля программы 15.1 с указанием расположения некоторых команд, их кодов, смещений, мнемонических обозначений и описания их действия. Показана также часть сегмента данных.

Сегмент команд начинается с процедуры draw. Первая команда этой процедуры mov AH,0Ch имеет поэтому смещение (относительный адрес в сегменте команд) 0000h. Процедура draw занимает 14h=20 байт с относительными адресами от 0000h до 0013h. Последней командой процедуры draw является однобайтовая команда ret с кодом C3h.

```
Смещение Код        Предложения          Действие
         команды    программы            команды

                    text segment 'code'
                    assume CS:text,DS:data
0000                draw proc
0000     B4 0C      mov AH,0Ch
                    ...
0013     C3         ret ──────────► из стека 0026 ──────► в IP
0014                draw endp
0014                main proc
0014     B8 4476    mov AX,data
0017     8E D8      mov DS,AX
                    ...
0023     E8 FFDA    call draw ──────► 1) IP=0026h ──────► в стек
                                      2) 0026h+FFDA=0000 ─► в IP
0026     FF 06 0000 inc x
                    ...
003C                main endp
                    text ends
                    data segment
0000     0096       x dw 150
0002     00AF       y dw 175
                    ...
0006                data ends
```

Рис. 16.1. Фрагменты загрузочного модуля программы 15.1 с поясняющей информацией.

За процедурой draw располагается главная процедура main. Ее первая команда mov AX,data имеет смещение 0014h. Код команды включает код операции mov (B8h) и значение имени data, равное сегментному адресу сегмента данных. При загрузке программы под управлением отладчика CodeView сегментный адрес data оказался равным 4476h.

Команда call draw расположена по адресу 0023h. В ее полный код входит код операции call (E8h) и адрес процедуры draw, на которую надо осуществить переход. Этот адрес записан в виде смещения к началу процедуры draw относительно текущего содержимого IP, т.е. относительно адреса следующей команды (в нашем случае команды inc x). Смещение это знаковое и в данном случае отрицательное, так как процедура draw располагается до процедуры main. Поскольку адрес draw равен 0, а адрес следующей команды равен 26h, в коде команды записано число -26h, которое по правилам записи отрицательных чисел выражается кодом FFDAh (знаковые числа будут рассмотрены в статье 36).

Главная процедура занимает 18h=24 байт, а первый свободный байт после конца этой процедуры имеет смещение 003Ch. На этом заканчивается сегмент команд. С ближайшего адреса, кратного 16 (44760h в нашем случае), начинается сегмент данных. Относительные адреса в нем опять начинаются с 0, поэтому смещение первой переменной x равно 0, смещение следующей переменной y - 2 и т.д. Весь сегмент данных занимает всего 6 байт.

Вернемся к рассмотрению команд call и ret. При выполнении команды call процессор помещает адрес возврата (содержимое IP, т.е. адрес следующей команды) в стек, а в IP заносит относительный адрес процедуры draw, который находится суммированием текущего содержимого IP и смещения, записанного в коде команды call. В результате указатель стека SP смещается вверх на одно слово, а процессор переходит на выполнение подпрограммы.

Команда ret выполняет обратную операцию - извлекает из верхнего слова стека (с восстановлением исходного состояния указателя стека SP) адрес возврата и загружает его в IP, в результате чего процессор возвращается к выполнению вызывающей процедуры.

Из сказанного ясно, что если в подпрограмме используется стек, с ним надо работать очень аккуратно: все, что заносится в стек в процессе выполнения подпрограммы, должно быть обязательно снято с него до выполнения команды ret, иначе эта команда извлечет из стека и загрузит в IP не адрес возврата, а какое-то данное, что заведомо приведет к нарушению выполнения программы.

Рассмотренный нами вызов подпрограммы носит название прямого ближнего (или внутрисегментного) вызова. Прямым такой вызов называется потому, что адрес перехода хранится непосредственно в коде команды (а это, в свою очередь, получилось потому, что мы указали в качестве операнда команды call имя подпрограммы). Если бы адрес подпрограммы хранился в каком-то другом месте (именно, в регистре или в ячейке памяти), то вызов был бы косвенным. Мы столкнемся с косвенными вызовами подпрограмм в последующих статьях. Вторая характеристика вызова говорит о том, что вызываемая подпрограмма находится в том же сегменте, что и вызывающая процедура. В этом случае для перехода на подпрограмму надо знать лишь "половину" полного адреса подпрограммы, именно, относительный адрес точки перехода. Сегментный адрес остается тем же; он не фигурирует в строке вызова подпрограммы и отсуствует в коде команды. В дальнейшем мы рассмотрим и другой вид подпрограмм - дальние подпрограммы, для обращения к которым следует применять межсегментные вызовы.

Статья 17

Преобразование шестнадцатеричных цифр в символьную форму

При отладке сложных программ, таких, как обработчики прерываний, драйверы и др., возникает необходимость динамического, по ходу выполнения отлаживаемой программы, вывода на экран содержимого регистров процессора или ячеек памяти. Такого рода информацию удобнее всего получать в 16-ричной форме. Однако, как мы уже выяснили ранее, на экран должны поступать не числа, а коды ASCII выводимых цифр. Таким образом, возникает задача преобразования двоичного 16-битового числа в четыре 16-ричные цифры, записанные в символьной форме. В настоящей статье будет рассмотрена первая часть этой задачи - преобразование одной 16-ричной цифры в символьную форму. Преобразование в символьную форму всего 16-битового числа будет описано в следующей статье.

Одна 16-ричная цифра описывает 4 двоичных разряда. Мы должны, в зависимости от содержимого каждых четырех двоичных разрядов, получать код ASCII соответствующей 16-ричной цифры от '0' до 'F. Подпрограмма, реализующая эту операцию, приведена в примере 17.1.

Пример 17.1. Преобразование шестнадцатеричной цифры в ASCII

```
hexasc  proc
;Подпрограмма получения символьного представления четырехбитового
;числа. На входе: исходное число в младшей половине AL, адрес
строки, куда надо поместить результат, в DS:SI. На выходе:
;ASCII-представление полученного числа в памяти по адресу DS:SI
;Исходное содержимое регистра AL разрушается
        push   BX          ;Сохраним используемый регистр
        mov    BX,offset tblhex;BX=адрес таблицы трансляции
        xlat               ;Команда табличной трансляции
        mov    [SI],AL     ;Сохраним результат
        pop    BX          ;Восстановим регистр BX
        ret                ;Возврат в вызывающую программу
hexasc  endp
main    proc
        ...
        mov    AL,15       ;Преобразуемое число
        mov    SI,offset result;Настроим SI
        call   hexasc      ;Вызов подпрограммы
        mov    AH,09h      ;Функция вывода строки
```

```
        mov     DX,offset result
        int     21h
        ...
main    endp
;Поля данных
tblhex  db      '0123456789ABCDEF'
result  db      '*h',10,13,'$';Выводимый текст
```

Пусть исходное число находится в младшем полубайте регистра AL. Для преобразования этого числа в код ASCII мы воспользуемся командой табличной трансляции xlat, которая позволяет осуществить выборку байта из таблицы по его индексу. Эта команда для своей работы требует настройки трех регистров. В регистрах DS:BX должен находиться полный двухсловный адрес таблицы трансляции (естественно, в регистре DS - сегментный адрес, а в регистре BX - относительный), а в регистре AL - смещение в таблице к выбираемому байту, т.е. его индекс. Команда xlat выбирает из таблицы указанный байт и возвращает его в регистре AL, разрушая тем самым его исходное содержимое. Условимся, что при вызове подпрограммы преобразования hexasc в регистре AL должно находиться исходное преобразуемое число, а в регистре SI - адрес ячейки памяти для результата преобразования.

В таблице с именем tblhex мы запишем (по байтам) символьные представления шестнадцатеричных цифр от '0' до 'F'. Для результата преобразования предусмотрим строку result, в первый байт которой будет пересылаться полученный символ. Для наглядного представления на экране результата работы программы строка result дополнена знаком шестнадцатеричного числа "h", кодами возврата каретки и перевода строки, а также завершающим для функции DOS 09h символом "$".

В главной процедуре main мы засылаем в регистр AL произвольное число от 0 до 15, в регистр SI - смещение поля result и вызываем подпрограмму hexasc. Далее для контроля правильности проведенного преобразования строка result выводится на экран.

Рассмотрим теперь работу процедуры hexasc. Поскольку это процедура общего назначения, которая может быть вызвана из любого места программы, ее надо написать таким образом, чтобы она не нарушала ход выполнения вызывающей программы. Для этого в первом же предложении процедуры сохраняется в стеке используемый в ней регистр BX. Далее BX настраивается на смещение таблицы трансляции. Поскольку в регистре AL уже находится преобразуемое число, которое, согласно построению таблицы tblhex, является индексом требуемого символа, можно выполнить команду xlat. Результат преобразования из регистра AL отправляется в ячейку памяти, на которую указывает регистр SI. Обозначение [SI] означает, что адресуется ячейка памяти, смещение которой находится в регистре SI. Такой способ адресации носит название косвенного (более конкретно - косвенного индексного,

поскольку для адресации используется один из индексных регистров). Далее восстанавливается сохраненный регистр BX и командой ret управление передается в вызывающую процедуру.

Небольшое замечание относительно сохранения регистров. Как правило, в начале подпрограммы сохраняются, а в конце восстанавливаются те регистры, содержимое которых разрушается в процессе работы подпрограммы. Обычно это не относится к регистрам, через которые передаются параметры. При этом в зависимости от алгоритма подпрограммы содержимое регистров с параметрами может сохраниться, но может и разрушиться. У нас получилось, что в процессе выполнения подпрограммы hexasc исходное содержимое регистра AL, через которые передается преобразуемое число, разрушается. Это обстоятельство следует иметь в виду при включении данной подпрограммы в программный комплекс.

Написав полный текст программы, выполните ее несколько раз, занося в регистр AL разные числа. Если эти изменения вводятся в исходном тексте, каждый раз программу следует заново транслировать и компоновать. Можно поступить иначе - запустить программу под управлением отладчика и в нем после выполнения команды основной процедуры mov AL,15 изменять содержимое регистра AL.

Статья 18

Преобразование беззнаковых двоичных чисел в символьную форму

Рассмотрим теперь программу преобразования в символьную форму всего 16-битового числа. Подпрограмма hexasc из предыдущей статьи преобразует в 16-ричную цифру младшую четверку битов регистра AL. Полное 16-битовое слово содержит четыре четверки битов. Таким образом, нам надо последовательно приложить процедуру hexasc ко всем этим четверкам, предварительно выделяя их из всего слова и помещая в младший полубайт регистра AL. Эту операцию выполняет подпрограмма binasc (пример 18.1), которая, выделив очередную четверку битов и установив указатель SI на соответствующий байт выходной строки, вызывает подпрограмму hexasc для получения одной 16-ричной цифры. Таким образом, программа 18.1 содержит, кроме основной процедуры

main, две процедуры-подпрограммы и является одновременно иллюстрацией вложенных вызовов подпрограмм.

Пример 18.1. Преобразование двоичного числа в 16-ричную символьную форму

```
hexasc  proc
;На входе: Число в младшей половине AL
;Адрес строки, куда надо поместить результат, в SI
        ...                 ;См. пример 17.1
        ret
hexasc  endp
binasc  proc
;Подпрограмма преобразования двоичного слова в 16-ричную
;символьную форму. При вызове: AX=преобразуемое число,
;DS:SI=адрес строки, куда помещается результат преобразования
        push    CX          ;(1)Сохраним используемый регистр
        push    AX          ;(2)Сохраним наше число в стеке
        and     AX,0F000h   ;(3)Выделим старшую четверку битов
        mov     CL,12       ;(4)Счетчик сдвига
        shr     AX,CL       ;(5)Сдвиг вправо на 12 бит
        call    hexasc      ;(6)Преобразуем в символ и отправим
                            ;по адресу [SI]
        pop     AX          ;(7)Вернем в AX исходное число
        push    AX          ;(8)И снова сохраним в стеке
        and     AX,0F00h    ;(9)Выделим вторую четверку битов
        mov     CL,8        ;(10)Счетчик сдвига
        shr     AX,CL       ;(11)Сдвиг вправо на 8 бит
        inc     SI          ;(12)Сдвинемся к следующему байту в
                            ;(13)строке результата
        call    hexasc      ;(14)Преобразуем в символ и отправим
                            ;по адресу [SI]
        pop     AX          ;(15)Вернем в AX исходное число
        push    AX          ;(16)И снова сохраним в стеке
        and     AX,0F0h     ;(17)Выделим третью четверку битов
        mov     CL,4        ;(18)Счетчик сдвига
        shr     AX,CL       ;(19)Сдвиг вправо на 4 бита
        inc     SI          ;(20)Сдвинемся к следующему байту в
                            ;строке результата
        call    hexasc      ;(21)Преобразуем в символ и отправим
                            ;по адресу [SI]
        pop     AX          ;(22)Вернем в AX исходное число
        push    AX          ;(23)И снова сохраним в стеке
        and     AX,0Fh      ;(24)Выделим младшую четверку битов
        inc     SI          ;(25)Сдвинемся к следующему байту в
                            ;строке результата
        call    hexasc      ;(26)Преобразуем в символ и отправим
                            ;по адресу [SI]
        pop     AX          ;(27)Восстановим стек
        pop     CX          ;(28)Восстановим регистр CX
        ret                 ;(29)Возврат в вызвавшую процедуру
binasc  endp
main    proc
        ...
        mov     AX,92C4h    ;Преобразуемое число
        mov     SI,offset result;Настроим SI
        call    binasc      ;Вызов подпрограммы
```

```
        mov     AH,09h      ;Функция вывода строки
        mov     DX,offset result;Адрес выводимой строки
        int     21h
        ...
main    endp
;Поля данных
result  db      '****h',10,13,'$';Выводимый текст
tblhex  db      '0123456789ABCDEF'
```

Пусть исходное число находится в регистре AX. Поскольку нам придется многократно обращаться к нему с целью выделения четверок битов, оно сразу сохраняется в стеке (предложение 2). Перед этим сохраняется в стеке содержимое используемого в подпрограмме регистра CX, чтобы подпрограмма не нарушила работу основной программы, в которой этот регистр тоже, возможно, используется. Команда логического умножения and (предложение 3) обнуляет все биты первого операнда, в данном случае AX, соответствующие сброшенным битам ее второго операнда (в данном случае числа F000h) и оставляет без изменения остальные биты первого операнда (рис. 18.1).

```
Исходное число    9    2    C    4h (16-ричное представление
                  ┌─┐  ┌─┐  ┌─┐  ┌─┐
Исходное число    1001 0010 1100 0100 (двоичное представление)
Операнд-маска     1111 0000 0000 0000
Результат в AX    1001 0000 0000 0000
```

Рис. 18.1. Результат действия команды and.

Мы выделили старшую четверку битов исходного числа, однако она находится в самом старшем полубайте регистра AX, в то время, как для правильной работы подпрограммы hexasc преобразуемая четверка должна располагаться в младшем полубайте AL. Команда логического сдвига вправо shr (shift right, сдвиг вправо, предложение 5) сдвигает вправо все содержимое операнда (в данном случае AX) на число битов, указанное с помощью регистра CL. "Выпавшие" с правого конца регистра биты теряются (строго говоря, последний выпавший бит сохраняется в флаге CF, но для данной программы это значения не имеет), а освобождающиеся биты с левой стороны операнда заполняются двоичными нулями. Сдвинув старший полубайт AX на 12 разрядов, мы переместили его на место младшего полубайта AL, что и требуется для подпрограммы hexasc. Поскольку по условиям вызова подпрограммы binasc регистр SI уже должен указывать на начало символьной строки, куда отправятся результаты преобразования, можно вызвать подпрограмму hexasc, которая заполнит первый байт выходной строки (предложение 6).

В предложениях 7...14 выполняется обработка следующей четверки битов. Сначала в регистр AX из стека выталкивается его исходное

содержимое, которое тут же снова сохраняется в стеке. Далее выделяется вторая слева четверка битов, которой соответствует операнд F00h команды and (предложение 9) и которая сдвигается уже не на 12, а на 8 разрядов (предложения 10-11). Результат трансляции этой 16-ричной цифры должен быть записан в следующий байт выходной строки, поэтому командой inc (increment, инкремент) указатель SI получает приращение 1. Вызовом процедуры hexasc завершается этот фрагмент программы.

В предложениях 15...21 аналогично выполняется обработка третьей слева четверки битов, в предложениях 22...26 - самой правой четверки. Далее восстанавливается содержимое регистров AX и CX (предложения 27-28), после чего командой ret управление возвращается в вызывающую программу.

В полях данных программы предусматривается та же таблица трансляции tblhex. Выходная строка result расширена, поскольку в нее записываются четыре 16-ричные цифры.

В главной процедуре main двухбайтовое исходное число (92C4h в примере) записывается в регистр AX, в регистр SI заносится адрес строки для результата преобразования и вызывается подпрограмма binasc, которая в свою очередь четырежды вызывает подпрограмму hexasc для преобразования каждой из четырех 16-ричных цифр исходного числа.

Статья 19

Дамп памяти и регистров

Разработанную нами программу вывода на экран символьных эквивалентов 16-ричных чисел можно использовать для получения динамического дампа интересующих нас ячеек памяти или регистров в процессе выполнения программы. Пусть, например, мы хотим узнать, где в памяти находится наша программа.

Любая программа, загруженная в память, включает три важных для программиста компонента: окружение, префикс программы PSP и собственно программу, которая (в случае программы типа .EXE) может состоять из нескольких сегментов. Поскольку окружение и сама программа (включая PSP) рассматриваются DOS, как отдельные блоки памяти, и та, и другая структура предваряются блоками управления памяти

(Memory Control Block, MCB) размером 16 байт. С помощью этих блоков DOS ведет учет свободной и занятой памяти (рис. 19.1).

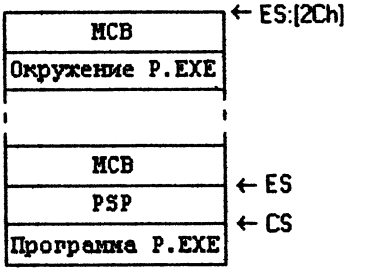


Рис.19.1. Память, занимаемая программой.

Окружение представляет собой область памяти, в которой в виде символьных строк записаны значения переменных, называемых переменными окружения, например, PROMPT= pg. Здесь PROMPT - переменная окружения, а pg - ее конкретное значение, которое может быть и другим.

Начальное окружение командного процессора, создаваемое в процессе загрузки DOS, чаще всего содержит переменные COMSPEC, PROMPT и PATH, которые заносятся в окружение из файла AUTOEXEC.BAT. Эти переменные используются командным процессо-ром при его работе. Пользователь может включить в окружение строки определения дополнительных переменных с помощью команды SET. Часто в качестве значений таких переменных указываются пути к каталогам с вспомогательными файлами или ключи, задающие режим работы программы.

При загрузке прикладной программы содержимое начального окружения копируется в создаваемое окружение прикладной программы, которая, таким образом, имеет доступ как к системным переменным (которые, скорее всего, ей не нужны), так и к переменным, включенным в окружение пользователем и адресованным именно ей.

Вопросы использования блоков управления памятью и окружения будут рассмотрены в дальнейших статьях. Здесь мы рассмотрим программу, которая после запуска выводит на экран терминала сегментные адреса окружения, префикса программного сегмента и сегмента команд.

После загрузки программы в память сегментные регистры ES и DS указывают на PSP, сегментный регистр CS - на сегмент команд, а в слове со смещением 2Ch от начала PSP содержится сегментный адрес окружения программы.

В примере 19.1 приведена только главная процедура main. Процедуры-подпрограммы hexasc и binasc, требуемые для работы программы, были рассмотрены в статьях 17 и 18.

Пример 19.1. Динамический вывод на экран характерных сегментных адресов.

```
main    proc
        mov     AX,data
        mov     DS,AX
;Выведем адрес окружения. Он находится по адресу 2Ch от начала PSP
        mov     AX,ES:[2Ch]
```

```
        mov    SI,offset envir+10
        call   binasc
        mov    AH,09h
        mov    DX,offset envir
        int    21h
;Выведем адрес PSP. Он находится в ES
        mov    AX,ES
        mov    SI,offset psp+10
        call   binasc
        mov    AH,09h
        mov    DX,offset psp
        int    21h
;Выведем адрес сегмента команд. Он находится в CS
        mov    AX,CS
        mov    SI,offset csseg+10
        call   binasc
        mov    AH,09h
        mov    DX,offset csseg
        int    21h
        mov    AX,4C00h   ;Функция DOS завершения программы
        int    21h        ;Вызов DOS
main    endp
;Поля данных
envir   db     'Окружение=****h',10,13,'$'
psp     db     'Префикс=  ****h',10,13,'$'
csseg   db     'Программа=****h',10,13,'$'
tblhex  db     '0123456789ABCDEF'
```

Для повышения наглядности вывода в программе предусмотрены три отдельные символьные строки для выводимых сегментных адресов. Каждая строка начинается с текста, поясняющего, какой именно адрес выводится в данной строке. Поля строк, предназначенные для хранения преобразованных адресов, начинаются с байта 10 от начала каждой строки.

Главная процедура состоит из трех практически одинаковых участков, в которых выполняется занесение в регистр AX преобразуемой величины, настройка регистра SI на начало поля для результата преобразования, вызов подпрограммы binasc и вывод с помощью функции DOS 09h всей строки на экран.

Обратите внимание на команду mov AX,ES:[2Ch]. В ней указано не символическое имя адресуемой ячейки (сегмент с PSP формирует система, и у его ячеек нет никаких имен), а известное нам числовое смещение 2Ch от начала PSP. Кроме того, явно указано, что адресоваться надо к сегменту, адрес которого находится в ES. Такой способ записи адреса, когда вместо подразумеваемого по умолчанию сегментного регистра DS явно указывается другой сегментный регистр, используется в практическом программировании очень часто и носит специальное название замены сегмента.

Статья 20

Основы организации подпрограмм

Подпрограммы, разработанные нами в предыдущих статьях, могут оказаться весьма полезным инструментальным средством для облегчения отладки программ и изучения их функционирования. Однако структура программного комплекса, в котором они использовались, не выдерживает никакой критики. Действительно, в единый файл с исходным текстом входят наравне и главная программа, и подпрограммы. Хотя поля данных и выделены в отдельный сегмент, однако в нем часть данных относятся к главной процедуре, часть же (конкретно, таблица трансляции) является необходимым элементом функционирования одной из подпрограмм. Для того, чтобы использовать процедуры hexasc и binasc для отладки какой-то другой программы, нам придется включить их исходный текст в исходный текст новой программы, а также дополнить сегмент данных программы описанием таблицы трансляции.

Обычно подпрограммы общего назначения оформляют в виде отдельных файлов (исходных модулей), транслируют независимо от основной программы, а затем полученные объектные модули подпрограмм присоединяют с помощью компоновщика к объектному модулю основной программы, в результате чего образуется единый выполнимый модуль, в состав которого входит и главная процедура, и процедуры-подпрограммы. Эта широко распространенная методика имеет целый ряд вариантов и требует специальных программных средств.

Начнем рассмотрение этих средств на примере простой подпрограммы, которая не требует передачи параметров и не работает с полями данных.

В примере 14.1 использовался распространенный прием остановки программы до нажатия произвольной клавиши. Оформим фрагмент остановки программы в виде отдельной процедуры-подпрограммы сначала в составе единого исходного модуля.

Пример 20.1. Подпрограмма без параметров в составе единого исходного модуля.

```
text    segment'code'    ;Начало общего сегмента команд .
        assume CS:text,DS:data
;Процедура-подпрограмма остановки выполнения до нажатия
произвольной клавиши
stop    proc
```

```
        mov     AH,08h      ;Функция ввода с клавиатуры
        int     21h         ;Вызов DOS
        ret                 ;Возврат в вызывающую программу
stop    endp
;Главная процедура
main    proc
        mov     AX,data     ;Инициализация сегментного
        mov     DS,AX       ;регистра DS
;Организуем бесконечный вывод на экран тестового сообщения
begin:  mov     AH,09h      ;Функция вывода
        mov     DX,offset message;Адрес сообщения
        int     21h         ;Вызов DOS
        call    stop        ;Вызов подпрограммы
        jmp     begin       ;Бесконечный цикл
main    endp
text    ends                ;Конец общего сегмента команд
data    segment             ;Начало сегмента данных
message db      '<> $'
data    ends                ;Конец сегмента данных
stk     segment             stack 'stack'
        dw      128 dup (0)
stk     ends
        end     main
```

В главной процедуре (которая в данном примере носит чисто демонстрационный характер) осуществляется непрерывный (в цикле) вывод на экран тестовой группы символов "<> ". В цикл включен вызов подпрограммы остановки, поэтому после вывода очередной тестовой группы программа останавливается до нажатия любой клавиши. Для завершения программы следует нажать <Ctrl>/<C> (в программе даже не предусмотрены стандартные строки завершения с помощью функции DOS 4Ch).

Отладив приведенную программу и убедившись, что она работает правильно, приступайте к модификации ее текста.

Пример 20.2. Основная программа и подпрограмма, оформленные в виде отдельных исходных модулей.

```
;Исходный текст основной программы (файл P.ASM)
text    segment public 'code'
        assume CS:text,DS:data
        extrn  stop:proc; Будет вызываться внешняя процедура stop
;Главная процедура
main    proc
        mov    AX,data
        mov    DS,AX
;Организуем вывод на экран строк текста
begin:  mov    AH,09h
        mov    DX,offset message
        int    21h
        call   stop        ;Вызов подпрограммы
        jmp    begin
main    endp
```

```
text    ends
data    segment
message db      '<> $'
data    ends
stk     segment stack 'stack'
        dw      128 dup (0)
stk     ends
        end     main        ;Оператор end с точкой входа

;Исходный текст подпрограммы stop (файл P1.ASM)
text    segment public 'code' ;Подпрограмма stop должна войти
                              ;в сегмент text
        assume CS:text        ;Регистр DS здесь не используется
        public stop           ;Процедура общего пользования
stop    proc
        mov    AH,08h
        int    21h
        ret                   ;Возврат в вызывающую программу
stop    endp
text    ends                  ;Конец сегмента команд
        end                   ;Оператор end без точки входа
```

Разделение элементов программного комплекса по отдельным исходным модулям с целью их последующего объединения компоновщиком потребовало внесения определенных изменений в тексты обеих процедур. Поскольку текст главной процедуры у нас невелик (заметно меньше 64К), разумно включить подпрограмму в тот же сегмент команд. При этом отдельные части одного сегмента описываются в разных исходных модулях, и в задачу компоновщика входит их слияние в один сегмент. Для того, чтобы компоновщик правильно выполнил это слияние, сегмент команд должен иметь описатель public (публичный, общего пользования). Это так называемый тип объединения сегмента. Тип public обозначает, что все сегменты этого типа с одним именем (text в нашем случае) будут сливаться последовательным подсоединением друг к другу (конкатенацией), причем, что очень важно в данном случае, адреса таких слившихся частей будут отсчитываться относительно самого начала получившегося объединенного сегмента. Без описателя public сегменты тоже объединятся, но адреса каждой части будут отсчитываться от ее начала, что приведет к дублированию адресов и неправильным обращениям из одной части сегмента в другую.

Второе новшество касается особых строк с объявлением подпрограммы как в главной (вызывающей) процедуре, так и в самой подпрограмме. Для того, чтобы транслятор не воспринял, как ошибку, ссылку в тексте главной процедуры на имя процедуры (stop) из другого модуля, это имя следует объявить внешним, для чего предусмотрен оператор extrn (external, внешний). В поле оператора extrn перечисляются все внешние имена, если их несколько, причем после каждого имени необходимо указать его тип. Поскольку stop - процедура, для нее указывается тип proc. Другие типы внешних имен будут описаны в дальнейшем.

Оператор extrn, как и многие другие директивы ассемблера, может стоять в любом месте текста основной программы, даже в другом сегменте или вообще за пределами сегментов.

В файле с исходным текстом подпрограммы ее имя должно быть объявлено с помощью директивы public. Такое объявление заставит транслятор занести информацию об этом имени в объектный файл. Это, в свою очередь, позволит компоновщику определить, в каком модуле содержится данная внешняя ссылка. Директива public, как и директива extrn, может стоять в любом месте текста подпрограммы.

Стоит заметить, что директива public, с помощью которой имена объектов (подпрограмм, меток, констант, переменных) объявляются именами общего пользования, не имеет прямого отношения к одноименному описателю public, с помощью которого определяется тип объединения конкретного сегмента.

Наконец, третья особенность многомодульных программ - объявление точки входа. Хотя все исходные модули должны заканчиваться директивой конца трансляции end, только в одном из них, содержащем главную процедуру, указывается точка входа - имя главной процедуры или входная метка. Остальные директивы end не имеют операндов.

Подготовьте файлы P.ASM и P1.ASM. Файлам можно дать любые имена и, возможно, разумнее было бы присвоить им имена MAIN.ASM и STOP.ASM, совпадающие с именами содержащихся в них процедур. Выбором имен P и P1 мы просто подчеркнули отсутствие связи между именами модуля и содержащейся в нем процедуры. Оттранслируйте оба модуля командами

```
MASM/ZI P,P,P;
MASM/ZI P1,P1,P1;
```

Образуются два объектных модуля P.OBJ и P1.OBJ. Компоновка их в единый загрузочный модуль с именем COMPLEX.EXE осуществляется командой

```
LINK/CO P.OBJ P1.OBJ,COMPLEX.EXE;
```

Имя результирующего модуля выбрано нами вполне произвольно. Можно было назначить ему, например, имя P.EXE, совпадающее с именем исходного файла с главной процедурой. Расширения OBJ и EXE можно не указывать, так как они действуют по умолчанию.

Запустите программу COMPLEX.EXE, убедитесь в правильности ее работы.

Имея объектный модуль с отлаженной подпрограммой, можно легко подсоединять ее на этапе компоновки к любым программам. Естественно, что в основной, вызывающей программе в нужных местах должны иметься строки вызова подпрограммы типа call stop.

Рассмотрим теперь вопрос о реальном расположении в памяти элементов нашего программного комплекса. Запустите программу COMPLEX.EXE под управлением отладчика и просмотрите выводимый на экран результат деассемблирования. На рис. 20.1 приведен возможный вариант отредактированного вывода отладчика с пояснениями (рисунок получен с помощью отладчика DEBUG).

```
                                        ┌── Начало сегмента команд
                                        │
0F8A:0000 B88D0F    MOV  AX,0F8D        mov AX,data
0F8A:0003 8ED8      MOV  DS,AX
0F8A:0005 B409      MOV  AH,09          begin: mov AH,09h
0F8A:0007 BA0000    MOV  DX,0000        mov DX,offset message
0F8A:000A CD21      INT  21
0F8A:000C E81100    CALL 0020           call stop
0F8A:000F EBF4      JMP  0005           jmp begin
0F8A:0011 0000      ADD  [BX+SI],AL  ┐
0F8A:0013 0000      ADD  [BX+SI],AL  │
0F8A:0015 0000      ADD  [BX+SI],AL  │
0F8A:0017 0000      ADD  [BX+SI],AL  ├── Нули в загрузочном модуле
0F8A:0019 0000      ADD  [BX+SI],AL  │
0F8A:001B 0000      ADD  [BX+SI],AL  │
0F8A:001D 0000      ADD  [BX+SI],AL  ┘
0F8A:001F 00B408CD  ADD  [SI+CD08],DH 00 mov AH,08h int
0F8A:0023 21C3      AND  BX,AX          21h ret
0F8A:0025 0000      ADD  [BX+SI],AL  ┐
0F8A:0027 0000      ADD  [BX+SI],AL  ├── Нули в загрузочном модуле
0F8A:0029 0000      ADD  [BX+SI],AL  │   до начала сегмента данных
0F8A:002B 0000      ADD  [BX+SI],AL  │     ┌── Начало сегмента данных
0F8A:002D 0000      ADD  [BX+SI],AL  ┘     │
0F8A:002F 003C      ADD  [SI],BH          00 '<'
0F8A:0031 3E        DS:                   '>'
0F8A:0032 2024      AND  [SI],AH          ' $'
0F8A:0034 0000      ADD  [BX+SI],AL  ┐
0F8A:0036 0000      ADD  [BX+SI],AL  │
0F8A:0038 0000      ADD  [BX+SI],AL  ├── Нули в загрузочном модуле
0F8A:003A 0000      ADD  [BX+SI],AL  │   до конца сегмента данных
0F8A:003C 0000      ADD  [BX+SI],AL  │
0F8A:003E 0000      ADD  [BX+SI],AL  ┘   Конец сегмента данных
```

Рис. 20.1. Результат деассемблирования программы COMPLEX.EXE.

Сегмент команд получил сегментный адрес 0F8A. Загрузочный модуль начинается с главной процедуры main не потому, что она главная, а лишь потому, что в строке вызова компоновщика модуль P.ASM с этой процедурой был первым в списке объектных модулей. При желании порядок процедур в загрузочном модуле можно изменить. Последняя команда процедуры main jmp begin занимает относительные адреса 000Fh и 0010h. Далее до адреса 001Fh загрузочный модуль заполнен нулями (которые отладчик пытается деассемблировать в команды add [BX+SI],AL), а с адреса 0020h начинаются команды подпрограммы. Подпрограмма заканчивается в байте с адресом 0024h (команда ret),

после чего опять идет последовательность нулей до адреса 002F. Начиная с относительного адреса 0030h отладчик выводит некоторые бессмысленные команды, хотя нетрудно догадаться (воспользовавшись таблицей кодов ASCII), что здесь располагаются данные программы, т.е. ее сегмент данных. Реально данные занимают всего 4 байта.

Разрыв в сегменте команд (15 байтов нулей) согласуется с командой call stop, которая выглядит в колонке вывода отладчика, как call 0020h. Процедура stop как раз и начинается с относительного адреса 0020h. Чем объясняется столь неэкономное расходование памяти?

Прежде всего заметим, что сегмент команд начинается на границе параграфа (по относительному адресу 0000h). Это естественно, так как физический адрес начала сегмента вычисляется процессором путем умножения сегментного адреса на 16, что выравнивает сегмент на границу 16-байтового участка, т.е. параграфа. По той же причине начинается на границе параграфа и сегмент данных. Отладчик не смог определить, где кончаются команды и начинаются данные, поэтому сегмент данных в выводе отладчика никак не выделен. Однако его начало (0F8Ah:0030h) действительно находится на границе параграфа и соответствует сегментному адресу 0F8Ah+3h=0F8Dh. Таким образом, участок нулей в загрузочном модуле по относительным адресам 0025h-002Fh вполне объясним.

Обратим теперь внимание на то, что процедура stop тоже начинается на границе параграфа (относительный адрес 0020h). Произошло так потому, что компоновщик, формируя загрузочный модуль, по умолчанию выровнял все встретившиеся ему куски сегментов на границу параграфа. На работоспособность получившейся программы это нисколько не влияет, однако с точки зрения расходования памяти программа оказывается построенной неоптимально. В случае одной подпрограммы это, конечно, не очень важно, но при наличии десятков или сотен коротких подпрограмм сумма пустых участков программы может оказаться весьма значительной.

Для того, чтобы исправить положение, следует указать компоновщику, что выравнивать отдельные участки сегментов надо не на параграф, а на слово или, еще лучше, на байт. Это делается путем указания еще одного описателя в предложении с оператором segment - типа выравнивания. Тип выравнивания может принимать значения byte (байт), word (слово), dword (double word, двойное слово), para (paragraph, параграф) и page (страница, 256 байт). Таким образом, для сокращения размера программы сегменты команд в процедурах следует описывать операторами такого вида:

```
text    segment 'code' byte
```

На рис. 20.2 приведен вывод отладчика для такой программы.

```
                                                  Начало сегмента команд
                                                  ┌
0F8A:0000 B88C0F   MOV   AX,0F8C     mov AX,data
0F8A:0003 8ED8     MOV   DS,AX
0F8A:0005 B409     MOV   AH,09       begin: mov AH,09h
0F8A:0007 BA0000   MOV   DX,0000     mov DX,offset message
0F8A:000A CD21     INT   21
0F8A:000C E80200   CALL  0011        call stop
0F8A:000F EBF4     JMP   0005        jmp begin
0F8A:0011 B408     MOV   AH,08
0F8A:0013 CD21     INT   21
0F8A:0015 C3       RET
0F8A:0016 0000     ADD   [BX+SI],AL  ┐
0F8A:0018 0000     ADD   [BX+SI],AL  │
0F8A:001A 0000     ADD   [BX+SI],AL  ├── Нули в загрузочном модуле
0F8A:001C 0000     ADD   [BX+SI],AL  │   до начала сегмента данных
0F8A:001E 0000     ADD   [BX+SI],AL  ┘
0F8A:0020 3C3E     CMP   AL,3E         '<>'
0F8A:0022 2024     AND   [SI],AH       ' $'
0F8A:0024 0000     ADD   [BX+SI],AL  ┐
0F8A:0026 0000     ADD   [BX+SI],AL  │
0F8A:0028 0000     ADD   [BX+SI],AL  ├── Нули в загрузочном модуле
0F8A:002A 0000     ADD   [BX+SI],AL  │   до конца сегмента данных
0F8A:002C 0000     ADD   [BX+SI],AL  │
0F8A:002E 0000     ADD   [BX+SI],AL  ┘   ──────Конец сегмента данных
```

Рис. 20.2. Результат деассемблирования программы COMPLEX.EXE при выравнивании сегмента команд на байт.

Из рисунка видно, что теперь обе процедуры располагаются в памяти вплотную друг к другу. Соответственно изменился и оператор вызова подпрограммы, он передает управление на байт с адресом 0011h вместо 0020h. Что же касается сегмента данных, то он попрежнему начинается на границе параграфа (относительный адрес 20h), так как по умолчанию для сегментов действует выравнивание на параграф. Указание для сегмента данных типа выравнивания byte ликвидирует и второй пустой участок в загрузочном модуле. Рекомендуем читателю задуматься над вопросом, как же сегмент данных может начаться не точно на границе параграфа? Видоизмените программу, объявив сегмент данных оператором

```
data    segment byte
```

Просмотрите программу в отладчике, обратив внимание на сегментный адрес сегмента данных и на относительный адрес находящихся в нем данных (поле с именем message).

Статья 21

Процедуры и поля данных

Вернемся к примеру 18.1, в котором был описан программный комплекс, обеспечивающий вывод на экран содержимого ячеек памяти или регистров. В отличие от примеров предыдущей статьи, где рассматривалась одна подпрограмма без параметров, в примере 18.1 было две подпрограммы, причем для работы одной из них требовалось специфическое поле данных (таблица трансляции). В таком более общем случае выделение подпрограмм из состава основной программы и оформление их в отдельные файлы требует решения ряда методических вопросов.

Если у нас имеются несколько подпрограмм общего назначения, их можно оформить тремя различными способами:

1) для каждой подпрограммы иметь отдельный файл с исходным текстом и, соответственно, отдельный объектный модуль. При этом подпрограммы могут образовывать отдельные сегменты команд или описываться в сегментах, одноименных с сегментом основной программы;

2) объединить все подпрограммы в один исходный файл и иметь, соответственно, один объектный модуль со всеми подпрограммами, которые могут при этом входить в единый сегмент команд, а могут составлять несколько сегментов. Последняя возможность важна в тех случаях, когда подпрограммы велики по размеру или их много и они не помещаются в один сегмент (64K);

3) объединить объектные модули всех подпрограмм в составе объектной библиотеки, которая создается специальной программой-библиотекарем.

Второй, прямо не связанный с количеством подпрограмм, вопрос касается полей данных, используемых подпрограммами. Если через поля данных подпрограмме передаются какие-то параметры из вызывающей программы, эти поля данных естественно в ней и расположить. Однако подпрограммы могут использовать в своей работе поля данных, которые вызывающей программе не нужны. Было бы естественно расположить данные такого рода где-то в тексте подпрограммы. Здесь также возможны два варианта:

1) данные располагаются в сегменте команд подпрограммы;

2) данные помещаются в сегмент данных, либо общий с вызывающей программой, либо специально созданный для обслуживания подпрограммы.

Рассмотрим сначала вопрос о размещении внутренних данных подпрограмм. В качестве рабочего материала мы используем программный комплекс, разработанный в статьях 18-19. Разнесем процедуры комплекса по отдельным исходным файлам: MAIN.ASM для главной процедуры main, HEX.ASM для процедуры hexasc и BIN.ASM для процедуры binasc (напомним, что имена файлам с процедурами можно назначать произвольно). При этом внутренние поля данных процедуры hexasc расположим в общем для всего комплекса сегменте команд text. Для облегчения участи читателя в примере 21.1 приведены полные тексты всех программ.

Пример 21.1. Основная программа и подпрограммы, оформленные в виде отдельных исходных модулей. Поля данных подпрограммы размещены в сегменте команд.

```
Исходный текст основной программы (файл MAIN.ASM)
text    segment'code' public byte
        assume CS:text, DS:data
        extrn  binasc:proc;Описание внешней процедуры,
                           ;вызываемой из данного файла
main    proc
        mov    AX,data
        mov    DS,AX
;Выведем адрес окружения. Он находится по адресу 2Ch от начала PSP
        mov    AX,ES:[2Ch]
        mov    SI,offset envir+10
        call   binasc
        mov    AH,09h
        mov    DX,offset envir
        int    21h
;Выведем адрес PSP. Он находится в ES
        mov    AX,ES
        mov    SI,offset psp+8
        call   binasc
        mov    AH,09h
        mov    DX,offset psp
        int    21h
;Выведем адрес сегмента команд. Он находится в CS
        mov    AX,CS
        mov    SI,offset csseg+10
        call   binasc
        mov    AH,09h
        mov    DX,offset csseg
        int    21h
;Завершим программу
        mov    AX,4C00h
        int    21h
main    endp
text    ends
data    segment
```

```
envir   db      'Окружение=****h',10,13,'$'
psp     db      'Префикс=****h',10,13,'$'
csseg   db      'Программа=****h',10,13,'$'
data    ends
stk     segment stack
        dw      128 dup (0)
stk     ends
        end     main        ;Конец текста программы с точкой входа
```

Исходный текст подпрограммы BINASC (файл BIN.ASM)

```
text    segment 'code' public byte
        assume CS:text
        public binasc       ;Объявление данной процедуры
                            ;процедурой общего пользования
        extrn  hexasc:proc;Описание внешней процедуры,
                            ;вызываемой в данном файле
binasc  proc
;Подпрограмма преобразования двоичного слова в 16-ричную
;символьную форму. При вызове: AX=преобразуемое число,
;SI=адрес строки, куда помещается результат преобразования
        push   CX
        push   AX
        and    AX,0F000h
        mov    CL,12
        shr    AX,CL
        call   hexasc
        pop    AX
        push   AX
        and    AX,0F00h
        mov    CL,8
        shr    AX,CL
        inc    SI
        call   hexasc
        pop    AX
        push   AX
        and    AX,0F0h
        mov    CL,4
        shr    AX,CL
        inc    SI
        call   hexasc
        pop    AX
        push   AX
        and    AX,0Fh
        inc    SI
        call   hexasc
        pop    AX
        pop    CX
        ret
binasc  endp
text    ends
        end    ;Конец текста программы без точки входа
```

Исходный текст подпрограммы HEXASC (файл HEX.ASM)

```
text    segment'code' public byte
        assume CS:text
        public hexasc       ;Объявление данной процедуры
                            ;процедурой общего пользования
```

```
hexasc  proc
;Подпрограмма получения символьного представления четырехбитового
;числа. На входе: Число в младшей половине AL, адрес строки, куда
;надо поместить результат, в SI
        push   BX          ;(1)Сохраним используемый регистр
        mov    BX,offset tblhex;(2)BX=адрес таблицы трансляции
        push   DS          ;(3)Сохраним наш DS
        push   CS          ;(4)Отправим адрес сегмента команд
        pop    DS          ;(5)в DS
        xlat               ;(6)Команда табличной трансляции
        pop    DS          ;(7)Восстановим DS
        mov    [SI],AL     ;(8)Вернем результат преобразования
        pop    BX          ;(9)Восстановим регистр
        ret                ;(10)Возврат в вызывающую программу
tblhex  db '0123456789ABCDEF';(11)Таблица трансляции
hexasc  endp
text    ends
        end;Конец текста программы без точки входа
```

Файл с исходным текстом основной программы назван MAIN.ASM. Из него удалены обе процедуры-подпрограммы, разнесенные по отдельным исходным файлам. В исходном тексте основной программы с помощью директивы extrn указано, что символическое обозначение binasc, встречающееся в тексте главной процедуры, является именем внешней процедуры. Поскольку вложенная процедура hexasc не вызывается из главной процедуры непосредственно, она в ней и не описана. Текст главной процедуры не претерпел никаких изменений, за исключением того, что в сегменте данных отсутствует таблица трансляции.

Файл с исходным текстом процедуры binasc назван BIN.ASM. В нем с помощью директивы public объявлено, что данная процедура является процедурой общего пользования, а с помощью директивы extrn описано внешнее имя hexasc. Оставшаяся часть программы не изменилась.

В файле HEX.ASM с подпрограммой hexasc объявлено, что это процедура общего пользования. Ее текст отличается от примера 18.1, так как изменилось местоположение таблицы трансляции tblhex: из сегмента данных она перекочевала в сегмент команд, что избавило нас от необходимости, составляя текст основной программы, заботиться о внутренних полях данных вызываемых процедур. Любые поля данных в составе сегмента команд должны располагаться таким образом, чтобы исключить их "выполнение", как программных строк. В нашем примере таблица tblhex расположена после команды ret. Можно было бы разместить ее и в начале программы, до оператора proc.

Как отразится на тексте программы размещение требуемых ей данных в сегменте команд? В нашем случае к этим данным обращается команда xlat, для правильного функционирования которой необходимо, чтобы адрес таблицы трансляции находился в регистрах DS:BX. Поэтому на время выполнения этой команды в регистр DS следует занести адрес того сегмента, в котором реально находится таблица трансляции,

т.е. сегмента команд. С этой целью в предложении 3 процедуры hexasc сохраняется текущее значение DS, который в настоящий момент указывает на сегмент данных основной программы, а затем в предложениях 4 и 5 содержимое CS загружается (через стек) в DS. После выполнения команды xlat немедленно восстанавливается старое содержимое регистра DS (предложение 7). Оставшаяся часть процедуры не изменилась.

Для подготовки загрузочного модуля описанного программного комплекса следует выполнить раздельную трансляцию всех трех исходных файлов и объединить процедуры на этапе компоновки:

```
MASM/ZI/Z/N MAIN,MAIN,MAIN;
MASM/ZI/Z/N BIN,BIN,BIN;
MASM/ZI/Z/N HEX,HEX,HEX;
LINK/CO MAIN BIN HEX,PRN_ADDR;
```

Здесь результирующему загрузочному модулю присваивается (произвольное) имя PRN_ADDR.EXE.

Рассмотренная методика размещения данных в сегменте команд используется очень широко, так как во многих случаях в программе, использующей небольшой объем локальных данных, нецелесообразно заводить собственный сегмент данных. Однако часто оказывается более удобным собрать все данные - и главной процедуры, и подпрограмм, в едином сегменте данных. Такой способ организации данных позволяет всем составляющим программного комплекса обращаться ко всем данным, и собственным, и "чужим". Если при этом отдельные поля данных присущи конкретным подпрограммам, целесообразно объявлять их не в основной программе, а в текстах подпрограмм. Тогда в случае компоновки программного комплекса в другом варианте (с другим составом подпрограмм) состав полей данных изменится автоматически.

Рассмотрим вариант организации того же программного комплекса, в котором подпрограмма hexasc имеет доступ к общему сегменту данных и может вносить в него свой вклад.

Пример 21.2. Основная программа и подпрограммы, оформленные в виде отдельных исходных модулей. Поля данных подпрограммы размещены в сегменте данных.

Исходный текст основной программы (файл MAIN.ASM)

```
text    segment 'code' public byte
        assume CS:text, DS:data
        extrn binasc:proc
main    proc
        ...
        call   binasc
        ...
main    endp
text    end
data    segment public
envir   db     'Окружение=****h',10,13,'$'
```

```
psp     db      'Префикс=****h',10,13,'$'
csseg   db      'Программа=****h',10,13,'$'
data    segment         public
stack   segment         stack
        ...
stack   ends
        end     main;Конец текста программы с указанием точки входа

Исходный текст подпрограммы BINASC (файл BIN.ASM)
text    segment 'code' public
        assume CS:text
        public binasc
        extrn  hexasc:proc
binasc  proc
        ...
        call   hexasc
        ...
        ret
binasc  endp
text    ends
        end;Конец текста программы без указания точки входа

Исходный текст подпрограммы HEXASC (файл HEX.ASM)
data    segment public
tblhex  db      '0123456789ABCDEF'
data    ends
text    segment 'code' public byte
        assume CS:text
        public hexasc
hexasc  proc
        push   BX          ;Сохраним используемый регистр
        mov    BX,offset tblhex;BX=адрес таблицы трансляции
        xlat               ;Команда табличной трансляции
        mov    [SI],AL     ;Вернем результат преобразования
        pop    BX          ;Восстановим регистр
        ret                ;Возврат в вызывающую программу
hexasc  endp
text    ends
        end;Конец текста программы без указания точки входа
```

В рассматриваемом варианте текст основной программы изменился лишь одной деталью: сегмент данных data объявлен с типом объединения public и с типом выравнивания byte. Указание типа объединения потребует от компоновщика сцепить вместе отдельные части сегмента данных, объявленные в разных исходных модулях. Указание типа выравнивания byte позволит получить оптимальную по расходованию памяти программу, в которой все составляющие сегментов примыкают друг к другу вплотную. При отсутствии описателя byte компоновщик позиционирует отдельные вхождения в сегмент данных по умолчанию на границу параграфа, что приведет к возникновению в загрузочном модуле пустых участков.

Подпрограмма binasc, которая не работает с полями данных, не претерпела никаких изменений.

В тексте подпрограммы hexasc появились строки объявления сегмента данных data, в котором размещена таблица трансляции tblhex. Существенно, что этому сегменту дано то же имя, что и у сегмента данных основной программы. Только в этом случае компоновщик присоединит поле данных tblhex к сегменту данных основной программы.

Процедура hexasc несколько упростилась. Поскольку к моменту ее вызова сегментный регистр DS заведомо указывает на общий для всего комплекса сегмент данных, отпала необходимость в настройке DS.

В результате мы получили программный комплекс, где каждая подпрограмма может иметь в общем сегменте данных собственные данные, которые описываются в ее тексте и о которых не нужно заботиться при составлении текста основной программы.

Статья 22

Библиотеки подпрограмм

Оформление каждой подпрограммы в виде отдельного модуля допустимо лишь в тех случаях, когда число подпрограмм невелико. При большом количестве подпрограмм, что типично для современного программного обеспечения, их лучше объединить в один файл. Как уже отмечалось в статье 20, это можно выполнить двумя способами, на уровне исходных текстов и на уровне объектных модулей.

Структура файла с двумя подпрограммами binasc и hexasc, взятыми из примера 21.2, приведена в примере 22.1. Исходные тексты подпрограмм просто следуют друг за другом. Если в какой-то подпрограмме описаны несколько сегментов, они могут располагаться в любом порядке. В конце всего текста помещается завершающая (и единственная) директива end без указания точки входа.

Пример 22.1. Объединение исходных текстов подпрограмм в один модуль.

```
text    segment 'code' public byte
        assume CS:text
        public binasc
binasc  proc
        ...
        ret
binasc  endp
hexasc  proc
```

```
        ...
        ret
hexasc  endp
text    ends
data    segment public byte
tblhex  db      '0123456789ABCDEF'
data    ends
        end     ;Завершающая директива end
```

По сравнению с примером 21.2 в тексты подпрограмм внесены некоторые "организационные" изменения. В описании процедуры binasc изъят оператор extrn, объявляющий подпрограмму hexasc внешней процедурой. Действительно, теперь эта подпрограмма расположена в том же модуле, что и вызывающая ее процедура binasc, и транслятору в процессе трансляции binasc не составит труда найти имя hexasc. По той же причине нет необходимости объявлять hexasc процедурой общего пользования (оператором public).

Файлу с подпрограммами дается произвольное имя (например, SUBS.ASM) и он транслируется обычным образом:

```
MASM/ZI SUBS,SUBS,SUBS;
```

Компоновка объектных файлов осуществляется так же, как и в случае нескольких файлов с подпрограммами. Если объектный файл основной программы имеет имя MAIN.OBJ, а результирующему загрузочному модулю решено дать имя PRN_ADDR.EXE, строка вызова компоновщика будет выглядеть так:

```
LINK/CO MAIN SUBS,PRN_ADDR;
```

При разработки сложных программных комплексов большие удобства предоставляет использование объектной библиотеки подпрограмм. Объектная библиотека создается специальной программой-библиотекарем (например, LIB.EXE фирмы Microsoft или TLIB.EXE фирмы Borland). Включение объектных файлов с подпрограммами во вновь создаваемую объектную библиотеку выполняется следующим образом (для библиотекаря LIB.EXE Microsoft):

```
LIB MYLIB.LIB +BIN.OBJ +HEX.OBJ, MYLIB.LST
```

В этой строке LIB - имя программы-библиотекаря, MYLIB.EXE - имя создаваемой объектной библиотеки, BIN.OBJ и HEX.OBJ - имена помещаемых в библиотеку объектных файлов, а MYLIB.LST - имя создаваемого файла с каталогом библиотеки. Знаки "+" обозначают, что указанные объектные файлы добавляются в библиотеку. 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 MYLIB +NEW1 +NEW2, MYLIB.LST, MYLIB
```

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

В процессе отладки программ часто возникает необходимость заменить некоторый объектный модуль в библиотеке на его новый вариант. Для этого надо сначала удалить из библиотеки старый вариант модуля, а затем добавить в нее новый вариант. Эти действия можно выполнить одной командой:

```
LIB MYLIB.LIB -BIN.OBJ +BIN.OBJ, MYLIB.LST, MYLIB.LIB
```

При наличии объектной библиотеки в команде вызова компоновщика следует указать имя библиотеки. Оно указывается в качестве четвертого параметра команды:

```
LINK/CO MAIN.OBJ, PRN_ADDR.EXE, PRN_ADDR.MAP, MYLIB.LIB
```

В приведенном примере MAIN.OBJ - объектный файл с основной программой, PRN_ADDR.EXE - образуемый компоновщиком выполнимый модуль, PRN_ADDR.MAP - листинг компоновки, а MYLIB.LIB - объектная библиотека со всеми подпрограммами, вызываемыми из основной программы и ее подпрограмм. Как и в предыдущих примерах, все расширения можно опустить, так они предполагаются или назначаются по умолчанию.

Статья 23

Вывод на экран текста средствами BIOS

В предыдущих статьях этой книги были описаны различные способы вывода на экран символьной (текстовой) информации. Для этого можно воспользоваться функциями DOS, например, функцией 09h, которая выводит текст до знака "$", или функцией 40h, для которой задается длина выводимой строки (или нескольких строк). Однако, воз-

можности DOS весьма ограничены: DOS не имеет функций для изменения цвета выводимых символов и позиционирования курсора. Поэтому текст, выводимый средствами DOS, представляет собой сплошную последовательность черно-белых строк, появляющихся друг за другом с нижнего края экрана. Кроме того, с помощью DOS можно выводить не все символы кодовой таблицы.

Для вывода цветных информационных кадров можно воспользоваться расширением системного драйвера консоли - устанавливаемым драйвером ANSI.SYS. Включение в выводимые на экран строки Esc-последовательностей позволяет очищать экран, изменять цвет выводимых символов, позиционировать курсор и выполнять другие полезные действия. При этом необходимо иметь в виду, что программы, использующие для вывода информации Esc-последовательности, при отсутствии в системе драйвера ANSI.SYS будут работать неправильно.

Все возможности видеосистемы компьютера можно реализовать с помощью функций прерывания 10h BIOS, которым мы уже пользовались в статье 14 для вывода на экран графических изображений. Программирование с помощью средств BIOS громоздко, однако большие возможности и высокая (по сравнению с функциями DOS) скорость вывода обуславливают широкое использование этого метода в прикладных программах. Стоит еще заметить, что средства DOS начинают функционировать только после полной загрузки операционной системы, а средства BIOS, записанные в ПЗУ, можно использовать даже в условиях отсутствия или неработоспособности DOS.

В BIOS имеется целый ряд функций для обслуживания текстового режима. Многие из них требуют задания атрибута выводимых символов. Атрибут символа занимает один байт и определяет цвет символа и фона под ним, а также некоторые дополнительные характеристики изображения на экране. Структура байта атрибутов приведена на рис. 23.1.

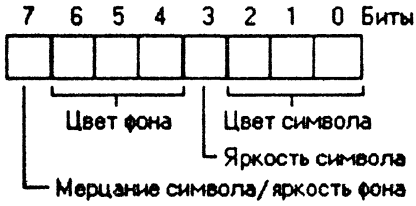


Рис.23.1. Структура байта атрибутов.

В биты 0...2 байта атрибутов записывается код цвета символа, а бит 3 при исходной настройке видеоадаптера управляет яркостью символа. Таким образом, каждый символ, независимо от других, может принимать любой из 16 возможных цветов. Коды атрибутов (совпадающие с кодами цветов при выводе графической информации) были приведены в табл. 14.1.

Биты 4...6 байта атрибутов задают цвет фона под данным символом. Что же касается последнего бита 7, то он, в зависимости от режима видеоадаптера, определяет либо яркость фона под данным символом (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 попрежнему определяется младшим полубайтом); значение Ch - не розовый, а красный фон при мерцающем символе и т.д.

Если видеоадаптер поставлен в режим мерцания 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.

В примере 23.1 показано использование некоторых наиболее употребительных функций BIOS для работы с экраном (прерывание 10h).

Пример 23.1. Вывод на экран символьной информации средствами BIOS

```
;Очистим экран, наложив на него черно-белое окно
        mov     AH,06h      ;Функция задания окна
        mov     AL,0        ;Режим создания (не прокрутки)
        mov     BH,07h      ;Атрибут всех символов в окне - ч/б
        mov     CH,0        ;Верхняя Y-координата
        mov     CL,0        ;Левая X-координата
        mov     DH,24       ;Нижняя Y-координата
        mov     DL,79       ;Правая X-координата
        int     10h         ;Прерывание BIOS
;Нарисуем на экране небольшое цветное окно
        mov     AH,06h      ;Функция задания окна
        mov     AL,0        ;Режим создания (не прокрутки)
        mov     BH,1Eh      ;Атрибут желтый по синему
        mov     CH,5        ;Верхняя Y-координата
        mov     CL,40       ;Левая X-координата
        mov     DH,9        ;Нижняя Y-координата
        mov     DL,75       ;Правая X-координата
        int     10h         ;Прерывание BIOS
;Позиционируем курсор
        mov     AH,02h      ;Функция позиционирования
        mov     BH,0        ;Видеостраница
        mov     DH,7        ;Строка
        mov     DL,45       ;Столбец
        int     10h         ;Прерывание BIOS
;Выведем в окно строку символов без атрибутов (т.е. с атрибутами
;окна)
        mov     CX,len1     ;Длина строки
        mov     BX,offset mes1;Адрес строки символов
        mov     AH,0Eh      ;Функция вывода одного символа
wr:     mov     AL,[BX]     ;Символ в AL
        inc     BX          ;Сдвинемся по строке
        int     10h         ;Прерывание BIOS
        loop    wr          ;Цикл по строке
;Выведем строку вне окна, задав атрибуты символов
        mov     AH,13h      ;Функция вывода строки
        mov     AL,0        ;Режим (атрибут в BL)
```

```
        mov     BH,0        ;Видеостраница
        mov     BL,04h      ;Атрибут всех символов
        mov     CX,len2     ;Длина строки
        mov     DH,16       ;Начальная позиция - строка
        mov     DL,25       ;Начальная позиция - столбец
        push    DS          ;Настроим
        pop     ES          ;ES на наш сегмент данных
        mov     BP,offset mes2;ES:BP ->выводимая строка
        int     10h         ;Прерывание BIOS
;Позиционируем курсор в начало последней строки экрана
        mov     AH,02h      ;Функция позиционирования
        mov     BH,0        ;Видеостраница
        mov     DH,24       ;Строка
        mov     DL,0        ;Столбец
        int     10h         ;Прерывание BIOS
;Поля данных
mes1    db      16,'Строка, выведенная в окно',17
len1=$-mes1
mes2    db      22,22,22,'Строка, выведенная вне окна',22,22,22
len2=$-mes2
```

Опишем кратко возможности использованных в примере функций BIOS.

С помощью функции 06h в заданном месте экрана дисплея создается цветное прямоугольное окно заданного размера. Если в созданные ранее окна выведен какой-либо текст, то с помощью этой же функции текст можно прокручивать вверх (функция 07h позволяет прокручивать цвет вниз). При этом текст, уходящий за край окна, пропадает, а из-под противоположного края появляются пустые строки с заданными атрибутами. Для заполнения появляющихся строк текстом следует использовать подходящие функции BIOS.

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 видеостраницу, причем функция 02h позиционирует курсор на любой заданной странице, независимо от того, какая страница является активной.

Для вывода на экран символов и символьных строк предусмотрены функции 09h, 0Ah, 0Eh и 13h. В примере 23.1 использованы функции 0Eh (вывод символа) и 13h (вывод строки).

Функция 0Eh фильтрует управляющие коды 07h (звуковой сигнал), 08h (возврат на шаг), 10h (перевод строки) и 13h (возврат каретки), выполняя соответствующие им действия. После вывода каждого символа курсор перемещается на следующую позицию, что дает возможность выводить целые строки (используя вызов функции в цикле). Однако атрибут символа установить нельзя, выводимый символ приобретает атрибут той позиции, куда он выводится.

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 после вывода 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 текста средствами 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.

Статья 24

Косвенные вызовы подпрограмм

До сих пор при вызове подпрограмм мы пользовались командой call с указанием в качестве ее операнда имени вызываемой подпрограммы. Такой вызов подпрограммы называется прямым; он нагляден, но не отличается гибкостью. Действительно, для того, чтобы той же строкой вызвать другую подпрограмму, необходимо изменить исходный текст и перетранслировать программу. Большей гибкостью обладают косвенные вызовы, в которых адрес перехода извлекается не из кода команды, а из

ячеек памяти или регистров; в коде команды содержится информация о том, где находится адрес перехода. Изменяя программным образом содержимое адресуемых командой call ячеек или регистров, т.е. засылая в них адрес той или иной подпрограммы, можно программно настроить команду call на вызов требуемой в настоящий момент подпрограммы.

В примере 24.1 дана иллюстрация косвенного вызова с использованием для адреса вызываемой подпрограммы ячейки памяти.

Пример 24.1. Косвенный вызов подпрограмм

```
main    proc
        ...
;Выведем с помощью функции DOS 09h сообщение prompt
        ...
;Поставим запрос на ввод символа
inpt:   mov   AH,01h     ;Функция ввода символа с эхом
        int   21h
        cmp   AL,'y'     ;Нажата клавиша 'y'?
        je    mode_dos   ;Да, работаем с DOS
        cmp   AL,'n'     ;Нажата клавиша 'n'?
        je    mode_bios  ;Да, работаем с BIOS
        jmp   inpt       ;Нажато не y/x, повторить ввод
mode_dos:mov  addr,offset dos;Зашлем адрес процедуры dos
        jmp   cont       ;и на продолжение
mode_bios:mov addr,offset bios;Зашлем адрес процедуры bios
cont:   call  DS:addr    ;Косвенный вызов процедуры
;Завершим программу
        ...
main    endp
dos     proc             ;Процедура вывода средствами DOS
        mov   AH,09h
        mov   DX,offset mes1
        int   21h
        ret
dos     endp
bios    proc             ;Процедура вывода средствами BIOS
;Очистим экран
        mov   AH,06h     ;Функция задания окна
        mov   AL,0       ;Режим создания окна
        mov   BH,07h     ;Атрибут всех символов в окне (ч/б)
        mov   CX,0       ;Координаты верхнего левого угла
        mov   DH,79      ;Нижняя Y-координата
        mov   DL,24      ;Правая X-координата
        int   10h        ;Прерывание BIOS
;Выведем строку
        mov   AH,13h     ;Функция вывода строки
        mov   AL,0       ;Режим 0 (атрибут в BL)
        mov   BH,0       ;Видеостраница
        mov   BL,0Eh     ;Атрибут всех символов
        mov   CX,len     ;Длина строки
        mov   DH,12      ;Начальная позиция - строка
        mov   DL,20      ;Начальная позиция - столбец
        push  DS         ;Настроим ES на наш
        pop   ES         ;сегмент данных
        mov   BP,offset mes2;ES:BP ->выводимая строка
```

```
        int     10h         ;Прерывание BIOS
;Позиционируем курсор в начало последней строки экрана
        mov     AH,02h      ;Функция позиционирования
        mov     BH,0        ;Видеостраница
        mov     DH,24       ;Строка
        mov     DL,0        ;Столбец
        int     10h         ;Прерывание BIOS
        ret                 ;Возврат из прерывания
bios    endp
;Поля данных
addr    dw      0 ;Поле для адреса подпрограммы
prompt  db      'Драйвер ANSI.SYS установлен? [Y/N] : $'
mes1    db      27,'[2J',27,'[12;20H',27,'[31;1m'
        db      'Начинаем работать, используя средства DOS'
        db      27,'[0m',27,'[25;1H$'
mes2    db      'Начинаем работать, используя средства BIOS'
len=$-mes2
```

В программе имеются главная процедура main и две процедуры-подпрограммы dos и bios. Обе выполняют очистку экрана и вывод в середину экрана "своей" цветной строки "Начинаем работать...". Процедура dos использует для очистки экрана, позиционирования курсора и задания цвета Esc-последовательности и, следовательно, может функционировать только при наличии в системе драйвера ANSI.SYS. Процедура bios выводит на экран то же самое, но средствами BIOS и не требует драйвера ANSI.SYS. В целях наглядности в процедурах для выводимых символов используются разные цвета.

Основная процедура выводит на экран вопрос о наличии в системе драйвера ANSI.SYS и вводит в программу ответ пользователя в виде символов y (yes, да) или n (no, нет). Далее введенный символ анализируется и, в зависимости от ответа пользователя, осуществляется переход на метки mode_dos или mode_bios. В предложениях с этими метками ячейка addr загружается адресом требуемой подпрограммы. После настройки ячейки addr выполняется команда call DS:addr, которая и осуществляет косвенный переход.

Указание перед именем ячейки памяти обозначения сегментного регистра, в данном случае DS, задает косвенность вызова. Другой способ задания того же - описатель word ptr (word pointer, указатель на слово):

```
        call    word ptr addr
```

Косвенные вызовы можно выполнять с помощью широкого набора способов адресации. Так, если адрес вызываемой подпрограммы занести в один из базовых или индексных регистров, то команда вызова упрощается:

```
        mov     BX,offset bios;В BX адрес самой подпрограммы
        call    BX          ;Косвенный вызов
```

Регистровая адресация возможна и в том случае, когда адрес подпрограммы находится в ячейке памяти:

```
mov    SI,offset addr;В регистре SI адрес ячейки
                     ;с адресом подпрограммы
call   [SI]          ;Косвенный вызов
```

Статья 25

Прерывания пользователя

Программы обработки прерываний (или попросту обработчики прерываний) относятся к важнейшим программным средствам персональных компьютеров. Запросы на обработку 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 цепями самого процессора, например, при попытке выполнения операции деления на ноль (такие прерывания называются внутренними, или отказами), а могут приходить из периферийного оборудования (внешние прерывания). Внешние аппаратные прерывания вызываются, например, сигналами микросхемы таймера, сигналами от принтера или контроллера диска, нажатием или отпусканием клавиши. Таким образом, можно говорить о прерываниях трех типов: внутренних, внешних и программных. Независимо от источника, действия процессора по обслуживанию поступившего прерывания всегда выполняются одинаково, как для аппаратных, так и для программных прерываний. Эти действия обычно называют процедурой прерывания. Подчеркнем, что здесь идет речь лишь о реакции самого процессора на сигнал прерывания, а не об алгоритмах обработки прерывания, предусматриваемых пользователем в обработчике прерываний.

Объекты вычислительной системы, принимающие участие в процедуре прерывания, и их взаимодействие показаны на рис. 25.1.

Самое начало оперативной памяти от адреса 0000h до 03FFh отводится под векторы прерываний - четырехбайтовые области, в которых хранятся адреса программ обработки прерываний (ПОП). В два старших

байта каждого вектора записывается сегментный адрес ПОП, в два младших - относительный адрес точки входа в ПОП в сегменте. Векторы, как и соответствующие им прерывания, имеют номера, причем вектор с номером 0 располагается, начиная с адреса 0, вектор 1 - с адреса 4, вектор 2 - с адреса 8 и т.д. Вектор с номером N занимает, таким образом, байты памяти от N*4 до N*4+3. Всего в выделенной под векторы области памяти помещается 256 векторов.

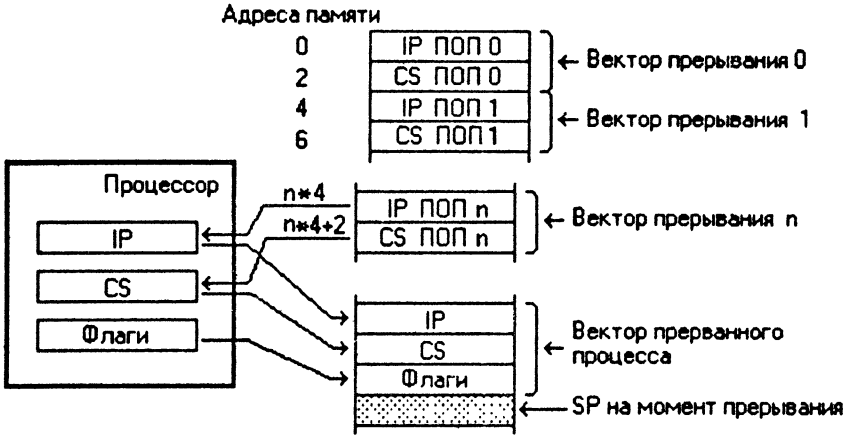


Рис. 25.1. Процедура прерывания.

Получив сигнал на выполнение процедуры прерывания с определенным номером, процессор сохраняет в стеке выполняемой программы текущее содержимое трех регистров процессора: регистра флагов, CS и IP. Два последних числа образуют полный адрес возврата в прерванную программу. Далее процессор загружает CS и IP из соответствующего вектора прерываний, осуществляя тем самым переход на ПОП.

Программа обработки прерывания обычно заканчивается командой возврата из прерывания iret (interrupt return, возврат из прерывания), выполняющей обратные действия - загрузку IP, CS и регистра флагов из стека, что приводит к возврату в основную программу в ту самую точку, где она была прервана.

Большая часть векторов прерываний предназначена для выполнения определенных действий и автоматически заполняется адресами системных программ при загрузке системы; часть векторов зарезервирована для будущих применений, а часть (конкретно с номерами 60h...66h) свободна и может использоваться в прикладных программах.

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. При вызове функции 25h в регистр AL помещается номер модифицируемого вектора, а в регистры DS:DX - полный двухсловный адрес нового обработчика.

Рассмотрим методику использования в прикладной программе прерывания пользователя.

Пример 25.1. Обработка прерываний пользователя

```
new_65h proc
;Процедура наложения на экран цветного окна для динамической
;очистки экрана по ходу выполнения программы
        mov     AH,06h      ;Фуикция задания окна
        mov     AL,0        ;Режим создания окна
        mov     BH,1Bh      ;Атрибут всех символов в окне:
                            ;Светло-бирюзовые символы, синий фон
        mov     CX,0        ;Координаты верхнего левого утла 0,0
        mov     DH,24       ;Нижняя Y-координата
        mov     DL,79       ;Правая X-координата
        int     10h         ;Прерывание BIOS
        iret
new_65h endp
main    proc
        mov     AX,data
        mov     DS,AX
;Заполним вектор прерывания пользователя адресом нашего
обработчика
        mov     AH,25h      ;Функция заполнения вектора прерывания
        mov     AL,65h      ;Номер вектора
        mov     DX,offset new_65h;Смещение прикладного обработчика
        push    DS          ;Сохраним DS
        push    CS          ;Настроим DS на сегмент команд (в
        pop     DS          ;котором находится наш обработчик)
        int     21h         ;Вызовем DOS
        pop     DS          ;Восстановим DS
;Будем в цикле выводить на экран строки с предварительной
;очисткой экрана
gogo:   int     65h         ;Вызов прикладного обработчика (очистка
                            ;экрана перед выводом текста)
;Позиционируем курсор
        mov     AH,02h      ;Функция позиционирования
        mov     BH,0        ;Видеостраница
        mov     DH,line     ;Строка
        mov     DL,coloumn ;Столбец
        int     10h         ;Прерывание BIOS
;Выведем на экран строку символов
        mov     AH,0Ah      ;Функция вывода символа без атрибута
        mov     AL,sym      ;Символ
        mov     BH,0        ;Видеостраница
        mov     CX,60       ;Коэффициент повторения
        int     10h         ;Прерывание BIOS
;Изменим символ и позицию и зациклим программу с возможностью ее
;завершения по нажатию клавиш <Ctrl>/C
        inc     sym         ;Следующий символ по таблице ASCII
```

```
        inc     line        ;Следующая строка экрана
        mov     AH,08h      ;Функция ввода без эха, чувствует
        int     21h         <Ctrl>/C
        jmp     gogo        ;Бесконечный цикл
main    endp
;Поля данных
line    db      2           ;Строка
coloumn db      10          ;Столбец
sym     db      01h         ;Выводимый символ
```

Процедура new_65h, вызываемая с помощью программного прерывания (для которого выбран вектор 65h), выполняет простую операцию - очищает экран, накладывая на него окно с заданным атрибутом.

В основной программе прежде всего заполняется вектор прерывания 65h. Поскольку функция заполнения вектора 25h требует, чтобы адрес прикладного обработчика содержался в паре регистров DS:DX, а DS у нас указывает на сегмент данных, перед вызовом DOS в DS следует занести сегментный адрес того сегмента, в котором находится обработчик, т.е., в нашем случае, общего сегмента команд. Этот адрес извлекается из CS.

Далее в бесконечном цикле выполняется вызов нашего обработчика, позиционирование курсора с помощью функции 02h BIOS и вывод на чистый экран строки символов (функцией 0Ah BIOS). Эта функция не позволяет задавать атрибуты выводимых символов. Символы приобретают атрибут тех позиций, куда они выводятся, т.е., в нашем случае, атрибут окна. После вывода на экран строки выполняется изменение кода символов и номера строки экрана, куда эти символы выводятся.

Функция DOS 08h (ввод символа без эха), включенная в цикл, выполняет две задачи. Во-первых, она останавливает выполнение программы и позволяет изучить содержимое экрана в каждом шаге цикла. Для того, чтобы продолжить выполнение программы, достаточно нажать на любую клавишу. Во-вторых, эта функция, будучи чувствительна к вводу с клавиатуры сочетания <Ctrl>/C, позволяет завершить программу, которая в противном случае выполнялась бы вечно.

Статья 26

Табличные вызовы подпрограмм

В тех случаях, когда программа с включенными в нее подпрограммами выполняется всегда по более или менее определенному алгоритму, вызывая подпрограммы в заданном заранее порядке, подпрограммы можно разместить в отдельных процедурах (или назначить им входные метки) и вызывать по именам. В более гибких программных комплексах бывает удобно предусмотреть механизм вызова требуемой подпрограммы по ее номеру. Такая ситуация типична, в частности, для программ DOS и BIOS. Действительно, для вызова требуемой функции DOS мы задаем (в регистре AH) ее номер и вызываем с помощью прерывания int 21h диспетчер DOS. Диспетчер извлекает из регистра AH номер функции и активизирует по этому номеру соответствующую программу из числа содержащихся в DOS. Так же вызываются и функции BIOS.

Рассмотрим упрощенную имитацию диспетчера DOS. Пусть мы имеем в программе группу подпрограмм и хотим вызывать их так же, как вызываем функции DOS - с помощью какого-либо прерывания пользователя, указывая в регистре AH номер "функции". Тогда в нашей программе, кроме собственно подпрограмм, должен быть еще и диспетчер, вызываемый с помощью прерывания и передающий далее управление на подпрограмму с заданным в регистре AH номером. Адрес этого диспетчера должен быть занесен в какой-либо из свободных векторов. Используем так же, как и в примере 25.1, вектор пользователя 65h.

Пример 26.1. Табличный вызов подпрограмм

```
;Диспетчер прерывания 65h
new_65h:push   BX          ;Для порядка сохраним
        push   DX          ;используемые регистры
        mov    BL,AH       ;"Функцию" отправим в BL
        mov    BH,0        ;Теперь номер функции в BX
        shl    BX,1        ;Умножим на два
        call   addr_tbl[BX];Вызовем подпрограмму
        pop    DX          ;Восстановим
        pop    BX          ;регистры
        iret               ;Возврат из прерывания
;Подпрограммы-"функции"
sub0:   mov    AH,9
        mov    DX,offset mes0
```

```
        int     21h
        ret
sub1:   mov     AH,9
        mov     DX,offset mes1
        int     21h
        ret
sub2:   mov     AH,9
        mov     DX,offset mes2
        int     21h
        ret
sub3:   mov     AH,9
        mov     DX,offset mes3
        int     21h
        ret
sub4:   mov     AH,9
        mov     DX,offset mes4
        int     21h
        ret
main    proc
;Используем прерывание пользователя 65h для вызова подпрограмм по
;номеру. Установим прикладной обработчик прерывания 65h
        mov     AX,2565h    ;Функция заполнения вектора (65h)
        mov     DX,offset new_65h
        push    CS          ;Настроим DS
        pop     DS          ;на сегмент команд
        int     21h
        mov     AX,data     ;Инициализация сегментного
        mov     DS,AX       ;регистра DS
;Вызовем последовательно наши подпрограммы
;Номер подпрограммы в регистре AH
        mov     AH,00h
        int     65h
        mov     AH,01h
        int     65h
        mov     AH,02h
        int     65h
        mov     AH,03h
        int     65h
        mov     AH,04h
        int     65h
;Завершим программу
        mov     AX,4C00h
        int     21h
main    endp
;Поля данных
;Таблица адресов подпрограмм
addr_tbl dw     sub0
        dw      sub1
        dw      sub2
        dw      sub3
        dw      sub4
;Выводимые сообщения
mes0 db 27,'[31mОтработала подпрограмма номер 0',10,13,27,'$'
mes1 db 27,'[32mОтработала подпрограмма номер 1',10,13,27,'$'
mes2 db 27,'[33mОтработала подпрограмма номер 2',10,13,27,'$'
mes3 db 27,'[34mОтработала подпрограмма номер 3',10,13,27,'$'
mes4 db 27,'[35mОтработала подпрограмма номер 4',10,13,27,'[0m$'
```

В диспетчере прерывания int 65h сохраняются используемые в нем и в подпрограммах регистры, после чего номер вызываемой "функции" пересылается из байтового регистра AH в словный регистр BX, который в дальнейшем будет использоваться в качестве индексного. Далее командой shl (shift left, арифметический сдвиг влево) содержимое BX сдвигается влево на 1 двоичный разряд, что эквивалентно умножению на два. Теперь содержимое BX можно рассматривать как индекс в таблице адресов addr_tbl, в которой записаны смещения (относительные адреса) подпрограмм в порядке, определяющем их номера.

Команда

```
call   addr_tbl[BX]
```

косвенного вызова передает управление на требуемую подпрограмму. После восстановления регистров диспетчер завершается командой iret, возвращая управление в основную процедуру.

Все использованные в примере подпрограммы практически одинаковы. Они выводят на экран некоторые сообщения с помощью функции DOS 09h. Для каждой подпрограммы предусмотрено собственное сообщение, в которые для красоты включены Esc-последовательности установки цвета.

Алгоритм основной процедуры main не содержит ничего нового. В ней устанавливается прикладной обработчик прерывания int 65h (можно было, разумеется, выбрать и другой вектор из диапазона 60h...66h), сегментный регистр DS настраивается на сегмент данных и, наконец, с помощью команд int 65h вызываются последовательно подпрограммы, номера которых указываются с помощью регистра AH.

Статья 27

Макрокоманды

Программы, написанные на языке ассемблера, часто содержат повторяющиеся участки текста с одинаковой структурой. Такие участки текста можно оформить в виде так называемых макроопределений (макрокоманд, макросов), характеризующихся произвольными именами и списками формальных параметров. После того, как макроопределение сделано, появление в программе строки, содержащей имя макроопреде-

ления и список фактических параметров, приводит к генерации всего требуемого текста, называемого макрорасширением. Варьируя фактические параметры, можно, сохраняя неизменной структуру макрорасширения, изменять отдельные его элементы.

Пример 27.1. Макрокоманда в составе программы

```
text    segment 'code'
        assume CS:text,DS:data
;Макроопределение макрокоманды write вывода строки на экран
write   macro  str
        mov    AH,09h
        mov    DX,offset str
        int    21h
        endm
main    proc
        ...
        write  string1
        write  string2
        ...
main    endp
;Поля данных
string1 db     'Вывод текста с помощью макрокоманды',10,13,'$'
string2 db     'Еще одна строка',10,13,'$'
```

Макроопределение начинается с директивы macro, перед которой указывается имя макрокоманды (write в нашем случае). После оператора macro можно указать одно или несколько произвольных имен, которые воспринимаются ассемблером, как имена формальных аргументов. В случае указания нескольких параметров они разделяются запятыми. Заканчивается макроопределение оператором endm. Расположение макроопределения не имеет значения, так как оно не транслируется, а только запоминается транслятором для последующего использования. Однако макроопределение должно предшествовать строкам его вызова.

Текст макроопределения пишется так же, как и текст любой программы. Формальные аргументы используются в макрокоманде так, как если бы это были обычные элементы программной строки. В нашем примере используется один параметр str, который служит для передачи в макрорасширение имени выводимой на экран строки. Вообще параметры макроопределения могут заменять собой любые элементы языка: обозначения регистров, команд, описателей, констант и т.д.

Хотя использование макрокоманд обычно упрощает программу и делает ее нагляднее, это относится только к исходному тексту программы. Объектный и загрузочный модули не изменяются. Этим макрокоманды отличаются от подпрограмм, которые позволяют сократить размер выполнимой программы за счет описания повторяющихся участков лишь однажды (в тексте подпрограммы). Однако макрокоманды предоставляют гораздо большие возможности изменения текста макрорасши-

рения в зависимости от значений фактических параметров. В целом можно сказать, что макрокоманды служат для упрощения процесса программирования, в то время как подпрограммы упрощают реализацию требуемого алгоритма.

В примере 27.1 текст макрокоопределения включен непосредственно в текст программы. 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. В примере 27.2 приведен текст файла макробиблиотеки с произвольным именем MAC.MAC, содержащей четыре макрокоманды write, outprog, delay и stop.

Пример 27.2. Содержимое файла MAC.MAC с макроопределениями.

```
write   macro str          ;Начало макроопределения write
        mov   AH,09h
        mov   DX,offset str
        int   21h
        endm
outprog macro              ;Начало макроопределения outprog
        mov   AX,4C00h
        int   21h
        endm
delay   macro time         ;Начало макроопределения delay
        local outer,inner
        push  CX
        mov   CX,time
outer:  push  CX
        mov   CX,65535
inner:  loop  inner
        pop   CX
        loop  outer
        pop   CX
        endm
stop    macro              ;Начало макроопределения stop
        push  AX
        mov   AH,08h
        int   21h
        pop   AX
        endm
```

Макрокоманда write, рассмотренная выше, осуществляет вывод на экран текстовой строки.

Макрокоманда outprog вызывает функцию DOS завершения программы 4Ch. Почти каждая программа должна заканчиваться вызовом этой функции, поэтому использование макрокоманды outprog при интенсивном программировании может привести к некоторой экономии времени.

Макрокоманда delay реализует задержку на заданное параметром time время. Конкретная величина задержки зависит от типа компьютера и должна быть определена экспериментально. В тексте макроопределения есть особенность, связанная с использованием циклов. Поскольку метки outer и inner не являются формальными параметрами, при повторном вызове этой макрокоманды они повторятся в тексте макрорасширения, что приведет к ошибке трансляции (множественное объявление имени). Для того, чтобы избавиться от множественного объявления, метки outer и inner объявлены с помощью директивы local локальными. В процессе макрорасширения локальные имена преобразуются в символические обозначения вида ??0001, ??0002 и т.д. и при многократных вызовах будут различаться. Эта макрокоманда написана несколько изящнее предыдущих в том отношении, что в ней предусмотрено сохранение (в начале) и восстановление (в конце) используемого в макрокоманде регистра CX. Таким образом, эту макрокоманду можно безбоязненно использовать в любых местах программы - ее вызов не приведет к разрушению регистров.

Макрокоманда stop служит для приостановки выполнения программы до нажатия любой клавиши. Для этого использована функция DOS 08h, которая ожидает ввода с клавиатуры символа, а получив символ, не отображает его на экране. Таким образом, макрокоманда не будет искажать текущее содержимое экрана. В ней так же сохраняется и восстанавливается используемый регистр AX.

Файл макробиблиотеки должен храниться в исходном виде, его не следует транслировать. В примере 27.3 приведена программа, использующая некоторые макрокоманды из файла MAC.MAC.

Пример 27.3. Вызов макрокоманд из макробиблиотеки.

```
text    segment 'code'
        include mac.mac  ;Сообщение транслятору имени файла
                         ;с макробиблиотекой
        assume CS:text, DS:data
main    proc
        mov    AX,data
        mov    DS,AX
        write  string1    ;Вызовы
        delay  200        ;макрокоманд
        write  string2    ;с конкретными
        delay  500        ;аргументами
        write  string3    ;
        outprog               ;
main    endp
text    ends   ;Конец сегмента команд
;Поля данных
string1 db     'Строка 1',10,13,'$'
string2 db     'Строка 2',10,13,'$'
string3 db     'Строка 3',10,13,'$'
```

Для того, чтобы транслятор подставил на место имен макрокоманд их макрорасширения, в тексте программы следует объявить имя макробиблиотеки с помощью директивы include. После этого в программе можно использовать любые макрокоманды из этой макробиблиотеки.

Статья 28

Структуры

Структуры представляют собой шаблоны с описаниями форматов данных, которые можно накладывать на различные участки памяти, чтобы затем обращаться к полям этих участков с помощью мнемонических имен, определенных в описании структуры. Структуры особенно удобны в тех случаях, когда мы обращаемся к областям памяти, не входящим в сегменты программы, т.е. к областям, поля которых нельзя описать с помощью символических имен. Рассмотрим в качестве примера задачу определения метки тома (дискеты или жесткого диска). Для решения этой задачи нам придется сначала познакомиться с рядом новых понятий.

Метка тома создается обычно с помощью команды DOS LABEL. Метка может иметь до 11 символов, разрешенных в именах файлов, т.е. большую часть знаков кодовой таблицы (включая русские буквы и псевдографические символы), хотя рекомендуется использовать в метке только цифры, латинские буквы и некоторые специальные знаки, например, -, _, ~, $, !, @, #, %.

Метка занимает один из блоков данных корневого каталога, предназначенных для записей о создаваемых в корневом каталоге файлах и подкаталогах и представляется системе как один из файлов этого каталога. Чтобы отличить метку тома от файла, ей присваивается атрибут 8; кроме того, в записи о метке в поля длины и начального кластера заносятся нули. В то же время метка, как и обычный файл или каталог, характеризуется датой и временем создания. Метка всегда создается в корневом каталоге; на каждом томе может быть только одна метка.

Для того, чтобы определить метку тома, надо просмотреть корневой каталог диска и попытаться найти в нем запись с атрибутом 8. Если такая запись имеется, это и есть запись о метке. Если такой записи нет, это значит, что данный диск не имеет метки.

Для поиска файлов на диске в DOS предусмотрены две функции - 4Eh (поиск первого файла) и 4Fh (поиск следующих файлов). Обычно эти функции используются для поиска всех файлов, соответствующих указанному шаблону групповой операции, например, всех программных файлов (шаблоны *.EXE или *.COM) или всех файлов с определенным именем и любыми расширениями (MYPROG.*). Кроме того, можно указывать атрибуты искомых файлов. При поиске группы файлов сначала вызывается функция поиска первого файла, для которой в качестве входных параметров указывается групповое имя искомых файлов и их атрибут. После нахождения первого файла из группы вызывается (в цикле) функция поиска следующих файлов; для нее входные параметры отсутствуют. Поиск следующих файлов ведется до тех пор, пока функция установкой флага CF не сообщит, что больше файлов нет.

Обе функции поиска, найдя первый или очередной файл, передают его характеристики (имя, размер, атрибуты, дату и время создания) в специальную системную область, которая называется дисковой областью передачи данных (disk transfer area, или сокращенно DTA). По умолчанию эта область располагается в префиксе программы (PSP) со смещением от его начала 80h. Если пользователя почему-то не устраивает такое расположение DTA, он может создать альтернативную DTA в своем сегменте данных, для чего предусмотрена функция DOS с номером 1Ah. Имеется также и возможность определения адреса текущей DTA (функция 2Fh).

Функции поиска файлов возвращают информацию о характеристиках найденных файлов в определенном формате. В таблице 28.1 приведены адреса полей данных DTA, заполняемых этими функциями.

Таблица 28.1. Поля данных DTA после заполнения их функциями 4Eh или 4Fh.

Смещение	Число байтов	Содержимое
00h	21	Недокументированная область
15h	1	Атрибуты файла или каталога
16h	2	Время создания файла
18h	2	Дата создания файла
1Ah	4	Логический размер файла в байтах
1Eh	13	Имя и расширение файла в формате ASCIIZ

Имя и расширение файла записываются в DTA прописными буквами и разделяются точкой (отсутствующей в записи каталога). Спецификация завершается байтом с двоичным нулем. В отличие от формата каталога, если в имени меньше 8 символов, оно не дополняется пробелами. Таким образом, спецификация может иметь длину от одного байта в позиции 1Ah (плюс нуль в позиции 1Bh) до 13 символов (8 символов в

имени, точка, 3 символа в расширении и нуль), занимающих все отведенное для спецификации файла место от 1Eh до 2Ah.

Форматы записи даты и времени создания файла (рис. 28.1) совпадают с соответствующими форматами каталога.

Рис. 28.1. Форматы записи даты и времени создания файла в каталоге и в дисковой области пере-

Метка записывается в поле DTA точно так же, как и имя файла. Поэтому если в метке больше 8 символов, то между первыми 8-ю и последними 3-мя символами метки стоит точка (код 2Dh). Это необходимо иметь в виду в случае программного анализа найденной метки тома.

Пример 28.1. Структуры. Чтение метки тома

```
;Определение структуры
dta       struc               ;(1)Начало определения
wild_card db   15h dup (?);(2)Зарезервированы
attrib    db   ?              ;(3)Атрибуты файла
file_time dw   ?              ;(4)Время создания
file_date dw   ?              ;(5)Дата создания
file_size dw   2 dup (?)      ;(6)Размер файла
file_name db   12 dup(?)      ;(7)Имя и расширение
dta       ends                ;(8)Конец определения
main      proc                ;(9)Начало главной процедуры
          mov    AX,data      ;(10)Инициализация сегментного
          mov    DS,AX        ;(11)регистра DS
;Найдем метку тома
          mov    AH,4Eh       ;(12)Функция поиска первого файла
          mov    CX,8         ;(13)Атрибут метки тома
          mov    DX,offset dname;(14)Спецификация искомых файлов
          int    21h          ;(15)
          jc     nolabel      ;(16)Метка отсутствует
;Настроимся на DTA (ES после загрузки программы указывает на PSP)
          mov    BX,80h       ;(17)ES:BX=адрес DTA
;Настроим регистры для команд работы со строками
          push   DS           ;(18)Обменяем содержимое
          push   ES           ;(19)регистров DS и ES, чтобы
          pop    DS           ;(20)DS указывал на PSP, а ES
          pop    ES           ;(21)на сегмент данных программы
          mov    DI,offset lbl1 ;(22)ES:DI -> поле для имени файла
                              ;(метки тома) в программе
          lea    SI,[BX].file_name;(23)DS:SI -> имя метки в DTA
          cld                 ;(24)Движение вперед по строке
;Будем перемещать метку из DTA в программу посимвольно до нуля
movlbl:   lodsb               ;(25)Очередной символ метки в AL
          cmp    AL,0         ;(26)Метка кончилась?
```

```
        je      go1             ;(27)Да, на выход из цикла
        stosb                   ;(28)Нет, отправим символ в программу
        jmp     movlbl          ;(29)На повторение
go1:    mov     AX,[BX].file_date;(30)Получим дату создания
        push    ES              ;(31)Восстановим адресуемость данных
        pop     DS              ;(32)через сегментный регистр DS
        push    AX              ;(33)Сохраним дату
;Преобразуем день
        and     AX,1Fh          ;(34)Оставим в AX только биты дня
        mov     DL,10           ;(35)Поделим на 10. AL=частное (число
        div     DL              ;(36)десятков дней), AH=остаток(число
                                ;дней)
        add     AX,'00'         ;(37)И то, и другое преобразуем в
                                ;символы
        mov     lbl3+1,AH       ;(38)Отправим в программу
        mov     lbl3+0,AL       ;(39)Поместим в сообщение день месяца
;Преобразуем месяц
        pop     AX              ;(40)Восстановим дату в AX
        push    AX              ;(41)И снова сохраним
        and     AX,1E0h         ;(42)Оставим в AX только биты месяца
        mov     CL,5            ;(43)Будет сдвиг на 5 бит
        shr     AX,CL           ;(44)Сдвинем вправо к началу регистра
        mov     AH,AL           ;(45)Скопируем номер месяца в AH
        add     AL,AH           ;(46)Прибавим его же
        add     AL,AH           ;(47)И еще раз, т.е. умножим на 3
        sub     AL,3            ;(48)Скорректируем
        cbw                     ;(49)Преобразуем в слово
        mov     DI,offset lbl3+3;(50)ES:DI -> поле для имени месяца
        mov     SI,offset months;(51)DS:SI -> таблица имен месяцев
        add     SI,AX           ;(52)Прибавим номер месяца*3
        mov     CX,3            ;(53)Будем перемещать 3 символа
rep     movsb                   ;(54)Поместим в сообщение месяц
        pop     AX              ;(55)Еще раз восстановим дату в AX
        and     AX,0FE00h       ;(56)Оставим в AX только биты года
        mov     CL,9            ;(57)Будет сдвиг на 9 бит
        shr     AX,CL           ;(58)Сдвинем вправо к началу регистра
        add     AX,80           ;(59)В дате год от 1980. Получим
                                ;правильный год (от 1900)
        div     DL              ;(60)Поделим на 10. AL=число десятков
                                ;лет, AH=остаток (число единиц лет)
        add     AX,'00'         ;(61)И то, и другое преобразуем в
                                ;символы
        mov     lbl3+10,AH      ;(62)Отправим в программу
        mov     lbl3+9,AL       ;(63)Поместим в сообщение год
;Отредактируем метку - уберем точку перед последними тремя
;символами
        mov     SI,offset lbl1+9;(64)DS:SI->символ за точкой
        mov     DI,offset lbl1+8;(65)ES:DI->точка
        mov     CX,3            ;(66)Перемещать 3 символа расширения
rep     movsb                   ;(67)Переместим
        mov     lbl1+11,' ';(68)Затрем пробелом последний символ
        mov     BP,offset lbl   ;(69)BP->сообщение о найденной метке
        mov     CX,lbl_len      ;(70)Длина сообщения
        jmp     outmes          ;(71)На вывод сообщения на экран
;Обработка ошибки - на дискете нет метки
nolabel:mov     BP,offset mes;(72)BP->сообщение об отсутствии метки
        mov     CX,mes_len      ;(73)Длина этого сообщения
```

```
;Выведем сообщение о найденной метке тома или об ее отсутствии
outmes: mov    AH,40h      ;(74)Функция вывода на экран
        mov    BX,1        ;(75)Дескриптор стандартного вывода
        mov    DX,BP       ;(76)BP->сообщение о метке
        int    21h         ;(77)
        ...                ;Завершим программу
;Поля данных
dname   db     'A:\*.*',0 ;Просмотрим все имена в A:\
mes     db     10,13,'На дискете нет метки!',10,13
mes_len=$-mes
lbl     db     10,13,'Метка A: '
lbl1    db     12 dup (' '),10,13
lbl2    db     'Дата создания: '
lbl3    db     '       19  ',10,13
lbl_len=$-lbl
months  db     'янвфевмарапрмайиюниюлавгсеноктноядек'
```

Программа начинается с определения структуры dta, в которой описаны поля области передачи данных (см. рис. 28.1). Наложение структуры на реальную область передачи данных позволит использовать в программе вместо числовых смещений к соответствующим полям более наглядные мнемонические обозначения (например, file_name - имя файла); кроме того, мы избавляемся от необходимости использовать описатели word ptr и byte ptr, которые обычно требуются при обращении к "безымянным" участкам памяти.

С помощью функции 4Eh поиска первого файла в корневом каталоге диска A: ищется первый (и, естественно, единственный) файл с атрибутом 8 - метка тома. Если DOS не нашла такого файла, перед возвратом из функции 4Eh устанавливается флаг CF. В этом случае команда jc (jump if carry, переход, если перенос) предложения 16 передает управление на вывод сообщения об отсутствии на дискете метки тома. Если метка найдена, ее имя и характеристики помещаются системой в текущую DTA.

В регистр BX заносится смещение DTA в сегменте PSP (число 80h, предложение 17). Теперь в ES:BX находится полный двухсловный адрес текущей DTA.

Вывести метку на экран можно непосредственно из DTA, однако удобнее перенести имя метки из DTA в область данных программы, где к имени можно прибавить дополнительный текст.

Для обработки цепочек (строк) байтов или слов в процессоре предусмотрена специальная группа команд. Одну из таких команд, команду сравнения байтов строк cmpsb, мы уже рассматривали в статье 12. В примере настоящей статьи используются еще две команды той же группы: stosb (store string byte, сохранение байта строки) и movsb (move string byte, пересылка байта строки). Как и команда cmpsb, они имеют разновидности для обработки слов (stosw и movsw). Все команды обработки

строк имеют ряд общих черт. Операнд-источник адресуется ими через регистры DS:SI, а операнд-приемник - через регистры ES:DI. Команды выполняют операцию над единичным байтом (или словом); если требуется выполнить операцию над последовательностью элементов строки, перед командой следует поставить один из префиксов повторения и поместить в регистр CX требуемое число повторений. После каждого единичного выполнения команд указатели SI и DI получают положительное или отрицательное приращение в зависимости от состояния флага направления DF.

К настоящему моменту регистр DS настроен на сегмент данных программы, а регистр ES на сегмент с областью передачи данных (фактически - на PSP). Для перемещения имени метки из DTA в программу с помощью команд работы со строками содержимое DS и ES должно быть обратным. Обмен содержимого DS и ES осуществляется наиболее экономным способом - через стек (предложения 18...21). В регистр DI помещается смещение поля в программе для имени метки.

В предложении 23 использована команда lea (load effective address, загрузка эффективного адреса). Эта команда загружает в регистр, указанный в качестве первого операнда, относительный адрес второго операнда (не значение операнда!). Команда lea фактически эквивалентна команде mov reg, offset mem, однако имеет больше возможностей описания адреса интересующей нас ячейки и особенно удобна в тех случаях, когда требуется определить адрес ячейки, не имеющей мнемонического имени. В нашем случае команда

```
lea    SI,[BX].file_name
```

загружает в регистр SI смещение к имени файла в DTA. Для указания смещения используется мнемоническое обозначение из структуры dta. При этом базовый адрес DTA уже находится в регистре BX, что и дает нам право использовать его в качестве базового. Точка перед мнемоническим обозначением смещения file_name указывает на использование обозначения из определения структуры.

Перемещение имени метки из DTA в программу затруднено тем, что неизвестна длина метки. Если переносить в программу всегда 12 байт, то в случае короткой метки в программу может скопироваться мусор. Поэтому метка переносится посимвольно с проверкой каждого символа на 0. Команда lodsb (предложение 25) загружает очередной символ из строки-источника в регистр AL с автоматическим инкрементом индексного регистра SI; команда stosb (предложение 28) помещает содержимое регистра AL в строку-приемник с автоматическим инкрементом индексного регистра DI. Как только в регистр AL попадет байт с двоичным нулем, завершающий имя файла в DTA, командой je go1 осуществляется выход из цикла на продолжение программы.

В предложении 30 дата создания файла загружается из DTA в регистр AX. Здесь опять использовано мнемоническое обозначение из структуры dta. При отсутствии определения структуры нам пришлось бы написать

```
mov   AX,word ptr [BX]+18h
```

(см. табл. 28.1), что, конечно, гораздо менее наглядно. Между прочим, еще одно преимущество использования структур заключается в том, что при изменении расположения данных, на которые накладывается структура, и, соответственно, их смещений, достаточно внести коррективы в определения структуры и перетранслировать программу. Если же при программировании использовать смещения в численной форме, то придется выискивать в программе и корректировать все ссылки на измененные поля. Другой способ избавиться от этой работы - дать числовым значениям смещений мнемонические имена, например, в начале программы определить константу file_date=18h и затем использовать не численное, а мнемоническое значение смещения.

В предложениях 31-32 выполняется восстановление адресуемости данных через регистр DS. Далее дата создания файла сохраняется в стеке. С помощью маски 1Fh и команды and в регистре AX выделяется день месяца. Для преобразования в символьную форму номер дня делится на 10, к обоим результатам (частному и остатку) добавляются коды символа '0' и результат заносится в выводимую на экран строку. Деление осуществляется байтовым вариантом команды div (division, деление). Эта команда делит содержимое регистра AX на указанный в команде операнд (в нашем случае на содержимое DL). Частное помещается в регистр AL, остаток - в регистр AH.

В предложении 40 из стека извлекается сохраненное там исходное значение даты (и сразу же сохраняется в стеке снова), и из него маской выделяется номер месяца, который с помощью команды логического сдвига вправо сдвигается к началу регистра AX. Для преобразования номера месяца в его символьное трехбуквенное обозначение номер месяца умножается на 3 (предложения 45...47) и из результата вычитается 3 (так как номера месяцев начинаются с 1, а смещения в таблице имен месяцев - с 0). Вычитание осуществляется командой sub (subtraction, вычитание), которая вычитает из левого операнда правый и записывает разность на место левого операнда. По ходу умножения на 3 мы "засорили" регистр AH. Для его очистки можно было воспользоваться командой mov AH,0 однако более эффективно это выполнить с помощью команды cbw (convert byte to word, преобразование байта в слово). Эта команда 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. После настройки индексных регистров DI и SI и установки

счетчика числа переносимых байтов командой movsb с префиксом rep (предложение 54) имя месяца переносится в выходную строку.

Обработка года осуществляется аналогично обработке дня - маскированием, сдвигом, делением на 10 и преобразованием в символьную форму. Перед делением на 10 к значению кода из DTA добавляется 80, так как в записи каталога и в DTA указывается не сам год, а число лет, прошедших от 1980 года.

Последняя операция по обработке метки - удаление из нее точки, которая естественна в имени обычного файла, но выглядит странно в метке. Удаление точки выполняется сдвигом с помощью команды movsb (предложение 67) трех последних символов на один байт влево и затиранием затем последнего символа символом пробела.

Приведенная программа замечательна тем, что в отличие от команд DOS LABEL, VOL и DIR, она выводит на экран не только значение метки тома, но и дату ее создания.

Статья 29

Записи

Записи, как и структуры, представляют собой шаблоны, накладываемые на реальные данные с целью введения удобных мнемонических обозначений отдельных элементов данных. В отличие от структур, дающих имена байтам, словам или массивам, в записях определяются строки битов внутри байтов, слов или двойных слов.

Вспомним (см. рис. 28.1), что дата записывается в элементе каталога в 16-разрядном слове, причем старшие 7 бит этого слова обозначают год, следующие 4 бита - месяц и последние 5 бит - день. Эти данные удобно описать с помощью записи date, определяемой в программе следующим образом:

```
date    record year:7, month:4, day:5
```

Ключевое слово record говорит о том, что имя date относится к записи, а мнемонические обозначения year, month и day являются произвольными именами отдельных битовых полей описываемого слова.

Включение в программу описания шаблона битовых полей позволяет отказаться от утомительного определения "вручную" смещений кон-

кретных полей внутри слова, а также значений масок, требуемых для выделения отдельных полей. Для этого используются операторы mask и width.

Оператор mask возвращает битовую маску, в которой биты, соответствующие данному полю, равны 1, а все остальные биты равны 0. Так, выражение

```
mask  day
```

эквивалентно двоичному числу 0000 0000 0001 1111 (=1Fh); выражение

```
mask  month
```

эквивалентно двоичному числу 0000 0001 1110 0000 (1E0h).

Оператор width возвращает ширину (в битах) указанного поля записи. Так, значение выражения

```
width day
```

равно 5; значение выражения

```
width month + width day
```

равно сумме ширин полей дня и месяца и составляет 9.

Введение в программу 28.1 описания записи date позволяет сделать строки выделения требуемых полей и сдвигов более наглядными (и, возможно, избежать ошибок "ручного" определения требуемых значений масок и ширин полей). Предлагаемые изменения сведены в табл. 29.1.

Таблица 29.1. Замена числовых аргументов при использовании записи

Старая редакция	Новая редакция
`and AX,1Fh`	`and AX,mask day`
`and AX,1E0h`	`and AX,mask month`
`mov CL,5`	`mov CL,width day`
`and AX,0FE00h`	`and AX,mask year`
`mov CL,9`	`mov CL,width month+width day`

Статья 30

Способы адресации и оптимизация программ

Изучив приведенные в этой книге программы, вы, вероятно, уже обратили внимание на то, что обращение к данным в них выполняется по-разному. Обрабатываемые в программе данные можно помещать непосредственно в регистр, записывать в качестве одного из операндов прямо в код команды, либо тем или иным способом указывать место в памяти, где эти данные расположены. Рассмотрим, например, программу, обеспечивающую поиск максимального элемента в массиве данных.

Пример 30.1. Поиск максимального значения

```
;Сегмент данных
mas     db 1,2,5,3,7,9,8,3,4;Массив с исходными данными
num=$-mas                   ;Число байтов (элементов) в массиве
;Сегмент команд
        mov     DL,mas      ;(1)DL=Первый элемент массива
        mov     CX,num-1    ;(2)CX=число сравниваемых элементов
                            ;минус один (первый)
        mov     BX,1        ;(3)BX=индекс второго элемента
cont:   cmp     DL,mas[BX]  ;(4)DL>следующего элемента?
        jg      next        ;(5)Да, на анализ следующего
        mov     DL,mas[BX]  ;(6)Нет, в DL заносим следующий элемент
next:   inc     BX          ;(7)И на анализ следующего
        loop    cont        ;(8)Цикл CX раз
```

В предложении 1 в регистр DL заносится содержимое первого байта последовательности данных, которая в этой программе обозначена как mas. Здесь мы сразу же встречаемся с двумя способами адресации данных - регистровым и прямым (прямая адресация к памяти).

При использовании регистровой адресации данные записываются в регистр - однобайтовый или двухбайтовый.

Прямой способ адресации является представителем группы способов обращения к данным, которые хранятся не в регистрах, а в ячейках памяти. В рассматриваемом случае при прямой адресации имя mas фактически является адресом памяти, то есть смещением в сегменте данных, начиная с которого хранятся данные. При этом в качестве сегментного регистра по умолчанию используется регистр DS. Содержимое указанной ячейки памяти переносится (в данном случае) в регистр DL. Если рассматриваемая ячейка памяти располагается в сегменте, базовый

адрес которого находится в другом сегментном регистр (ES, CS или SS), то обязательно указание этого регистра: mov DL,ES:mas.

Еще один способ адресации представлен в предложении 3, где в качестве одного из операндов указано число (конкретно 1). Такой операнд входит непосредственно в состав команды процессора. Этот способ адресации так и называется - непосредственный.

В предложениях 4 и 5 применена разновидность прямой адресации к памяти, в которой указывается не только обозначение ячейки памяти (mas), но и величина сдвига относительно этой ячейки (в данном случае в регистре BX). Очевидно, что такой способ адресации удобен при работе с массивами.

Все имеющиеся способы адресации можно условно разделить на три группы: непосредственный, регистровый и с указанием адреса памяти. При этом адрес памяти можно задавать по-разному: прямым указанием символического обозначения ячейки памяти, указанием регистра, в котором хранится требуемый адрес, или и того, и другого. Таким образом, третья группа включает, в сущности, целый ряд способов адресации. Они обычно носят названия: прямая, базовая, индексная, базово-индексная, а также базовая, индексная или базово-индексная со смещением.

Приведем краткий обзор способов адресации.

Регистровая адресация

Операнд (байт или слово) находится в регистре. Способ применим ко всем программно-адресуемым регистрам процессора.

Примеры

```
        push    DS          ;Сохранение DS в стеке
        mov     BP,SP       ;Пересылка содержимого SP в BP
```

Непосредственная адресация

Операнд (байт или) может быть представлен в виде числа, адреса, кода ASCII, а так же иметь символьное обозначение.

Примеры

```
        mov     AX,4C00h    ;Операнд - 16-ричное число
        mov     DX,offset mas;Смещение массива mas заносится в DX
        mov     DL,'!'      ;Операнд - код ASCII символа "!"
num=9                       ;Число 9 получает обозначение num
        mov     CX,num      ;Число, обозначенное num,
                            ;загружается в CX
```

Прямая адресация к памяти

В команде указывается символическое обозначение ячейки памяти, над содержимым которой требуется выполнить операцию.

Примеры

```
        mov     DL,meml     ;Содержимое байта памяти с символическим
                            ;именем meml пересылается в DL
```

Если нужно обратиться к ячейке памяти с известным абсолютным адресом, то этот адрес можно непосредственно указать в качестве операнда. Предварительно необходимо настроить какой-либо сегментный регистр на начало того участка памяти, в котором находится искомая ячейка

Пример

```
        mov     AX,0            ;Настроим сегментный регистр на
        mov     ES,AX           ;самое начало памяти (адрес 0)
        mov     AX,ES:[0]       ;Заберем в AX содержимое слова
                                ;памяти по адресу 0000h:0000h
```

Заметим, что в этом случае сегментный регистр указывать обязательно.

Все остальные способы адресации относятся к группе косвенной адресации к памяти.

Базовая и индексная адресации.

Относительный адрес ячейки памяти находится в регистре, обозначение которого заключается в квадратные скобки. При использовании регистров BX или BP адресацию называют базовой, при использовании регистров SI или DI - индексной. При адресации через регистры BX, SI или DI в качестве сегментного регистра подразумевается DS; при адресации через BP - SS. Таким образом, косвенная адресация через регистр BP предназначена для работы со стеком. Однако при необходимости можно явно указать требуемый сегментный регистр.

Примеры

```
        mov     AL,[BX]     ;Сегментный адрес предполагается в
                            ;DS, смещение в BX
        mov     DL,ES:[BX]  ;Сегментный адрес находится в ES,
                            ;смещение в BX
        mov     DX,[BP]     ;Сегментный адрес предполагается в
                            ;SS, смещение в BP
        mov     AL,[DI]     ;Сегментный адрес предполагается в
                            ;DS, смещение в DI
```

Базовая и индексная адресации со смещением.

Относительный адрес операнда определяется суммой содержимого регистра (BX, BP, SI или DI) и указанного в команде числа, которое довольно неудачно называют смещением.

Пример

```
mas     db      1,2,5,3,7,9,8,3,4;Массив символов
        mov     BX,2        ;BX-индекс элемента в массиве
        mov     DL,mas[BX]  ;В DL заносится третий элемент массива
```

В этом примере относительный адрес адресуемого элемента массива mas вычисляется как сумма содержимого BX (2) и значения символического обозначения mas, которое совпадает с относительным адресом

начала массива mas. В результате в регистр DL будет загружен элемент массива mas с индексом 2, то есть число 5. Предполагается, что базовый адрес сегмента, в который входит массив mas, загружен в DS. Такой же результат даст следующая последовательность команд:

```
mov    BX,offset mas;BX=относительный адрес ячейки mas
mov    DL,2[BX]
```

Здесь относительный адрес адресуемого элемента массива mas вычисляется как сумма содержимого регистра BX и дополнительного смещения, задаваемого константой 2. Последняя команда может быть записана в следующем виде:

```
mov    DL,[BX+2]
mov    DL,[BX]+2
```

Адресация с помощью регистров SI и DI осуществляется аналогично. При использовании регистра BP следует помнить, что в качестве сегментного регистра по умолчанию подразумевается регистр SS

Базово-индексная адресация.

Относительный адрес операнда определяется суммой содержимого базового и индексного регистров. Допускается использование следующих пар:

```
[BX][SI]
[BX][DI]
[BP][SI]
[BP][DI]
```

Если в качестве базового регистра выступает BX, то в качестве сегментного подразумевается DS (первые две команды), при использовании в качестве базового регистра BP сегментным регистром по умолчанию назначается SS (вторые две команды). При необходимости можно явно указать требуемый сегментный регистр.

Примеры

```
mov    BX,[BP][SI];В BX засылается слово из стека
                  (сегментный адрес находится в SS), а
                  ;смещение вычисляется как сумма
                  ;содержимого BP и SI
mov    ES:[BX+DI],AX;В ячейку памяти, сегментный адрес
                  ;которой хранится в ES, а смещение равно
                  ;сумме содержимого BX и DI, пересылается
                  ;содержимое AX
```

Базово-индексная адресация со смещением.

Относительный адрес операнда определяется суммой трех величин: содержимого базового и индексного регистров, а также дополнительного смещения. Допускается использование тех же пар регистров, что и в базово-индексном способе; так же действуют и правила определения сегментных регистров.

Примеры

```
        mov     mas[BX][SI],10;Символ с кодом 10 ("возврат
                               ;каретки") пересылается в ячейку
                               ;памяти, сегментный адрес которой
                               ;хранится в DS, а смещение равно
                               ;сумме содержимого регистров BX и SI
                               ;и относительного адреса ячейки mas
        mov     AX,[BP+2+DI];В AX пересылается из стека слово,
                               ;смещение которого равно сумме BP,
                               ;DI и добавки, равной двум
```

Значительная часть рассмотренных выше способов адресации служит для обращения к ячейкам памяти. Таким образом, один и тот же конечный результат можно получить с помощью различных способов адресации. Например, все три приведенные ниже команды

```
        mov     DL,mas+3
        mov     DL,mas[BX] ;В BX заранее занесено число 3
        mov     DL,[SI][BX];В BX заранее занесено число 3, а в
                           ;SI - смещение mas
```

приведут к загрузке в регистр DL четвертого элемента массива mas (если выполняются описанные в комментариях условия). Однако команды с использованием различных способов адресации занимают различный объем памяти и выполняются за разное время. Так, первая из приведенных выше команд потребует для выполнения 15 машинных тактов, вторая - 18, а третья - 16. Разница невелика, однако при многократном выполнении команд в циклах суммарный эффект может быть значителен. С другой стороны, первые две команды занимают в памяти по 4 байта, а третья только 2. Таким образом, тщательный выбор способов адресации позволяет в какой-то степени оптимизировать программы по времени выполнения или требуемой памяти, а иногда и по тому, и по другому.

Время выполнения команды можно определить, разделив требуемое для ее выполнения число машинных тактов на частоту работы конкретного процессора. Последняя величина колеблется для используемых в настоящее время персональных компьютеров в очень широких пределах - от единиц до ста МГц и более. Поэтому время выполнения команд и всей программы в целом оценивают обычно не в абсолютных величинах, а именно в числе машинных тактов.

Полное время выполнения команды можно выразить в виде суммы двух составляющих: базового времени выполнения команды, которое зависит от вида команды и способа адресации, и времени вычисления исполнительного адреса, если операнд находится в памяти. Вторая составляющая тоже зависит от способа адресации. Разброс времен выполнения для разных команд может быть весьма значителен. Так, команда сложения содержимого двух регистров требует 3 тактов, прямого внутрисегментного перехода - 15 тактов, а команда деления - около 100 так-

тов. То же, хотя и в меньшей степени, относится к времени вычисления исполнительного адреса. Если смещение ячейки памяти указывается в регистре (базовом или индексном), то к базовому времени выполнения команды следует прибавить 5 тактов, при базово-индексной адресации - 8, а в случае базово-индексной адресации со смещением - 12.

Рассмотрим несколько примеров оптимизации программы по памяти и времени выполнения.

1. Обнуление регистра

```
mov    AX,0        ;4 такта, 3 байт
sub    AX,AX       ;3 такта, 2 байт
xor    AX,AX       ;3 такта, 2 байт
```

Видно, что первая команда, будучи наиболее наглядной, в то же время уступает двум другим по своим оптимизационным характеристикам.

2. Загрузка в регистр относительного адреса ячейки памяти

```
mov DX,offset mem;4 такта, 3 байт
lea DX,mem       ;8 тактов, 4 байт
```

Первый способ загрузки в регистр смещения ячейки оказывается лучше во всех отношениях. С другой стороны, команда lea имеет больше возможностей, поскольку в ней можно использовать любые способы адресации.

3. Загрузка в пару регистров полного двухсловного адреса

```
addr    dd     array_1    ;Полный адрес массива array_1
                          ;в двухсловной ячейке addr
        mov BX, word ptr addr;Получение смещения,14 тактов,4 байт
        mov ES, word ptr addr+2;Сегмент, 14 тактов, 4 байт
        les BX, addr      ;Получение сегмента и смещения,
                          ;16 тактов, 4 байт
```

Результат выполнения двух первых команд такой же, как и одной третьей. Однако использование команды les (или lds для загрузки сегментного регистра DS) дает существенную экономию и времени, и памяти.

4. Умножение целого числа на степень двух

```
mov BX,32          ;Сомножитель в BX, 4 такта, 3 байт
mul BX             ;Умножение, до 133 тактов, 2 байт
mov CL,5           ;Величина сдвига, 4 такта, 3 байт
sal AX,CL          ;Сдвиг, 28 тактов, 2 байт
```

Последний пример несколько специфичен. Если вам необходимо умножить на степень числа 2 (2, 4, 8, 16,...), выполнение этой операции можно существенно ускорить, заменив умножение сдвигом влево на число позиций, равное показателю степени. Каждый сдвиг влево на

одну позицию приводит к умножению на два. Таким образом, первая и вторая пары команд в вышеприведенном примере эквивалентны по результату, но замена умножения сдвигом позволяет ускорить операцию в несколько раз. Те же рассуждения применимы к делению на степень числа 2, которое целесообразно заменять сдвигом вправо.

Статья 31

Программы с несколькими сегментами команд

До сих пор рассматривались небольшие программы, в которых и главная процедура, и подпрограммы, если они были, помещались в одном сегменте команд. Данные, используемые в программах, также были небольшого объема и помещались в одном сегменте данных. В этом случае суммарный объем команд во всех процедурах не мог превысить 64 Кбайт. Общий объем данных тоже не мог превысить 64 Кбайт. Про такие программы говорят, что они принадлежат к малой модели памяти (этот термин используется при программировании на языках высокого уровня типа C или C++).

Современные прикладные программы обычно имеют больший объем. Однако увеличить размер сегмента нельзя; если суммарный объем команд программы превышает 64 Кбайт, в ней следует организовать несколько сегментов команд. Программы с несколькими сегментами команд и одним сегментом данных относятся к средней модели памяти.

Программа, содержащая несколько сегментов команд, в принципе может иметь "линейный" характер. В этом случае сначала выполняются все команды первого сегмента, затем процессор переходит к выполнению команд второго сегмента и т.д. Для того, чтобы организовать такую программу, надо научиться осуществлять переход не в пределах одного сегмента, как это было до сих пор, а в другой сегмент команд. Однако удобнее большую программу составить из процедур-подпрограмм. В этом случае надо уметь вызывать подпрограмму из другого сегмента.

Любое обращение к другому сегменту команд носит название дальнего, или межсегментного. Обращение же в пределах одного сегмента называется ближним, или внутрисегментным. В соответствии с этим различают команды ближних и дальних переходов, а также команды ближних и дальних вызовов подпрограмм.

Для того, чтобы изучить правила организации многосегментных программ, мы воспользуемся программным комплексом, описанным в статьях 17...19. Комплекс состоял из двух процедур-подпрограмм: hexasc для получения символьного представления четырехбитового числа и binasc, преобразующей двоичное слово в 16-ричную символьную форму. При этом процедура binasc для преобразования каждой 16-ричной цифры вызывала подпрограмму hexasc.

Выделим обе процедуры-подпрограммы в отдельный сегмент команд. Это позволит нам при необходимости довести объем главной процедуры до 64 Кбайт и многократно увеличить суммарный объем всех используемых в комплексе процедур, поскольку для них можно предусмотреть не один, а сколько угодно сегментов команд.

Модифицируя программу, преобразуем главную процедуру примера 19.1 так, чтобы она выводила на экран сегментные адреса обоих сегментов команд, а также сегмента данных. Эта информация позволит нам вспомнить, как располагаются в физической памяти сегменты загружаемой программы. Для удобства читателя в примере 31.1 приводится полный текст программного комплекса, хотя процедуры binasc и hexasc целиком заимствованы из примеров 17.1 и 18.1.

В этом и последующих примерах для упрощения текста программ мы будем использовать библиотеку макрокоманд, описанную в статье 27. Библиотека представляет собой файл (мы дали ему имя MAC.MAC) с текстами следующих макроопределений:

write - вывод на экран строки текста с помощью функции DOS 09h;

outprog - завершение программы с помощью функции DOS 4Ch;

delay - программная задержка на время, задаваемое параметром макрокоманды;

stop - остановка программы с помощью функции 08h до нажатия любой клавиши.

Имя файла с макроопределением макрокоманд объявляется в тексте программы с помощью директивы ассемблера include.

Рассмотрим теперь саму программу. Хотя в ней будет несколько сегментов команд, все они могут быть описаны в единственном файле с исходным текстом программы (хотя могут быть и разнесены по разным файлам).

Пример 31.1. Программа с двумя сегментами команд

```
text1   segment 'code'    ;Сегмент для главной процедуры
        assume CS:text1, DS:data
        include mac.mac   ;Объявление макробиблиотеки
main    proc
        mov    AX,data
        mov    DS,AX
;Выведем адрес сегмента команд text1. Он находится в CS
        mov    AX,CS
```

```
        mov   SI,offset seg1+10
        call  far ptr binasc
        write seg1          ;Вывод на экран
;Выведем адрес сегмента команд text2.
        mov   AX,text2
        mov   SI,offset seg2+10
        call  far ptr binasc
        write seg2          ;Вывод на экран
;Выведем адрес сегмента данных. Он находится в DS
        mov   AX,DS
        mov   SI,offset dataseg+10
        call  far ptr binasc
        mov   AH,09h
        write dataseg       ;Вывод на экран
        outprog             ;Завершение программы
main    endp
text1   ends
text2   segment 'code'      ;Сегмент для подпрограмм
        assume CS:text2,DS:data
hexasc  proc
;Подпрограмма получения символьного представления четырехбитового
;числа. На входе: исходное число в младшей половине AL, адрес
;строки, куда надо поместить результат, в DS:SI. На выходе:
;ASCII-представление полученного числа в памяти по адресу DS:SI,
;исходное содержимое регистра AL разрушается
        push  BX            ;Сохраним используемый в подпрограмме
                            ;регистр
        mov   BX,offset tblhex;BX=адрес таблицы трансляции
        xlat                ;Команда табличной трансляции
        mov   [SI],AL       ;Сохраним результат
        pop   BX            ;Восстановим регистр BX
        ret                 ;Возврат в вызывающую программу
hexasc  endp
binasc  proc  far
;Подпрограмма преобразования двоичного слова в 16-ричную
;символьную форму. AX=преобразуемое число, DS:SI=адрес строки,
;куда помещается результат преобразования
        push  CX            ;Сохраним используемый регистр
        push  AX            ;Сохраним наше число в стеке
        and   AX,0F000h     ;Выделим старшую четверку битов
        mov   CL,12         ;Счетчик сдвига
        shr   AX,CL         ;Сдвиг вправо на 12 бит
        call  hexasc        ;Преобразуем в символ и отправим его
                            ;по адресу [SI]
        pop   AX            ;Вернем в AX исходное число
        push  AX            ;И снова сохраним в стеке
        and   AX,0F00h      ;Выделим вторую четверку битов
        mov   CL,8          ;Счетчик сдвига
        shr   AX,CL         ;Сдвиг вправо на 8 бит
        inc   SI            ;Сдвинемся к следующему байту в
                            ;строке результата
        call  hexasc        ;Преобразуем в символ и отправим его
                            ;по адресу [SI]
        pop   AX            ;Вернем в AX исходное число
        push  AX            ;И снова сохраним в стеке
        and   AX,0F0h       ;Выделим третью четверку битов
        mov   CL,4          ;Счетчик сдвига
```

```
        shr     AX,CL       ;Сдвиг вправо на 4 бита
        inc     SI          ;Сдвинемся к следующему байту в
                            ;строке результата
        call    hexasc      ;Преобразуем в символ и отправим его
                            ;по адресу [SI]
        pop     AX          ;Вернем в AX исходное число
        push    AX          ;И снова сохраним в стеке
        and     AX,0Fh      ;Выделим младшую четверку битов
        inc     SI          ;Сдвинемся к следующему байту в
                            ;строке результата
        call    hexasc      ;Преобразуем в символ и отправим его
                            ;по адресу [SI]
        pop     AX          ;Восстановим стек
        pop     CX          ;Восстановим используемый регистр
        ret                 ;Возврат в вызвавшую процедуру
binasc  endp
text2   ends
data    segment             ;Начало сегмента данных
seg1    db      'TEXT1=     ****h',10,13,'$'
seg2    db      'TEXT2=     ****h',10,13,'$'
dataseg db      'DATA=      ****h',10,13,'$'
tblhex  db      '0123456789ABCDEF'
data    ends                ;Конец сегмента данных
stack   segment stack 'stack';Начало сегмента стека
        dw 128 dup (0)      ;Стек
stack   ends                ;Конец сегмента стека
        end     main        ;Конец программы с указанием точки входа
```

Обратим внимание на отличия данного примера от программы 19.1. Сегмент команд с главной процедурой получил название text1. Соответственно изменено имя сегмента в операторах assume и ends. В главной процедуре, как и раньше, имеются три схожих участка. Сначала преобразуется в символьную форму и выводится на экран адрес текущего программного сегмента text1, который берется из сегментного регистра CS. На втором участке процедуры выводится сегментный адрес сегмента text2. Его адрес определяется, как значение имени сегмента text2. На третьем участке выводится адрес сегмента данных data, который извлекается из регистра DS, предварительно настроенного на этот сегмент.

В сегменте данных изменены обозначения полей с выводимыми сообщениями и сами сообщения.

Принципиальные отличия данного примера от его прототипа заключаются в том, что процедура binasc объявлена с описателем far (дальняя), а ее вызовы в главной процедуре сопровождаются описателями far ptr (far pointer, дальний указатель). Что изменилось при этом в программах процедур и чем оператор call far ptr отличается от простого call?

Рассмотрим фрагменты загрузочного модуля нашей многосегментной программы с указанием расположения некоторых команд, их кодов, смещений, мнемонических обозначений и описания их действия (рис. 31.1).

```
Смеще-  Код команды  Предложения программы   Действие команды
ние
                     text1 segment 'code'
                     assume CS:text1,DS:data
0000                 main proc
0000    B8 4455      mov AX,data
0003    8E D8        mov DS,AX
0005    8C C8        mov AX,CS
0007    BE 000A      mov SI,offset seg1+10
000A    9A 0009 4451 call far ptr binasc ---->1) CS=444D -> в стек
                                              2) IP=000F -> в стек
                                              3) 4451 -> в CS
                                              4) 0009 -> в IP
000F    B4 09        mov AH,09h;1-я строка макрорасширения write
...
003E                 main endp
003E                 text1 ends
0000                 text2 segment 'code'
                     assume CS:text2,DS:data
0000                 hexasc proc
0000    53           push BX
...
0008    C3           ret ----------> из стека 0015 -> в IP
0009                 hexasc endp
0009                 binasc proc far
0009    51           push CX
...
0012    E8 FFEB      call hexasc ----> 1) IP=0015h -> в стек
                                       2) 0015h+FFEB=0000 -> в IP
0015    58           pop AX
...
003A    CB           ret ----------> 1) из стека 000F -> в IP
                                     2) из стека 444D -> в CS
003B                 binasc endp
003B                 text2 ends
```

Рис. 31.1. Фрагменты загрузочного модуля программы 31.1 с поясняющей информацией.

Команда call far ptr binasc расположена по относительному адресу 000Ah. В ее код входит код операции дальнего вызова call far ptr (9Ah) и адрес процедуры binasc, на которую надо осуществить переход. Этот адрес записан в виде двух слов: относительного адреса процедуры binasc в том сегменте, где она расположена (0009Bh) и сегментного адреса этого сегмента, который, как выяснилось после загрузки программы в память, равен 4451h. Таким образом, процессор, считав из памяти код команды, имеет полную информацию о том, куда надо осуществить переход.

При выполнении команды call far ptr процессор помещает в стек два слова: сначала сегментный адрес текущего сегмента (text1), затем - адрес возврата (содержимое IP, т.е. адрес следующей команды, в данном случае 000Fh). Следует обратить внимание на порядок записи в память компонентов двухсловного адреса: всегда в слово памяти с большим адресом записывается сегментный адрес, а в слово памяти с меньшим ад-

ресом - относительный адрес, или смещение. После сохранения в стеке адреса возврата процессор заносит в сегментный регистр команд CS сегментный адрес процедуры binasc, а в IP - относительный адрес этой процедуры, которые он извлекает из кода команды call. В результате указатель стека смещается вверх на два слова, а процессор переходит на выполнение подпрограммы из другого сегмента.

Команда ret процедуры binasc, расположенная по адресу 003Ah в сегменте text2, выполняет обратную операцию - снимает со стека два верхних слова, загружая первое в IP, а второе - в CS, в результате чего процессор возвращается к выполнению вызывающей процедуры.

Так осуществляется дальний, или межсегментный вызов процедуры, расположенной в другом сегменте команд.

Процедура binasc по ходу своего выполнения вызывает вложенную процедуру hexasc, размещенную нами в том же сегменте text2. Этот вызов осуществляется ближним оператором call (смещение 0012h в сегменте text2). В этом случае, как уже описывалось в статье 16, в стеке сохранятся только относительный адрес возврата и, соответственно, команда ret процедуры hexasc (смещение 0008 в сегменте text2) снимает со стека только одно слово.

Почему же команда ret процедуры binasc снимает со стека два слова, в то время как такая же, на первый взгляд, команда процедуры hexasc снимает только одно? В действительности, однако эти две команды не эквивалентны. Сравнив строки с адресами 003Ah и 0008h во втором сегменте, можно заметить, что машинные коды этих двух команд ret различаются. Так получилось потому, что процедура binasc объявлена нами с помощью описателя far дальней. Транслятор воспринимает это объявление, как требование преобразовать мнемоническое обозначение ret, встретившееся в этой процедуре, в код команды дальнего возврата CBh. Процедура hexasc не имеет явного описателя типа, и по умолчанию рассматривается транслятором, как ближняя. С целью повышения наглядности программы ее можно было назначить ближней явным образом с помощью описателя near. В соответствии с типом процедуры и команда ret, встретившаяся в этой процедуре, транслируется в код ближнего возврата C3h.

Из приведенного рассмотрения должно быть ясно, что ближние процедуры следует вызывать только из того же сегмента командой ближнего вызова call, в то время, как процедуры, объявленные, как дальние, следует вызывать только с помощью команды дальнего вызова call fat ptr. Лишь в этом случае завершающие эти процедуры команды ret будут работать правильно.

Статья 32

Программы с несколькими сегментами данных

В предыдущей статье была рассмотрена программа с двумя сегментами команд. Вводя дополнительные сегменты команд, мы можем увеличивать общий объем команд, входящих в программу, усложняя ее алгоритм и увеличивая возможности. Часто, однако, относительно несложная программа обрабатывает значительные по объему массивы данных. В таких случаях требуется увеличивать число сегментов данных. Программы с одним сегментом команд и несколькими сегментами данных относятся к компактной модели памяти.

При наличии нескольких сегментов данных возникают проблемы их адресации. Поскольку процессор имеет только два сегментных регистра данных DS и ES (в процессорах 80386 и 80486 предусмотрены дополнительные сегментные регистры FS и GS, что увеличивает возможности этих процессоров по одновременной адресации данных, однако мы пока рассматриваем процессор 8086), в каждый момент времени программе могут быть доступны лишь два сегмента данных общим объемом 128 Кбайт. Если число сегментов данных в программе больше двух, работать с ними придется последовательно, предварительно настраивая сегментные регистры данных на требуемую пару сегментов.

Рассмотрим программу с двумя сегментами данных (пример 32.1). Программа носит несколько академический характер, однако в ней продемонстрированы некоторые полезные практические приемы. В программе создается тестовый массив данных объемом 32 Кбайт, который затем сортируется по признаку четности или нечетности числа установленных битов в каждом байте массива. Байтами с четным числом битов заполняется один выходной массив, байтами с нечетным числом битов - другой. Поскольку в общем случае количество тех и других байтов заранее неизвестно, под выходные массивы выделяется по 32 Кбайт. Эти массивы размещаются во втором сегменте данных, адресуемом через регистр ES.

Пример 32.1. Программа с двумя сегментами данных

```
        assume CS:text,DS:data1
main    proc
        mov    AX,data1    ;Настроим регистр DS
        mov    DS,AX       ;на первый сегмент data1
```

```
        mov    AX,data2    ;Настроим регистр ES
        mov    ES,AX       ;на второй сегмент data2
;Заполним исходный массив raw натуральным рядом чисел
        mov    CX,32768    ;Счетчик цикла
        mov    BX,0        ;Индекс в массиве raw
        mov    DL,0        ;Число-заполнитель, начнем с нуля
fill:   mov    raw[BX],DL ;Отправим число в исходный массив
        inc    DL          ;Образуем следующее число
        inc    BX          ;Сдвинемся к следующему байту
        loop   fill        ;Цикл по всему массиву
;Рассортируем числа из массива raw по двум массивам сегмента
;data2: в массив evenbit отправим байты с четным числом
;установленных битов, в массив oddbit - с нечетным
        mov    CX,32768    ;Счетчик цикла
        mov    BX,0        ;Индекс в исходном массиве raw
        mov    SI,0        ;Индекс в массиве четных байтов evenbit
        mov    DI,0        ;Индекс в массиве нечетных байтов oddbit
pros:   mov    DL,raw[BX] ;Получим байт из массива raw
        test   DL,0FFh     ;Проверим его четность
        pushf              ;Сохраним в стеке результаты проверки
        inc    BX          ;Инкремент индекса в массиве raw
        popf               ;Восстановим флаги процессора
        jp     parity      ;Если четно - переход на parity
        mov    ES:oddbit[DI],DL;Нечетно, в массив нечетных байтов
        inc    DI          ;Инкремент его указателя
        jmp    outc        ;И на следующий шаг цикла
parity: mov    ES:evenbit[SI],DL;Четно, в массив четных байтов
        inc    SI          ;Инкремент его указателя
outc:   loop   pros        ;Повторять CX раз
        ...
main    endp
text    ends
data1   segment
raw     db     32768 dup (0)
data1   ends
data2   segment
oddbit  db     32768 dup (0)  ;Числа с нечетным числом битов
evenbit db     32768 dup (0)  ;Числа с четным числом битов
data2   ends
```

В программе предусмотрены два сегмента данных data1 и data2. Сегментный регистр DS будет использоваться для работы с первым из них, поэтому в операторе assume имеется определение DS:data1.

Программа начинается с настройки обоих сегментных регистров. Далее осуществляется заполнение массива исходных данных raw сортируемыми числами. Для простоты массив заполняется натуральным рядом байтовых чисел, которые, естественно, повторяются с шагом 256.

Сортировка осуществляется в цикле. Четность или нечетность числа установленных в байте битов определяется с помощью пары команд test (тестирование) и jp (jump if parity, переход, если паритет четен). Двухоперандная команда test фактически выполняет операцию логического И (and) над двумя операндами и, в зависимости от результата, устанавливает флаги ZF, SF и PF. Команда удобна для определения целого ряда

характеристик анализируемого числа в целом или заданной его части (заданных битов). На рис. 32.1. на нескольких примерах показано действие этой команды.

```
Первый операнд (анализируемое число) 1111 0000 1010 0101 = F0A5h
Второй операнд (маска)               1111 1111 1111 1111 = FFFFh
-------------------------------------------------------------------
Результат операции И                 1111 0000 1010 0101 = F0A5h
(в данном случае результат совпадает с анализируемым числом)
Возможные переходы: jnz (результат не нуль)
                    js  (в результате установлен знаковый бит)
                    jp  (в младшем байте результата четное число
                         установленных битов)
```

а

```
Первый операнд (анализируемое число) 1111 0000 1010 0101 = F0A5h
Второй операнд (маска)               1000 0000 0000 0011 = 8003h
-------------------------------------------------------------------
Результат операции И                 1000 0000 0000 0001 = 8001h
Возможные переходы: jnz(результат не нуль)
                    js (в результате установлен знаковый бит)
                    jnp(в заданных битах младшего байта результата
                        нечетное число установленных битов)
```

б

```
Первый операнд (анализируемое число) 1111 0000 1010 0101 = F0A5h
Второй операнд (маска)               0000 0000 0000 1010 = 000Ah
-------------------------------------------------------------------
Результат операции И                 0000 0000 0000 0000 = 0000h
Возможные переходы: jz (результат нуль)
                    jp (в заданных битах младшего байта результата
                        четное число [нуль] установленных битов)
```

в

```
Первый операнд (анализируемое число) 1111 0000 1010 0101 = F0A5h
Второй операнд (маска)               0000 0000 1101 0000 = 00D0h
-------------------------------------------------------------------
Результат операции И                 0000 0000 1000 0000 = 0080h

Возможные переходы: jnz(результат не нуль)
                    jnp(в заданных битах младшего байта результата
                        нечетное число установленных битов)
```

г

Рис. 32.1. Несколько примеров действия команды test.

Стоит перечислить несколько типичных применений команды test.

Если в анализируемом числе установлен старший, знаковый бит, и он указан в маске (возможно, в числе прочих битов), то наряду со сбро-

сом флага ZF устанавливается флаг SF, что можно обнаружить с помощью команды js (рис. 32.1, а и б).

Если в анализируемом числе не установлен ни один из битов, заданных маской, то устанавливается флаг ZF, что можно обнаружить с помощью команд jz или je (рис. 32.1, в).

Если в анализируемом числе установлен хотя бы один из битов, заданных маской, то флаг ZF сбрасывается, что можно обнаружить с помощью команд jnz или jne (рис. 32.1, а, б и г).

Очередной байт исходного массива читается в регистр DL и командой test проверяется все его содержимое (поскольку второй операнд команды test, число FFh, задает для проверки все биты). О результате проверки на четность можно узнать по состоянию флага PF регистра флагов, который анализируется командой jp. Однако независимо от четности или нечетности байта необходимо выполнить инкремент указателя исходного массива BX. Так как команда inc влияет на состояние флага PF, перед ней содержимое регистра флагов сохраняется в стеке командой pushf (push flags, занесение в стек флагов), а после нее восстанавливается обратной командой popf (pop flags, восстановление из стека флагов). Это весьма распространенный прием, позволяющий "отсрочить" условный переход, если между командой анализа и условным переходом необходимо выполнить какие-то действия.

Каким бы ни был анализируемый байт - четным или нечетным, его следует записать во второй сегмент данных, адресуемый через ES. Поэтому в командах записи в память содержимого DL регистр ES указан в явной форме. Это еще один пример замены сегмента. Если опустить префикс ES:, адресация осуществилась бы через регистр DS, и мы попали бы в исходный, а не в результирующий массив. Полезно иметь в виду, что префикс замены сегмента увеличивает на 1 байт длину команды. В тех случаях, когда требуется максимальное сокращение размера программы, это может иметь значение.

В рассмотренной программе не предусмотрена наглядная демонстрация ее работы. Запустите ее в отладчике, остановите перед строками завершения и командой отладчика D просмотрите содержимое массивов raw, oddbit и evenbit.

Статья 33

Директива assume, инициализация сегментных регистров и замена сегментов

Обсудим несколько подробнее роль директивы ассемблера assume. Вернемся для этого к примеру 32.1 и рассмотрим некоторые предложения этой программы. В предложении с директивой assume, с которого начинался текст сегмента команд, устанавливалось соответствие сегмента команд text сегментному регистру CS и сегмента данных data1 сегментному регистру DS:

```
assume CS:text,DS:data1
```

Директивы assume, которых в программе может быть любое количество, в принципе могут располагаться в любом месте программы, однако обязательно до тех программных строк, в которых выполняется обращение к описываемым в конкретной директиве сегментам. Поэтому предложение вида

```
assume CS:text
```

должно стоять в программе до любых программных строк и даже до объявления процедур. Что же касается установки соответствия регистра DS и конкретного сегмента данных, то, если это соответствие не описано в первой директиве assume, оно может быть описано и позже, например (для программы 32.1):

```
        assume DS:data1
fill:   mov    raw[BX],DL;Отправим число в исходный массив
```

Поскольку к моменту трансляции указанной выше команды mov транслятор уже знает из директивы assume, что сегмент data1 сопоставляется с регистром DS, а имя raw описано именно в сегменте data1, команда mov транслируется в такой код, что процессор при его выполнении обратится к ячейке памяти с относительным адресом raw[BX], взяв сегментный адрес из регистра DS. Транслятор выполняет (не слишком строгую) проверку правильности ссылок на данные, так что если бы ячейка raw входила в другой сегмент, была бы зафиксирована ошибка. Таким образом, описание в директиве assume соответствия сегментного регистра DS сегменту данных позволяет в какой-то степени контроли-

ровать правильность написания программы и, главное, избавляет нас от необходимости указывать в каждой строке, содержащей ссылку на имя данных, в каком сегментом регистре находится сегментный адрес этих данных. По умолчанию будет подразумеваться регистр DS.

Однако из сказанного совершенно не следует, что к моменту выполнения строки с обращением к данным в регистре DS будет в действительности находиться сегментный адрес соответствующего сегмента. Более того, его там и не будет, если мы об этом специально не позаботимся. Предложения

```
mov   AX,data1
mov   DS,AX
```

как раз и выполняют загрузку в сегментный регистр DS сегментного адреса нашего сегмента данных. Как мы уже знаем, при загрузке программы в память оба сегментных регистра данных указывают на PSP, а совсем не на сегменты данных, так что до выполнения указанных выше строк наш сегмент данных неадресуем.

Несколько по-иному обстоит дело с дополнительными сегментами данных, к которым мы будем обращаться через сегментный регистр ES. Поскольку в программе 32.1 регистр ES в директиве assume не описан, его следует в явной форме указывать во всех предложениях программы, где выполняется адресация к дополнительному сегменту данных (data2 в примере 32.1):

```
        mov   ES:oddbit[DI],DL;Нечетно, в массив нечетных байтов
        inc   DI          ;Инкремент его указателя
        jmp   outc        ;И на следующий шаг цикла
parity: mov   ES:evenbit[SI],DL;Четно, в массив четных байтов
```

При выполнении приведенных выше команд mov процессор обратится к ячейкам памяти с относительными адресами oddbit[DI] и evenbit[SI], взяв при этом сегментный адрес из регистра ES. Однако и в этом случае занесение в регистр ES требуемого сегментного адреса должно быть выполнено в программе явным образом, например, предложениями

```
mov   AX,data2
mov   ES,AX
```

Директива assume позволяет создать умолчание и для регистра ES. Однако для этого необходимо, чтобы дополнительный сегмент данных data2 был описан в программе до сегмента команд (см. пример 33.1). В этом случае включение в текст программы директивы

```
assume ES:data2
```

избавит нас от необходимости указывать в явной форме регистр ES во всех командах с обращением к данным из сегмента data2. Если таких

предложений в программе много, то это может несколько облегчить процесс написания программы.

Пример 33.1. Описание дополнительного сегмента данных в директиве assume

```
data2   segment             ;(1)Объявление дополнительного
                            ;сегмента данных
oddbit  db      32768 dup (0);(2)числа с нечетным числом битов
evenbit db      32768 dup (0);(3)числа с четным числом битов
data2   ends                ;(4)
text    segment 'code'      ;(5)Объявление сегмента команд
        assume CS:text,DS:data1,ES:data2;(6)Описаны все
                            ;сегментные регистры (кроме SS)
main    proc                ;(7)
        mov     AX,data1    ;(8)Настроим регистр DS
        mov     DS,AX       ;(9)на первый сегмент data1
        mov     AX,data2    ;(10)Настроим регистр ES
        mov     ES,AX       ;(11)на второй сегмент data2
;Заполним исходный массив raw натуральным рядом чисел
        mov     CX,32768    ;(12)Счетчик цикла
        mov     BX,0        ;(13)Индекс в массиве raw
        mov     DL,0        ;(14)Число-заполнитель, начнем с нуля
fill:   mov     raw[BX],DL  ;(15)Отправим число в исходный массив.
                            ;По умолчанию используется регистр DS,
                            ;так как raw находится в сегменте data1
        inc     DL          ;(16)Образуем следующее число
        inc     BX          ;(17)Сдвинемся к следующему байту
        loop    fill        ;(18)Цикл по всему массиву
;Рассортируем числа из массива raw по двум массивам сегмента
;data2: в массив evenbit отправим байты с четным числом
;установленных битов, в массив oddbit - с нечетным
        mov     CX,32768    ;(19)Счетчик цикла
        mov     BX,0        ;(20)Индекс в исходном массиве raw
        mov     SI,0        ;(21)Индекс в массиве четных байтов
                            ;evenbit
        mov     DI,0        ;(22)Индекс в массиве нечетных байтов
                            ;oddbit
pros:   mov     DL,raw[BX]  ;(23)Получим байт из массива raw. По
                            ;умолчанию используется регистр DS,
                            ;так как raw в сегменте data1
        test    DL,0FFh     ;(24)Проанализируем его четность
        pushf               ;(25)Сохраним в стеке результаты анализа
        inc     BX          ;(26)Инкремент индекса в массиве raw
        popf                ;(27)Восстановим флаги процессора
        jp      parity      ;(28)Если четно - переход на parity
        mov     oddbit[DI],DL;(29)Нечетно, в массив нечетных
                            ;байтов. По умолчанию используется ES,
                            ;так как oddbit в сегменте data1
        inc     DI          ;(30)Инкремент его указателя
        jmp     outc        ;(31)И на следующий шаг цикла
parity: mov     evenbit[SI],DL;(32)Четно, в массив четных байтов.
                            ;По умолчанию используется регистр ES,
                            ;так как evenbit в сегменте data1
        inc     SI          ;(33)Инкремент его указателя
outc:   loop    pros        ;(34)Повторять CX раз
        mov     AX,4C00h    ;(35)Завершение
```

```
        int     21h             ;(36)программы
main    endp                    ;(37)
text    ends                    ;(38)Конец сегмента команд
data1   segment                 ;(39)Объявление сегмента данных
raw     db      32768 dup (0);(40)
data1   ends                    ;(41)
```

Программа, предложенная в статье 32 и несколько модифицированная в настоящей статье, написана неоптимально с точки зрения объема расходуемой памяти и времени выполнения. Некоторого улучшения программы можно достигнуть, заменив адресацию со смещением на чисто регистровую - базовую, индексную или базово-индексную. Ниже приводятся только измененные строки (нумерация строк сохраняется).

Пример 33.2. Оптимизированная программа

```
        ...
        assume CS:text,DS:data1;(6)Нет необходимости описывать ES
        ...
        mov     BX,offset raw;(13)Индекс в массиве raw
        ...
fill:   mov     [BX],DL     ;(15)Отправим число в исходный массив.
                            ;По умолчанию используется DS, так как
        ...                 ;raw в сегменте data1
        mov     BX,offset raw;(20)Индекс в исходном массиве raw
        mov     SI,offset evenbit;(21)Индекс в массиве четных
                            ;байтов evenbit
        mov     DI,offset oddbit;(22)Индекс в массиве нечетных
                            ;байтов oddbit
pros:   mov     DL,[BX]     ;(23)Получим байт из массива raw
        ...
        mov     ES:[DI],DL ;(29)Нечетно, в массив нечетных байтов.
                            ;Явно указан ES, так как нет явного
        ...                 ;указания имени данных
parity: mov     ES:[SI],DL ;(32)Четно, в массив четных байтов.
                            ;Явно указан ES, так как нет явного
        ...                 ;указания имени данных
```

Сравним примеры 33.1 и 33.2. В директиве assume описаны только регистры CS и DS, так как в этом варианте программы нет прямого обращения к памяти по именам ячеек. В предложении 13 в регистр BX заносится не смещение относительно начала массива raw (первоначально равное 0), а относительный адрес массива. Это оптимизирует предложение 15, в котором теперь нет необходимости указывать адрес массива. В дальнейшем последовательное увеличение содержимого BX на 1 позволит нам, как и раньше, смещаться по массиву raw. Аналогичные изменения внесены и в предложения 20, 21 и 22. Во всех трех регистрах теперь хранятся (и наращиваются) не индексы внутри массивов, а смещения байтов массивов относительно начал соответствующих сегментов.

Важно обратить внимание на предложения 29 и 32. В процессе оптимизации из этих предложений изъяты мнемонические обозначения

ячеек. По этой причине транслятор не имеет никакой возможности определить, к какому сегменту данных выполняется обращение, и если бы в предложениях не был указан регистр ES, на этапе выполнения процессор извлек бы сегментный адрес из регистра DS, используемого в таких случаях по умолчанию. Команды с базовой, индексной или базово-индексной адресацией следует использовать с особой осторожностью, помня, что при отсутствии префикса замены сегмента в них подразумевается сегментный регистр DS.

Директива assume позволяет в некоторых случаях не указывать в командах программы сегментные регистры, однако она не запрещает это делать в случае необходимости. Часто по ходу программы приходится к одному и тому же сегменту данных обращаться то через один, то через другой сегментный регистр. Такая ситуация типична при использовании команд обработки строк (movs, stos и др.). Если некоторое поле памяти служит сначала приемником данных, пересылаемых в него из другого участка памяти, а затем источником при пересылке этих данных на какое-то третье место, то сначала это поле должно адресоваться через регистры ES:DI, а затем - через DS:SI. Ясно, что как бы мы не объявили соответствие сегментных регистров и сегментов данных, в одном из этих случаев придется воспользоваться префиксом замены сегмента. При этом стоит заметить, что хотя использование префикса увеличивает на один байт размер команды, оно, с другой стороны, повышает наглядность программы.

Директива assume может использоваться внутри программы неоднократно. Предположим, что на некотором, достаточно продолжительном участке программы, к определенным полям данных надо обращаться через сегментный регистр DS, а на другом участке - через регистр ES. Тогда между этими участками можно сменить соответствие сегментных регистров сегментам данных, как это схематически показано ниже.

```
;Сегмент данных
data    segment
mem1    dw      ?
mem2    dw      ?
data    ends
;Программные строки в сегменте команд
        assume DS:data
        mov     AX,mem1     ;Команда будет выполняться,
                            ;как mov AX,DS:mem1
        mov     BX,mem2     ;Команда будет выполняться,
                            ;как mov BX,DS:mem2
        assume DS:nothing
        assume ES:data
        mov     AX,mem1     ;Команда будет выполняться,
                            ;как mov AX,ES:mem1
        mov     BX,mem2     ;Команда будет выполняться,
                            ;как mov BX,ES:mem2
```

Директива assume DS:nothing (nothing - ничего) снимает закрепление за сегментом data регистра DS и позволяет закрепить за ним другой регистр, ES. Того же эффекта можно было достигнуть, закрепив регистр DS за каким-то другим сегментом (если, конечно, он имеется). Необходимо только перед описанием соответствия регистра ES и сегмента data отменить закрепление за этим сегментом регистра DS.

Очевидно, что независимо от директивы assume, перед первым участком необходимо настроить на сегмент data регистр DS, в перед вторым - ES.

Статья 34

Программы типа .COM

В некоторых случаях оказывается удобно не дробить программу на отдельные сегменты, а включить все компоненты программы в один сегмент. Этот единственный сегмент должен, таким образом, содержать префикс программы (PSP), коды команд, данные и стек. Такие односегментные программы, соответствующие (в терминологии языков высокого уровня) минимальной, или крошечной модели памяти, обычно образуют загрузочные модули типа .COM (в отличие от модулей типа .EXE, которые рассматривались до сих пор), хотя иногда могут иметь и другие расширения (например, SYS). Программы типа .COM не имеют особых преимуществ перед программами типа .EXE, кроме своей компактности, однако они широко используются, прежде всего, в качестве резидентных программ.

При создании программы типа .COM необходимо выполнение двух условий: во-первых, исходный текст программы должен быть написан в определенном формате, с ограничениями, соответствующими минимальной модели памяти, и, во-вторых, после компоновки объектного модуля и получения обычного загрузочного файла с расширением .EXE, необходимо преобразовать этот файл в формат .COM с помощью системной утилиты EXE2BIN:

```
EXE2BIN P.EXE P.COM
```

Следует иметь в виду, что компоновщик LINK создает загрузочный модуль в формате .EXE. Если, однако, исходная программа написана в

формате .COM, этот модуль .EXE не является полноценной программой и в большинстве случаев будет неработоспособен. Утилита EXE2BIN изменяет формат загрузочного модуля, превращая его в полноценную программу типа .COM.

Для того, чтобы освоить написание программ в формате .COM, воспользуемся самой первой программой из статьи 1 и "переоборудуем" ее требуемым образом (пример 34.1). Между прочим, поскольку в этой программе не было ни сегмента стека, ни сегмента данных, она уже в какой-то степени удовлетворяет требованиям формата минимальной модели памяти.

Пример 34.1. Простейшая программа типа .COM

```
text    segment 'code'
        assume CS:text,DS:text
        org    256;Резервирование места для PSP
main    proc
        mov    AH,09h
        mov    DX,offset message
        int    21h
        mov    AX,4C00h
        int    21h
message db     'Наука умеет много гитик$'
main    endp
text    ends
        end    main
```

Программа содержит единственный сегмент text, которому присвоен класс 'code'. В директиве assume указано, что сегментные регистры CS и DS будут соответствовать этому сегменту. Как уже подробно обсуждалось выше, упоминание в директиве assume регистра DS в данном случае не требуется, так как в программе нет ссылок на поля данных. Для более сложных программ, однако, сопоставление регистра DS и сегмента команд (в котором в данном случае только и могут располагаться данные) весьма полезно.

Директива org 256 резервирует 256 байт для PSP. Заполнять PSP будет по-прежнему система, но место под него в начале сегмента должен отвести программист. В программе нет необходимости инициализировать 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 256 (100h), поэтому сразу за директивой org 256 должно стоять первое выполнимое предложение программы. Если в начале программы желательно расположить данные, перед ними следует поместить команду перехода на реальную точку входа, например jmp entry.

Образ памяти программы типа .COM показан на рис. 34.1.

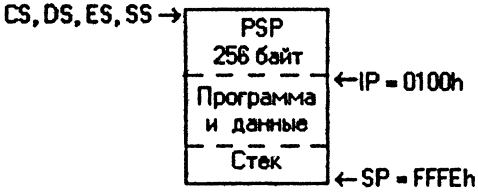


Рис. 34.1. Образ памяти программы .COM.

После загрузки программы все четыре сегментных регистра 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.

Создайте файл с программой из примера 34.1. Подготовьте программу к выполнению. Вы опять увидите сообщение компоновщика об отсутствии в программе сегмента стека:

```
LINK : warning L4021: no stack segment
```

В данном случае, однако, так и должно быть.

Выполните пробный прогон программы и убедитесь, что она работает, как ожидалось. Запустите программу под управлением отладчика. Внимательно рассмотрите содержимое регистров процессора: сегментных, указателя команд, указателя стека. Оцените, какого объема получился стек.

Статья 35

Ввод с клавиатуры десятичных чисел

Как мы уже видели, все данные, поступающие в компьютер с клавиатуры или выводимые на экран терминала, рассматриваются как коды ASCII отображаемых на экране символов. Для вывода на экран числа из ячейки памяти или регистра его следует предварительно преобразовать в коды ASCII соответствующих цифр той системы счисления, в которой

требуется отобразить число на экране. Точно также при вводе числа с клавиатуры полученные символы надо преобразовать в число (двоичное, поскольку любые данные хранятся в компьютере исключительно в двоичной форме). Очевидно, что процедура преобразования будет зависеть от того, какое число мы хотим набрать на клавиатуре - десятичное, шестнадцатеричное или двоичное. В настоящей статье будет рассмотрен ввод с клавиатуры десятичных чисел.

Помимо преобразования символьных кодов цифр в двоичные числа и объединения этих чисел с учетом весов отдельных десятичных разрядов, в программе должен быть предусмотрен анализ вводимых символов и исключение из входного потока кодов, не являющихся кодами десятичных цифр (такую операцию часто называют фильтрацией). Кроме того, следует предусмотреть какое-либо соглашение о способе завершения ввода. Например, можно всегда вводить 5 десятичных цифр (с лидирующими нулями, если число имеет меньше 5 десятичных разрядов). Удобнее вводить число без лидирующих нулей, завершая ввод нажатием клавиши <Enter> (или другой выделенной для этого клавиши). В соглашениях о правилах ввода (как говорят, в интерфейсе с программой) должна быть оговорена реакция программы на неправильно нажатую клавишу. Проще всего заставить программу игнорировать все нецифровые клавиши, хотя может быть предусмотрена и иная реакция, например, вывод сообщения об ошибке или подача звукового сигнала. Далее, могут быть варианты отображения вводимой информации на экране: отображать или нет случайно введенные нецифровые символы.

Программный фрагмент, приведенный в примере 35.1, осуществляет ввод с клавиатуры десятичных чисел с любым числом разрядов от 1 до 5. Ввод каждого числа завершается нажатием клавиши <Enter>. Вводимые символы проверяются на их принадлежность десятичному числу; нецифровые символы не воспринимаются и не отображаются на экране. Результат преобразования загружается в регистр AX.

Принцип преобразования вводимой строки символов в число заключается в следующем. Первая введенная цифра (старший десятичный разряд) преобразуется в двоичное число и засылается в результирующую ячейку number. После ввода и преобразования следующей цифры содержимое ячейки number умножается на 10 и к полученному произведению прибавляется введенное число. Далее этот процесс повторяется до нажатия клавиши <Enter>.

Пример 35.1. Ввод с клавиатуры десятичного числа

```
;Выведем на экран запрос, свидетельствующий об ожидании ввода
        mov     AH,02h      ;(1)
        mov     DL,'>'      ;(2)
        int     21h         ;(3)
        mov     DI,0        ;(4)Очистим регистр для результата
```

```
;Будем вводить и анализировать символы
inpt:   mov   AH,08h      ;(5)Функция ввода символа без эха
        int   21h         ;(6)
        cmp   AL,13       ;(7)Нажата клавиша <Enter>?
        je    done        ;(8)Да, ввод числа закончен
        cmp   AL,'9'      ;(9)Цифровой символ?
        ja    inpt        ;(10)Нет! На повторный ввод
        cmp   AL,'0'      ;(11)Цифровой символ?
        jb    inpt        ;(12)Нет! На повторный ввод
;Введен очередной цифровой символ. Выведем его на экран
        mov   AH,02h      ;(13)Функция вывода символа
        mov   DL,AL       ;(14)Символ должен быть в DL
        int   21h         ;(15)
        sub   AL,'0'      ;(16)Преобразуем символ в двоичное число
        xor   AH,AH       ;(17)Обнулим AH
        mov   CX,AX       ;(18)Сохраним полученную цифру в CX
        mov   AX,DI       ;(19)Результат преобразования предыдущих
                          ;введенных цифр
        mov   BX,10       ;(20)Множитель 10
        mul   BX          ;(21)AX=предыдущий результат * 10
        add   AX,CX       ;(22)Добавим новую цифру к старому числу
        mov   DI,AX       ;(23)Сохраним в регистре DI
        jmp   inpt        ;(24)На ввод следующей цифры
done:   mov   AX,DI       ;(25)Загрузим результат в AX
```

Программа прежде всего выводит на экран запрос в виде знака ">" (предложения 1...3). Перед началом ввода очищается регистр DI, где будет накапливаться результат ввода последовательных цифр исходного числа. Вызовом функции DOS 08h ставится запрос на ввод с клавиатуры одного символа без отображения его на экране (отображать мы будет только цифры). Функция 08h возвращает введенный символ в AL. Код ASCII нажатой клавиши сравнивается с кодом ASCII клавиши <Enter> (код 13, предложение 7). Если нажата эта клавиша, процесс ввода цифр заканчивается переходом на метку done. При любой другой клавише выполняется анализ введенного кода. Содержимое AL сравнивается с символьным представлением цифры 9 (предложение 9). Если введенный код больше '9', он не является цифрой и его следует отбросить. Эту ситуацию отрабатывает команда ja, осуществляя переход на начало блока ввода очередного символа. Если введенный код прошел проверку на верхнюю границу, он сравнивается с нижней границей - кодом '0' (предложение 11). Команда jb осуществляет переход на начало блока ввода символа, если введен символ с кодом меньше '0'. Следующее предложение 13 будет выполняться, только если нажата клавиша с цифрой. Функцией DOS 02h введенный символ выводится на экран (предложения 13...15) и начинается его преобразование в число.

В предложении 17 с помощью команды xor выполняется очистка старшего байта регистра AX. Вообще команда xor (exclusive or, исключающее ИЛИ) анализирует биты обоих операндов (которые в данном случае совпадают) и устанавливает биты результата по следующему

правилу: каждый бит результата устанавливается в 1, если соответствующие биты операндов различаются, и сбрасывается в 0, если соответствующие биты операндов совпадают (рис. 35.1).

```
Первый операнд          0101 0011 0000 1111 (двоичное представление)
Второй операнд          1110 0011 1111 0000 (двоичное представление)
-------------------------------------------------------------------
Результат (замещает 1011 0000 1111 1111 (двоичное представление)
первый операнд)
```

Рис. 35.1. Результат действия команды хог.

В предложении 17 в команде хог оба операнда совпадают. В этом случае независимо от первоначального содержимого операнда после выполнения команды хог в нем будет 0 (рис. 35.2).

```
Первый операнд          0101 0011 0000 1111
Второй операнд          0101 0011 0000 1111
-------------------------------------------
Результат               0000 0000 0000 0000
```

Рис. 35.2. Результат действия команды XOR над совпадающими операндами.

Команда хог reg,reg занимает в памяти меньше места, чем команда mov reg,0, что и объясняет использование этого не очень наглядного приема.

Обнулив старший байт регистра AX и имея в его младшем байте введенную десятичную цифру в виде двоичного числа, мы на время сохраняем содержимое AX в регистре CX (предложение 18). Далее результат преобразования предыдущих цифр (или 0, если вводится первая цифра) извлекается из ячейки number и умножается на 10 (предложения 19...21). Команда mul (multiplication, умножение) предполагает наличие одного из сомножителей в AX. В качестве другого сомножителя не может выступать число (непосредственный операнд), поэтому сомножитель 10 мы занесли в регистр BX. Результат умножения процессор заносит в два регистра: в DX (старшую половину результата) и в AX (младшую половину). В нашем случае результат умножения не может превысить 64K, поэтому содержимым DX мы не интересуемся.

Получив в AX результат преобразования предыдущих цифр, умноженный на 10, мы прибавляем к нему текущую цифру из регистра CX, отправляем результат в ячейку number и переходим на метку inpt с целью ввода следующей цифры.

При обнаружении кода клавиши <Enter> выполняется переход на метку done, полученное число заносится в регистр AX и программа на этом завершается.

Рассмотренная программа неудачна в том отношении, что результат ее выполнения теряется. Проверить ее работоспособность можно только

с помощью отладчика, поставив точку останова вслед за предложением 25 и анализируя содержимое регистра AX после ввода всех цифр и перехода на метку done. Воспользовавшись программным материалом, полученным при изучении предыдущих статей, нетрудно усовершенствовать эту программу, дополнив ее строками преобразования числа, образованного в ячейке number, в символьную форму и вывода полученной строки символов на экран.

Включим в сегмент данных описание строки, выводимой на экран, например, в виде

```
string  db      10,13,'****',10,13,'$'
```

В конце программы, сразу после предложения 25, вызовем подпрограмму binasc, настроив предварительно регистр SI на адрес выходного поля данных:

```
        mov     SI,string+2
        call    binasc
```

После этого нетрудно с помощью функции 09h DOS вывести строку string на экран. Естественно, надо побеспокоиться о том, чтобы подпрограмма binasc и вызываемая ею подпрограмма hexasc были либо включены в состав исходного текста нашей программы, либо подключены к выполнимому модулю на этапе компоновки. Все эти вопросы рассматривались в статьях 21-22.

Еще один недостаток программы заключается в том, что в ней не анализируется число введенных символов. Между тем в машинное слово нельзя записать число, превышающее 65535, поэтому ввод шестого и последующих десятичных цифр не имеет смысла. Для того, чтобы ограничить число вводимых законных десятичных цифр и, в то же время, не учитывать случайно нажатые алфавитные и прочие клавиши, счетчик проходов программы следует установить в той точке, где начинается обработка законной десятичной цифры, т.е. между предложениями 12 и 13. Счетчик можно организовать следующим образом.

Где-либо до метки inpt надо инициализировать счетчик проходов, в качестве которого удобно использовать какой-либо свободный регистр, например, SI

```
        mov     SI,5
```

а после предложения 12 включить в программу строки изменения и анализа содержимого счетчика:

```
        dec     SI
        jl      inpt
```

Команда jl (jump if less, переход, если меньше) не срабатывает, пока в SI положительное число или 0. Однако как только после 5 проходов

(т.е. ввода 5 законных десятичных цифр) число в SI станет меньше 0, команда jl будет после ввода любого символа передавать управление назад на метку inpt, блокируя тем самым вывод символов на экран и их преобразование в числа. В этой ситуации программа будет ожидать нажатия клавиши <Enter>, чтобы перейти на строки завершения.

Создав и отладив программу в последнем варианте, вы получите весьма полезный специализированный калькулятор: он переводит вводимые с клавиатуры десятичные числа в шестнадцатеричные.

Статья 36

Знаковые и беззнаковые числа и операции

Выше уже отмечалось, что в машинное слово или регистр можно записать 64K разных чисел в диапазоне от 0000h до FFFFh, или от 00000 до 65535. Это справедливо в том случае, если мы все возможные числа рассматриваем, как положительные (беззнаковые). Если, однако, мы хотим в какой-то программе работать как с положительными, так и с отрицательными числами, т.е. с числами со знаком, нам придется часть чисел из их полного диапазона (наиболее естественно - половину) считать отрицательными. В вычислительной технике принято отрицательными считать все числа, у которых установлен старший бит, т.е. числа в диапазоне 8000h-FFFFh. Положительными же считаются числа со сброшенным старшим битом, т.е. числа в диапазоне 0000h-7FFFh. При этом отрицательные числа записываются в дополнительном коде, который образуется из прямого путем замены всех нулей единицами и наоборот (обратный код) и прибавления к полученному числу единицы (рис. 36.1).

```
Прямой код числа 5, т.е. число +5:          0000 0000 0000 0101
Обратный код числа 5:                       1111 1111 1111 1010
                                                              +1
-------------------------------------------------------------------
Дополнительный код числа 5, т.е. число -5:  1111 1111 1111 1011
```

Рис. 36.1. Образование отрицательного числа.

Следует подчеркнуть, что знак числа условен. Одно и то же число FFFBh, изображенное в нижней строке рис. 36.1, можно в одном кон-

тексте рассматривать, как положительное (+65531), а в другом - как отрицательное (-5). Таким образом, знак числа является характеристикой не самого числа, а способа его обработки.

На рис. 36.2 представлена выборочная таблица 16-битовых чисел с указанием их знаковых и беззнаковых значений.

16-ричное представл.	Десятичное представление: беззнаковое	знаковое	
0000h	00000	+00000	Нуль
0001h	00001	+00001	Минимальное положительное число
0002h	00002	+00002	
0003h	00003	+00003	
...			Диапазон положительных чисел
7FFDh	32765	+32765	
7FFEh	32766	+32766	
7FFFh	32767	+32767	Максимальное положительное число
8000h	32768	-32768	Максимальное отрицательное число
8001h	32769	-32767	
8002h	32770	-32766	
...			Диапазон отрицательных чисел
FFFCh	65532	-00004	
FFFDh	65533	-00003	
FFFEh	65534	-00002	
FFFFh	65535	-00001	Минимальное отрицательное число

Рис. 36.2. Представление знаковых и беззнаковых чисел в 16-разрядном компьютере.

Процессор может выполнять операции не только над словами, но и над байтами. Как и в случае целых слов, число в байте можно рассматривать, как беззнаковое, и тогда оно может принимать значения от 000 до 255, или как число со знаком, и тогда диапазон положительных значений уменьшается в два раза (от 000 до 127), но возникает возможность записать столько же отрицательных чисел (от -001 до -128).

На рис. 36.3 представлена выборочная таблица байтовых (8-битовых) чисел с указанием их знаковых и беззнаковых значений.

При операциях с числами следует иметь в виду явление оборачивания, которое можно кратко выразить такими соотношениями:

```
FFFFh+0001h=0000h
0000h-0001h=FFFFh
```

Если последовательно увеличивать содержимое регистра (кроме сегментного) или ячейки памяти, то, достигнув верхнего возможного предела FFFFh, число "перевалит" через эту границу, станет равным нулю и продолжит нарастать в области малых положительных чисел (1, 2, 3 и т.д.). Точно также, если последовательно уменьшать некоторое положительное число, оно достигнув нуля, перейдет в область отрицательных (или, что то же самое, больших беззнаковых) чисел, проходя значения 2, 1, 0, FFFFh, FFFEh и т.д.

16-ричное представл.	Десятичное представление: беззнаковое	знаковое	
00h	000	+000	Нуль
01h	001	+001	Минимальное положительное число
02h	002	+002	
03h	003	+003	
04h	004	+004	
05h	005	+005	Диапазон положительных чисел
...			
7Dh	125	+125	
7Eh	126	+126	
7Fh	127	+127	Максимальное положительное число
80h	128	-128	Максимальное отрицательное число
81h	129	-127	
82h	130	-126	
83h	131	-125	
...			Диапазон отрицательных чисел
FBh	251	-005	
FCh	252	-004	
FDh	253	-003	
FEh	254	-002	
FFh	255	-001	Минимальное отрицательное число

Рис. 36.3. Представление знаковых и беззнаковых байтовых чисел.

Отсюда, между прочим, следует, что если при последовательном наращивании относительного адреса в сегменте данных (что обычно требуется при работе с массивами) перейти границу беззнакового представления чисел, то начнут адресоваться ячейки не за пределами нашего сегмента данных, а из самого его начала.

Среди команд процессора, выполняющих ту или иную обработку чисел, можно выделить команды, индифферентные к знаку числа (например, inc, dec, test), команды, предназначенные для обработки беззнаковых чисел (mul, div, ja, jb и др.), а также команды, специально рассчитанные на обработку чисел со знаком (imul, idiv, jg, jl и т.д.).

Рассмотрим в качестве примера команды умножения. Их две: mul multiplication, умножение) для умножения беззнаковых чисел и imul (integer multiplication, целочисленное умножение) для работы со знаковыми числами. Результаты их выполнения при одних и тех же операндах могут радикально различаться.

Обе команды могут работать как со словами, так и с байтами. Они выполняют умножение числа, находящегося в регистре AX (в случае умножения на слово) или AL (в случае умножения на байт), на операнд, который может находиться в каком-либо регистре или в ячейке памяти. Не допускается умножение на непосредственное значение, а также на содержимое сегментного регистра.

Размер произведения, т.е. число байтов в нем, всегда в два раза больше размера сомножителей. Для однобайтовых операций полученное произведение записывается в регистр AX. Для двухбайтовых операций результат умножения, который имеет размер 32 бит, записывается в регистры DX:AX (в DX - старшая половина, в AX - младшая).

Рассмотрим несколько конкретных примеров действия команд знакового и беззнакового умножения.

```
mov    AL,3        ;Первый сомножитель=003
mov    BL,2        ;Второй сомножитель=002
mul    BL          ;AX=0006h=00006
mov    AL,3        ;Первый сомножитель=003
mov    BL,2        ;Второй сомножитель=002
imul   BL          ;AX=0006h=+00006
```

Обе команды, mul и imul, дают в данном случае одинаковый результат, поскольку знаковые положительные числа совпадают с беззнаковыми. Обратите внимание на то, что результат умножения, будучи в данном случае небольшим, занимает тем не менее весь регистр AX, затирая его старший байт.

```
mov    AL,0FFh     ;Первый сомножитель=255
mov    BL,2        ;Второй сомножитель=002
mul    BL          ;AX=01FEh=00510
mov    AL,0FFh     ;Первый сомножитель=-001
mov    BL,2        ;Второй сомножитель=002
imul   BL          ;AX=FFFEh=-00002
```

Здесь действие команд mul и imul над одними и теми же операндами дают разные результаты. В первом примере беззнаковое число FFh, которое интерпретируется, как десятичное 255, умножается на 2, давая в результате беззнаковое 00510, или 01FEh. Во втором примере то же число FFh рассматривается, как знаковое. В этом случае оно составляет -001. Умножение на 2 дает -002, или FFFEh.

Разная интерпретация одного и того же числа FFh обусловлена использованием в первом случае команды для обработки беззнаковых чисел, а во втором - знаковых.

Аналогичная ситуация возникает и при умножении целых слов:

```
mov    AX,0FFFFh   ;Первый сомножитель=65535
mov    BL,2        ;Второй сомножитель=00002
mul    BL          ;DX:AX=0001h:FFFEh=0000131070
mov    AL,0FFFFh   ;Первый сомножитель=-00001
mov    BL,2        ;Второй сомножитель=00002
imul   BL          ;DX:AX=FFFFh:FFFEh=-0000000002
```

В первом примере беззнаковое число FFFFh (десятичное 65535) умножается на 2, давая в результате беззнаковое 0000131070, или 0001FFFEh. Это 32-битовое число размещается в двух регистрах. Старшая половина числа (0001h) записывается в регистр DX, затирая его

предыдущее содержимое, младшая половина (FFFEh) в регистр AX. Во втором примере то же число FFFFh рассматривается, как знаковое. В этом случае оно составляет -00001. Умножение на 2 дает -0000000002, или FFFFFFFEh. По-прежнему старшая половина этого числа (FFFFh) записывается в DX, младшая половина (FFFEh) - в AX.

Другая важная группа команд, в которой различаются команды обработки знаковых и беззнаковых чисел - это команды условных переходов. Эти команды позволяют осуществлять переходы на различные метки программы в зависимости от результата выполнения предыдущей команды или, в некоторых случаях, одной из предшествующих команд. Команды условных переходов часто используют после команд сравнения (cmp), инкремента (inc), декремента (dec), сложения (add), вычитания (sub), проверки (test) и ряда других.

Приведем перечень команд условных переходов, чувствительных к "знаковости" числа.

Знаковые команды

jg (jump if greater, переход, если больше)
jge (jump if greater or equal, переход, если больше или равно)
jl (jump if less, переход, если меньше)
jle (jump if less or equal, переход, если меньше или равно)
jng (jump if not greater, переход, если не больше)
jnge (jump if переход, если не больше и не равно)
jnl (jump if переход, если не меньше)
jnle (jump if переход, если не меньше и не равно)

Беззнаковые команды

ja (jump if above, переход, если выше)
jae (jump if above or equal, переход, если выше или равно)
jb (jump if below, переход, если ниже)
jbe (jump if below or equal, переход, если ниже или равно)
jna (jump if not above переход, если не выше
jnae (jump if not above or equal, переход, если не выше и не равно
jnb (jump if not below, переход, если не ниже
jnbe (jump if not below or equal, переход, если не ниже и не равно

Примеры команд, нечувствительных к знаку числа

je (jump if equal, переход, если равно)
jne (jump if not equal, переход, если не равно)
jc (jump if carry, переход, если флаг CF установлен)
jcxz (jump if CX=0, переход, если CX=0)

Разница между знаковыми и беззнаковыми командами условных переходов заключается в том, что знаковые команды рассматриваютпонятия "больше-меньше" применительно к числовой оси -32K...0...+32K, а беззнаковые команды - применительно к числовой оси 0...64K.

Поэтому для команд знаковых переходов число 7FFFh (+32767) больше числа 8000h (-32768), а для команд беззнаковых 7FFFh (32767) меньше, чем 8000h (32768). Аналогично для знаковых команд 0 больше, чем FFFFh (-1), а для беззнаковых - меньше.

Из приведенного перечня видно, что при сравнении знаковых чисел используются термины "больше" и "меньше", а при сравнении беззнаковых - "выше" и "ниже".

Все сказанное справедливо и в том случае, когда команды условных переходов используются для анализа содержимого байтовых операндов. Так, для знаковых команд 7Fh (+127) больше, чем 80h (-128), а 0 больше, чем FFh (-1), а для беззнаковых команд - наоборот.

Рассмотрим несколько типичных примеров использования команд условных переходов.

```
cmp     AX,limit    ;Сравнение AX и содержимого ячейки limit
jb      below       ;Переход на метку below, если AX меньше
                    ;содержимого ячейки limit в беззнаковой
                    ;шкале чисел
cmp     AX,limit    ;Сравнение AX и содержимого ячейки limit
jl      less        ;Переход на метку less, если AX меньше
                    ;содержимого ячейки limit в знаковой
                    ;шкале чисел
cmp     AL,'9'      ;Сравнение кода ASCII в AL с '9'
ja      not_num     ;Переход, если не цифра
dec     SI          ;Декремент счетчика в SI
jl      less_0      ;Переход на метку less_0, если
                    ;содержимое SI, уменьшаясь, перешло
                    ;через 0 и стало отрицательным
add     AX,BX       ;Сложение AX и BX
je      nul         ;Переход на nul, если сумма=0
jge     positiv     ;Переход на positiv, если сумма >=0
```

Статья 37

Обработка двоично-десятичных чисел

В ряде прикладных областей для представления чисел используется специальный формат, называемый двоично-десятичным (binary-coded decimal, BCD). В таком формате выдают данные некоторые измерительные приборы; он же используется КМОП-микросхемой компьютеров IBM PC/AT для хранения информации о текущем времени. В микропроцессорах 80x86 предусмотрен ряд команд для обработки таких чисел.

Двоично-десятичный формат существует в двух разновидностях: упакованный и распакованный. В первом случае в байте записывается двухразрядное десятичное число от 00 до 99. Каждая цифра числа занимает половину байта и хранится в двоичной форме. Из рис. 37.1 можно заметить, что для записи в байт десятичного числа в двоично-десятичном формате достаточно сопроводить записываемое десятичное число символом h.

```
1001 0111  Двоичное содержимое байта
  9    7   Десятичное обозначение числа
  9   7h   16-ричное обозначение числа
```

Рис. 37.1. Упакованный двоично-десятичный формат.

В машинном слове или в 16-разрядном реги-стре можно хранить в двоично-десятичном фор-мате четырехразрядные десятичные числа от 0000 до 9999 (рис.37.2).

```
0001 1001 1001 0010   Двоичное содержимое слова
  1    9    9    2   Десятичное обозначение числа
  1    9    9   2h   16-ричное обозначение числа
```

Рис. 37.2. Запись десятичного числа 1992 в слове.

Распакованный формат отличается от упакованного тем, что в каждом байте записывается лишь одна десятичная цифра (по-прежнему в двоичной форме). В этом случае в слове можно записать десятичные числа от 00 до 99 (см. рис. 37.3)

```
0000 0001 0000 1000  Двоичное содержимое слова
     1         8     Десятичное обозначение числа
  0    1    0   8h  16-ричное обозначение числа
```

Рис. 37.3. Запись десятичного числа 18 в распакованном виде.

При хранении десятичных чисел в аппаратуре обычно используется более экономный упакованный формат; умножение и деление выполняются только с распакованными числами, операции же сложения и вычитания применимы и к тем, и к другим.

Разработаем процедуры для выполнения арифметических операций с информацией о текущем времени, получаемой из часов реального времени (КМОП-микросхема) компьютера. В дальнейшем с помощью этих процедур будет построена программа будильника.

В КМОП-микросхеме время хранится в упакованном десятичном формате по две десятичных цифры в одном байте. Таким образом, для записи часов, минут и секунд требуется три байта. Байты секунд и минут могут содержать числа от 00 до 59; байт часов - от 00 до 23. Особенность арифметических операций с временем заключается в том, что признак переноса в следующий разряд времени (т.е. из разряда секунд в разряд минут или из разряда минут в разряд часов) должен возникать при превышении значения 59. Описанная ниже подпрограмма add_unit позволяет складывать секунды, минуты или часы в упакованном двоично-десятичном формате и формировать признак переноса, если результат сложения превышает 59. В этом примере, помимо двоично-десятичной арифметики, демонстрируются распространенные приемы передачи в подпрограмму параметров через стек и получения однобитового параметра через флаг переноса CF.

Пример 37.1. 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          ;(1)Сохраним используемые
        push   BX          ;(2)регистры
        mov    AH,0        ;(3)Подготовим AH
        mov    BP,SP       ;(4)Настроим базовый регистр
        mov    BX,[BP+6]   ;(5)Получим параметр
        add    AL,BL       ;(6)Сложим слагаемые
        daa                ;(7)как BCD
        jnc    less100     ;(8)Сумма меньше 100
        mov    AH,1        ;(9)Сумма больше 100, запишем 01h в AH
less100:cmp    AX,60h      ;(10)Надо ли корректировать следующий
                           ;разряд времени?
        jb     done        ;(11)Нет, сумма < 60
        sub    AX,60h      ;(12)Да, сумма > 60, вычтем 60
        das                ;(13)как BCD
        stc                ;(14)Установим CF, как признак
        jmp    done1       ;(15)переноса и на выход
done:   clc                ;(16)Сбросим CF (переноса нет)
done1:  pop    BP          ;(17)Восстановим используемые
        pop    BX          ;(18)регистры
        ret    2           ;(19)Скорректируем указатель стека
add_unit endp
```

Всякая подпрограмма общего назначения должна иметь четко определенный интерфейс, т.е. правила передачи в нее параметров и получения результата. Подпрограмма не должна разрушать содержимое каких-либо регистров или ячеек памяти, если же такое происходит, то это также должно быть оговорено.

Подпрограмма add_unit оформлена в виде процедуры и не содержит (и не использует) каких-либо полей данных. Предполагается, что

читатель, отладив эту подпрограмму, запишет ее в объектную библиотеку, как это описывалось в статье 22, поскольку подпрограмма будет использоваться в дальнейших примерах.

В предложениях 1-2 сохраняются в стеке используемые в подпрограмме регистры. Далее обнуляется регистр AH, так как если результат сложения в регистре AL превышает десятичное число 99, то он требует для хранения уже не двух, а трех полубайтов.

В предложении 4 в регистр BP заносится текущее содержимое указателя стека SP. Это позволяет извлечь из стека второе слагаемое, которое согласно оговоренному интерфейсу передается через стек. К рассматриваемому моменту состояние стека соответствует рисунку 37.4.

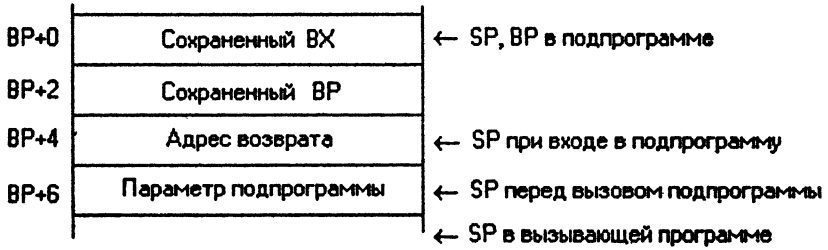


Рис. 37.4. Состояние стека после вызова подпрограммы add_unit и сохранения в стеке регистров BX и BP.

Командой mov BX,[BP+6] параметр подпрограммы загружается в регистр AX. В команде используется базовая адресация со смещением (смещением является константа 6). Особенность данной команды состоит в том, что в качестве базового используется регистр BP (а не BX, SI или DI). В этом случае по умолчанию операнд извлекается не из сегмента данных, а из сегмента стека, т.е. фактически данная команда транслируется, как BX,SS:[BP+6]. Как видно из рис. 37.4, интересующий нас параметр имеет в стеке смещение 6 относительно текущей вершины стека, адрес которой всегда находится в SP, а в нашем случае и в BP.

В соответствии с оговоренным интерфейсом второе слагаемое находится в младшем байте извлеченного из стека слова, т.е. в регистре BL. В предложении 6 оба слагаемых складываются с помощью обычной команды сложения add, которая прибавляет указанный операнд к содержимому AL и помещает полученную сумму в AL, давая пока что неверный результат. Для преобразования результата сложения в правильное двоично-десятичное число используется специальная команда daa (decimal adjust AL register after addition, десятичная коррекция в регистре AL после сложения). Если, например, складывались двоично-

десятичные числа 35 и 6, т.е. 35h и 06h, то после выполнения команды add в AL будет число 3Bh, которое не является двоично-десятичным. Команда daa скорректирует его, превратив в число 41h - правильную двоично-десятичную сумму исходных слагаемых. Максимальное значение суммы, которое может поместиться в байтовом регистре AL, составляет 99. Однако при сложении двух двухразядных десятичных чисел сумма может стать трехразрядной. Если в результате коррекции обнаружилось, что сумма составляет 100 или больше, команда daa устанавливает флаг переноса CF. В этом случае мы сами должны где-то сохранить образовавшийся третий десятичный разряд. В рассматриваемой программе он сохраняется в регистре AH (предложение 9). Если переноса не было, то команда jnc (jump if not carry, переход, если нет переноса), осуществляет переход на метку less100, обходя предложение 9, и в регистре AH остается нулевое значение.

Теперь мы должны выяснить, не получилось ли в результате сложения число, равное или большее 60. Этот анализ выполняется с помощью команды cmp (предложение 10). Если содержимое AX, рассматриваемое как беззнаковое число, меньше 60h, команда jb передает управление на метку done. Командой clc (clear carry flag, сброс флага переноса) сбрасывается флаг CF, восстанавливаются сохраненные регистры и осуществляется выход из подпрограммы. Если в результате сравнения выяснилось, что полученное число превышает 59, нам надо выполнить две операции. Во-первых, надо узнать число единиц времени данного разряда, для чего следует вычесть из полученной суммы 60 (если результат сложения равен 62 секундам, он составляет 1 минуту 2 секунды). Во-вторых, в соответствии с интерфейсом, надо при выходе из подпрограммы установить флаг CF.

Вычитание двоично-десятичных чисел выполнятся, как и сложение, в два этапа. Сначала командой sub осуществляется обычное вычитание, которое в случае двоично-десятичных чисел дает неверный результат. Затем полученный результат корректируется с помощью команды das (decimal adjust AL register after subtraction, десятичная коррекция регистра AL после вычитания). Строго говоря, здесь надо было бы проверять состояние флага CF, который устанавливается, если при вычитании потребовался заем из старшего разряда, однако в нашем случае такая ситуация возникнуть не может. В предложении 14 командой stc (set carry flag, установка флага переноса) устанавливается флаг CF, что проинформирует вызывающую программу о необходимости скорректировать следующий разряд времени (минуты или часы), после чего командой jmp осуществляется переход на строки завершения подпрограммы.

Рассматриваемая подпрограмма завершается несколько необычной командой возврата - ret с числовым параметром. Такая команда используется в тех случаях, когда параметры передаются в подпрограмму на

стеке, как это у нас и имеет место. Если после освобождения стека от сохраненных там регистров BP и BX выполнить команду ret, то верхнее слово стека (адрес возврата) будет загружен в указатель команд IP, в результате чего управление вернется в вызывающую программу. Однако на стеке останется параметр подпрограммы, который надо будет снимать со стека в вызывающей программе. Команда ret с числовым параметром как раз это и делает, освобождая вызывающую программу от забот о стеке. Числовой параметр должен быть, очевидно, кратен двум и его значение зависит от количества параметров, передаваемых в подпрограмму. При двух параметрах он будет равен 4, при трех - 6 и т.д.

На основе подпрограммы add_unit создадим подпрограмму следующего уровня add_time для сложения числовых значений времени, выраженных в виде трех двоично-десятичных чисел - часов, минут и секунд. Эта подпрограмма позволит нам выполнять арифметические операции со значениями текущего времени, полученными из КМОП-микросхемы. Будем считать, что при вызове подпрограммы число часов находится в регистре CH, число минут - в CL и число секунд - в DH. Пусть группа вторых слагаемых (тоже часы, минуты и секунды в двоично-десятичном формате) находится в памяти в трехбайтовом массиве, адрес которого перед вызовом подпрограммы записывается в регистры DS:SI. Поскольку подпрограмма add_time имеет довольно сложный алгоритм, рассмотрим сначала ее блок-схему (рис. 37.5).

На схеме обозначения CH, CL и DH относятся к соответствующим регистрам с числами часов, минут и секунд; hh, mm и ss обозначают элементы массива со второй группой слагаемых (часов, минут и секунд, соответственно). Блоки схемы, содержащие операции сложения, фактически представляют собой вызовы подпрограммы add_unit. На блок-схеме указаны только основные операции; операции сохранения регистров, извлечения из массива собственных параметров и подготовки параметров для подпрограммы add_unit опущены.

Подпрограмма начинается со сложения содержимого регистра DH с числом секунд ss из массива входных параметров путем вызова подпрограммы add_unit с соответствующими параметрами. Если после возврата из подпрограммы add_unit флаг CF сброшен, можно переходить к сложению следующего разряда времени - минут. Если флаг CF установлен, это значит, что суммарное число секунд превысило 59 и требуется добавить 1 к числу минут в регистре CL. Однако добавление 1 может увеличить исходное число минут до 60, поэтому после возврата из add_unit требуется проверить состояние флага CF. Если флаг CF сброшен, можно переходить к сложению следующего разряда времени - минут. Если флаг CF установлен, это значит, что суммарное число минут превысило 59 и требуется добавить 1 к числу часов в регистре CH, после чего по-прежнему перейти к сложению минут.

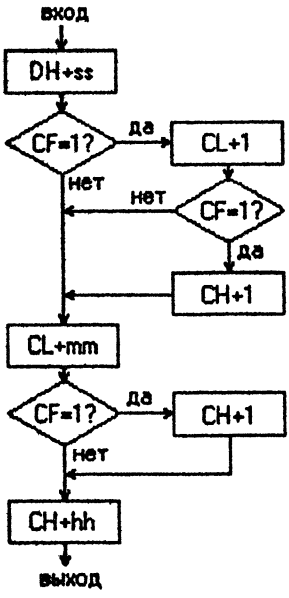


Рис. 37.5. Блок-схема подпрограммы сложения значений времени.

Число минут в регистре CL складывается с числом минут mm из массива параметров, после чего проверяется состояние флага CF. Если флаг CF сброшен, можно переходить к сложению следующего разряда времени - часов. Если флаг CF установлен, это значит, что суммарное число минут превысило 59 и требуется добавить 1 к числу часов в регистре CH, после чего по-прежнему перейти к сложению часов.

Число часов в регистре CH складывается с числом часов hh из массива параметров. Вообще говоря, для сложения часов следовало предусмотреть специальную процедуру, так как перенос в следующий разряд времени (дни) должен осуществляется после превышения не 59, а 23. Мы для простоты опустили этот анализ, лишив себя возможности устанавливать будильник на следующие сутки.

Рассмотрим теперь полный текст подпрограммы add_time (пример 37.2). Поскольку он точно соответствует приведенной ранее блок-схеме, остановимся только на неясных или любопытным моментах.

Пример 37.2. Подпрограмма сложения трех разрядов времени (часов, минут и секунд) в двоично-десятичном формате

```
add_time proc
;При вызове:CH=часы, CL=минуты, DH=секунды (исходное время в BCD).
;Прибавляемое время в трехбайтовом массиве (часы, минуты, секунды
;в BCD), адрес массива в DS:SI. При возврате в тех же регистрах
;результат сложения в том же формате
        push    AX          ;(1)Сохраним
        push    BX          ;(2)используемые
        push    DI          ;(3)регистры
        mov     DI,1        ;(4)Подготовим 1 для переноса
;Сложим секунды
        mov     AL,DH       ;(5)Исходные секунды отправим в AL
        mov     BL,[SI+2]   ;(6)Получим прибавляемые секунды
        mov     BH,0        ;(7)и в виде слова
        push    BX          ;(8)отправим в стек
        call    add_unit    ;(9)Вызов подпрограммы сложения
        mov     DH,AL       ;(10)Результат назад в DH
        jnc     mm          ;(11)Переноса нет, на сложение минут
;Перенос, прибавим 1 к минутам
        mov     AL,CL       ;(12)Исходные минуты отправим в AL
        push    DI          ;(13)Прибавляемую 1 в стек
```

```
        call    add_unit    ;(14)Вызов подпрограммы сложения
        mov     CL,AL       ;(15)Результат назад в CL
        jnc     mmm         ;(16)Переноса нет, на сложение минут
;Перенос, прибавим 1 к часам
        mov     AL,CH       ;(17)Исходные часы отправим в AL
        push    DI          ;(18)Прибавляемую 1 в стек
        call    add_unit    ;(19)Вызов подпрограммы сложения
        mov     CH,AL       ;(20)Результат назад в CH
;Сложим минуты
mmm:    mov     AL,CL       ;(21)Исходные минуты отправим в AL
        mov     BL,[SI+1]   ;(22)Получим прибавляемые минуты
        mov     BH,0        ;(23)и в виде слова
        push    BX          ;(24)отправим в стек
        call    add_unit    ;(25)Вызов подпрограммы сложения
        mov     CL,AL       ;(26)Результат назад в CL
        jnc     hhh         ;(27)Переноса нет, на сложение часов
;Перенос, прибавим 1 к часам
        mov     AL,CH       ;(28)Исходные часы отправим в AL
        push    DI          ;(29)Прибавляемую 1 в стек
        call    add_unit    ;(30)Вызов подпрограммы сложения
        mov     CH,AL       ;(31)Результат назад в CH
;Сложим часы
hhh:    mov     AL,CH       ;(32)Исходные часы отправим в AL
        mov     BL,[SI]     ;(33)Получим прибавляемые часы
        mov     BH,0        ;(34)и в виде слова
        push    BX          ;(35)отправим в стек
        call    add_unit    ;(36)Вызов подпрограммы сложения
        mov     CH,AL       ;(37)Результат назад в CH
;Восстановим используемые регистры
        pop     DI          ;(38)
        pop     BX          ;(39)
        pop     AX          ;(40)
        ret                 ;(41)
add_time endp
```

Регистр DI (предложение 4) используется для хранения числа 1, которое надо будет прибавлять к разрядам времени в случае возникновения переносов. Перед каждым вызовом подпрограммы add_unit для нее подготавливаются параметры в соответствии с рассмотренным ранее интерфейсом. Так, при сложении секунд в регистр AL заносится первое слагаемое (предложение 5), затем готовится второй параметр. Второе слагаемое забирается в промежуточный регистр BX из массива параметров с помощью индексной адресации со смещением (предложение 6), старшая половина BX обнуляется, и полученное число помещается в стек.

После возврата из подпрограммы add_unit полученная сумма секунд загружается в регистр DL на место первого слагаемого, после чего проверяется состояние флага CF. Такая последовательность операций возможна потому, что команда mov не изменяет состояние флагов процессора. В то же время сохранить результат сложения в регистре DL нам надо независимо от результатов проверки состояния флага CF. Такой

прием - проверка флага спустя несколько команд после той, которая на этот флаг воздействует - применяется в профессиональных программах весьма широко, не способствуя, естественно, ясности их текста.

Если флаг переноса сброшен, командой jnc управление передается на метку mmm, где выполняется сложение следующего разряда времени - минут; если же флаг CF установлен, настраиваются параметры для вызова подпрограммы add_unit,причем в качестве второго параметра выступает содержимое регистра DI, где хранится единица (предложения 12-13).

Мы считаем, что сложение часов всегда осуществляется в пределах текущих суток, поэтому после получения и сохранения суммы часов (предложения 36-37) осуществляется восстановление сохраненных регистров и выход из подпрограммы обычной командой ret.

Статья 38

Чтение текущего времени из КМОП-микросхемы

Компьютеры типа IBM PC/AT оснащаются КМОП-микросхемой с батарейным питанием, которая служит, во-первых, для хранения информации о конфигурации компьютера (количество и типы дисков, объем памяти и проч.), а во вторых - 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 (для данного сеанса работы на компьютере) времени и даты используются функции 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 време-

ни 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 символьную форму и выведя полученную строку на экран, мы получим программу-часы, которая при ее запуске будет выводить на экран текущее реальное время.

Рассмотрим сначала программу, преобразующую в символьную форму конкретное упакованное двухразрядное BCD-число, занимающее два полубайта. После преобразования оно должно занять два байта.

Пример 38.1. Преобразование двоично-десятичного числа в символьную форму

```
        mov   AL,35h      ;Исходное BCD-число 35
        push  AX          ;Сохраним его в стеке
        and   AL,0Fh      ;Выделим младший полубайт AL
        add   AL,'0'      ;Прибавим код ASCII '0'
        mov   num+3,AL    ;Отправим в выходную строку
        pop   AX          ;Восстановим исходное число
        mov   CL,4        ;Сдвинем вправо
        shr   AL,CL       ;на 4 разряда
        and   AL,0Fh      ;Выделим младший полубайт AL
        add   AL,'0'      ;Прибавим код ASCII '0'
        mov   num+2,AL    ;Отправим в выходную строку
        write num         ;Выведем все на экран
;Поля данных
num     db    'N=  $'
```

Исходное двухразрядное число находится в регистре AL. Содержимое AH роли не играет. Поскольку нам придется обращаться к AL за двумя разрядами дважды, весь регистр AX сохраняется в стеке. Далее с помощью команды and выделяется младший полубайт числа, т.е. цифра единиц (записанная в двоичной форме). После прибавления кода ASCII символа нуля (30h) в AL образуется код ASCII этой цифры, который пересылается в выходную строку num на соответствующее ему место.

После извлечения из стека исходного числа и сдвига его вправо на 4 двоичных разряда, чтобы цифра десятков переместилась в младший полубайт регистра AL, описанные операции повторяются для этой цифры.

Рассмотрим теперь получение из КМОП-микросхемы значения текущего времени и вывод его на экран. Поскольку значение времени включает 3 двухразрядных BCD-числа, описанную процедуру придется применить трижды, и ее полезно оформить в виде макрокоманды или подпрограммы. В примере 38.2 использована макрокоманда.

Пример 38.2. Преобразование двоично-десятичных чисел в символьную форму

```
;Макрокоманда для преобразования в символьную форму двухразрядного
;двоично-десятичного числа. Входные параметры:addr - относительный
;адрес выходной символьной строки, -offs - смещение в этой строке
tim_as  macro addr, offs ;Имя макрокоманды и формальные параметры
        push  CX         ;Сохраним используемый регистр
```

```
        push   AX           ;Сохраним в стеке исходное число
        and    AL,0Fh       ;Выделим младший полубайт AL
        add    AL,'0'       ;Прибавим код ASCII '0'
        mov    addr+offs+1,AL;Отправим в выходную строку
        pop    AX           ;Восстановим исходное число
        mov    CL,4         ;Сдвинем вправо
        shr    AL,CL        ;на 4 разряда
        and    AL,0Fh       ;Выделим младший полубайт AL
        add    AL,'0'       ;Прибавим код ASCII '0'
        mov    addr+offs,AL;Отправим в выходную строку
        pop    CX           ;Восстановим сохраненный регистр
        endm                ;Конец макроопределения
;Главная процедура
        mov    AH,02h       ;Функция получения времени
        int    1Ah          ;Прерывание BIOS
        mov    AL,CH        ;Заберем часы
        tim_as time,0       ;Преобразуем
        mov    AL,CL        ;Заберем минуты
        tim_as time,3       ;Преобразуем
        mov    AL,DH        ;Заберем секунды
        tim_as time,6       ;Преобразуем
        write  clock        ;Выведем строку на экран
;Поля данных
clock   db     'Текущее время '
time    db     0,0,':',0,0,':',0,0,'$'
```

Статья 39

Работа с видеобуфером

В этой статье на примере вывода текстовых данных в видеобуфер графического адаптера будет рассмотрен чрезвычайно важный вопрос обращения к ячейкам памяти с известными физическими адресами. До сих пор мы работали только с ячейками памяти внутри нашей программы. Эти ячейки могут располагаться как в сегменте данных, так и в сегменте команд. В исходном тексте программы им даются некоторые имена, по которым и происходит обращение к этим ячейкам в программных строках. Однако видеобуфер не входит в нашу программу, и у его ячеек нет никаких имен. В таких случаях обращение к памяти осуществляется по заданным физическим адресам. Для работы с видеобуфером необходимо знать, где в адресном пространстве он находится и какова его организация. Поэтому рассмотрим прежде всего некоторые элементы видеосистемы машин типа IBM PC.

Как уже отмечалось в статье 5, адресное пространство микропроцессоров, работающих в реальном режиме, составляет 1 Мбайт. В зависимости от модификации 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 приведена на рис. 39.1.

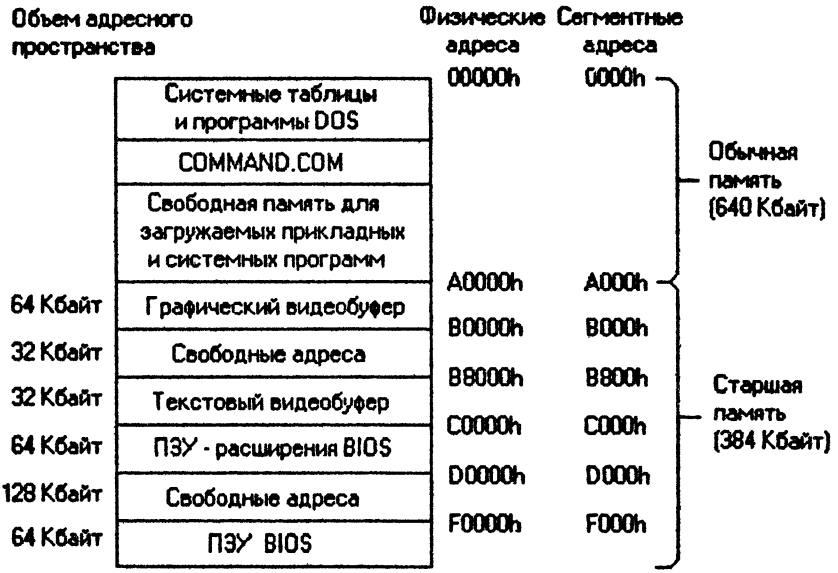


Рис. 39.1. Типичное распределение адресного пространства.

Первые 640 Кбайт адресного пространства с адресами от 00000h до 9FFFFh (и, соответственно, с сегментными адресами от 0000h до 9FFFh) отводятся под основную оперативную память, которую еще называют стандартной, или обычной. В начало этой области при начальной загрузке компьютера загружаются таблицы и программы DOS, которые обычно занимают несколько десятков Кбайт.

Вся оставшаяся память до границы 640 Кбайт свободна для загрузки любых системных или прикладных программ. Как правило, в начале сеанса в память загружают резидентные программы (русификатор, антивирусные программы). При наличии резидентных программ объем свободной памяти уменьшается.

Оставшиеся 384 Кбайт адресного пространства, называемого старшей памятью, первоначально были предназначены для размещения по-

стоянных запоминающих устройств (ПЗУ). Практически под ПЗУ занята только часть адресов. В самом конце адресного пространства, в области F0000h...FFFFFh располагается основное постоянное запоминающее устройство BIOS, а начиная с адреса C0000h - так называемое ПЗУ расширений BIOS для обслуживания графических адаптеров и дисков. Часть старшей памяти отводится для адресации к видеобуферам графического адаптера. Приведенное на рис. 39.1 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 по адресу B0000h).

Текстовый видеобуфер адаптера EGA включает 8 видеостраниц и занимает в адресном пространстве компьютера (за пределами обычной памяти) 32 Кбайт от сегментного адреса B800h. Начинается он с видеостраницы 0, адрес которой совпадает с адресом всего видеобуфера. Каждая страница занимает4 Кбайт, таким образом, страница 1 начинается с сегментного адреса B900h, страница 2 - с адреса BA00h и т.д. Весь видеобуфер простирается до границы (сегментной) C000h

При включении компьютера активной (видимой) становится видеостраница 0. Выводить текстовое изображение можно на любую страницу, в том числе невидимую. Смена видеостраниц осуществляется вызовом функции 05h прерывания 10h BIOS.

Любой код, записываемый в видеобуфер, сразу же отображается на экране в виде изображения цветного символа на одном из знакомест. Каждый символ занимает в буфере поле из двух байт (рис. 39.2.). Младшие (четные) байты всех полей отводятся под коды ASCII отображаемых символов, старшие (нечетные) байты - под их атрибуты. Соответствие атрибутов цветам приведено в табл. 39.1.

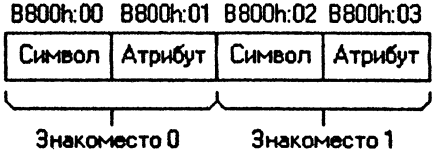


Рис.39.2. Логическая организация текстового видеобуфера.

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 симво-

лов. Трактовка же, например, кода ASCII 9, как символа табуляции, или кода ASCII 10 - как символа перевода строки выполняется программами DOS, которые в данном случае не активизируются.

Таблица 39.1. Коды цветов стандартной цветовой палитры EGA

Код Dec	Код Hex	Цвет	Код Dec	Код Hex	Цвет
0	0h	Черный	8	8h	Серый
1	1h	Синий	9	9h	Голубой
2	2h	Зеленый	10	Ah	Салатовый
3	3h	Бирюзовый	11	Bh	Светло-бирюзовый
4	4h	Красный	12	Ch	Розовый
5	5h	Фиолетовый	13	Dh	Светло-фиолетовый
6	6h	Коричневый	14	Eh	Желтый
7	7h	Белый	15	Fh	Ярко-белый

Для того, чтобы из программы получить доступ к видеобуферу, надо занести в один из сегментных регистров данных его сегментный адрес. После этого, задавая те или иные смещения, мы сможем выполнять запись в любые места видеобуфера.

Пример 39.1. Прямое программирование видеобуфера

```
include mac.mac             ;(1)Объявление макробиблиотеки
;Очистим экран с помощью макрокоманды write
;и Esc-последовательности clrscr
        write clrscr        ;(2)
;Настроим сегментный регистр ES на страницу 0 видеобуфера
        mov   AX,0B800h     ;(3)Сегментный адрес видеобуфера
        mov   ES,AX         ;(4)Загрузим его в ES
;Выведем два символа
        mov   BX,80*2*5     ;(5)Смещение в видеобуфере в байтах
        mov   AL,'*'        ;(6)Символ
        mov   AH,0Eh        ;(7)Цвет желтый по черному
        mov   ES:[BX],AX    ;(8)Запись в видеобуфер
        mov   ES:[BX+162],0B0Fh;(9)Цвет светло-бирюзовый по
                            ;черному, символ с кодом ASCII Fh
        stop                ;(10)Остановим программу
;Поля данных
clrscr  db 27,'[2J$' ;Esc-последовательность очистки экрана
```

Программа начинается с очистки экрана (предложение 2) с помощью соответствующей Esc-последовательности. Далее в сегментный регистр ES заносится сегментный адрес видеобуфера, т.е. его физический адрес, деленный на 16. Поскольку непосредственная запись адреса в сегментный регистр запрещена, эта операция осуществляется через промежуточный регистр AX (предложения 3-4). Для задания смещения (относительного адреса) удобно воспользоваться одним из базовых или индексных регистров; в данном случае используется регистр

BX. Учитывая, что в каждая строке экрана содержит 80 символов, а символ занимает 2 байт, выражение 80*2*5 (предложение 5) описывает смещение от начала видеобуфера первого байта строки 5. В регистр AL, соответсвующий первому (четному) байту знакоместа, записывается код ASCII требуемого символа; в регистр AH, соответствующий второму (нечетному) байту знакоместа - код атрибута. В предложении 9 полное описание символа (код ASCII и атрибут) пересылается в видеобуфер обычной командой mov.

В предложении 9 показан другой вариант команды записи. Здесь использована так называемая базово-индексная адресация со смещением. При таком способе адресации относительный адрес вычисляется процессором путем сложения содержимого регистра с указанным в коде команды числом (смещением). Таким образом удобно перемещаться по памяти на заданное число байт относительно некоторого базового смещения, хранящегося в базовом регистре. В предложении 9 эта "добавка" перемещает второй записываемый символ на одну полную строку экрана и еще одно знакоместо.

Другая особенность предложения 9 - в видеобуфер записывается коды символа и атрибута непосредственно в виде числа (0B0Fh). Это удобно в тех случаях, когда на экран выводятся символы, не имеющие закрепленных за ними клавиш. Код 0Fh соответствует изображению "солнышка".

В конце программы предусмотрена остановка до нажатия клавиши, чтобы выводимый на экран после завершения текущей программы запрос DOS не разрушал построенного нами изображения.

Как правило, в видеобуфер требуется записать не один - два символа, а значительно больший объем данных, вплоть до полной видеостраницы. Обычно выводимый на экран информационный кадр формируется заранее в буфере пользователя, располагающемся в сегменте данных программы. Таким образом, вывод на экран сводится к пересылке содержимого программного буфера в видеобуфер.

Как уже отмечалось, команда mov не может осуществлять пересылку из памяти в память. Для этого предусмотрены специальные команды movsb (move string byte, пересылка строки байтов) и movsw (move string word, пересылка строки слов). Эти команды пересылают по одному элементу строки, который может быть байтом или словом. Операнд-источник, т.е. исходная строка, адресуется через пару регистров DS:SI, операнд-приемник, т.е. строка назначения - через регистры ES:DI. Однократное выполнение команды пересылает один элемент строки, при этом после каждой операции пересылки регистры SI и DI получают положительное (если флаг DF регистра флагов процессора равен 0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1 или 2 в зависимости от размера пересылаемых элементов.

Для того, чтобы переслать целую строку, команда movsb или movsw предваряется префиксом повторения rep, который заставляет процессор выполнить команду CX раз.

Пример 39.2. Запись строки в видеобуфер

```
Очистим экран с помощью Esc-последовательности clrscr
        ...
;Настроим сегментный регистр ES на страницу 0 видеобуфера
        ...
;Перешлем в видеобуфер строку символов
;Для этого сначала настроим регистры SI, DI и CX
        mov     SI,offset msg;SI=адрес источкика
        mov     DI,80*2*12+37*2;DI=адрес приемника
        mov     CX,msglen  ;CX=число пересылаемых байтов
        cld                ;Сброс DF - вперед
rep     movsb              ;Пересылка в цикле
;Поля данных
clrscr  db 27,'[2J$' ;Esc-последовательность очистки экрана
msg     db 10h,0Eh,'T',84h,'e',84h,'c',84h,'т',84h,11h,0Eh
msglen=$-msg
```

Начало этого примера (очистка экрана, подготовка адресации видеобуфера) не отличается от предыдущего. далее выполняется настройка регистров для команды movsb. В регистр SI засылается адрес буфера с подготовленным текстом, в регистр DI - смещение в видеобуфере (на экране). Сдвиг относительно начала видеобуфера составляет 12 строк по 80 символов плюс 37 символов. Регистры DS и ES уже настроены должным образом: DS указывает на сегмент данных программы, в котором находится исходная строка, а в ES загружен сегментный адрес видеобуфера, куда строка будет пересылаться. В регистр CX заносится число пересылаемых байтов. Наконец, командой cld устанавливается направление пересылки строки - от указанных в регистрах SI и DI смещений вперед. В некоторых случаях операции со строками удобнее выполнять от конца строк. Тогда надо командой std (set direction flag, установка флага направления) установить направление назад.

После подготовки всех необходимых регистров выполняется команда movsb с префиксом rep.

Поскольку в видеобуфере коды ASCII отображаемых символов перемежаются с их атрибутами, пересылаемую в видеобуфер текстовую строку приходится формировать так же. При этом для каждого символа можно установить свой атрибут. Однако в целом такой метод описания текстовой строки слишком громоздок. В тех случаях, когда все символы строки имеют один и тот же атрибут, удобнее включение атрибутов в строку выполнять программно. Возможный алгоритм такого преобразования, в котором продемонстрировано использование еще двух широко используемых команд обработки строк, показан в примере 39.3.

Пример 39.3. Формирование строки для записи в видеобуфер

```
;Очистим экран с помощью Esc-последовательности
        ...
;Настроим сегментный регистр ES на страницу 0 видеобуфера
        ...
        mov    DI,80*2*5  ;(1)Начальное смещение на экране
;Будем переносить байт за байтом на экран, вставляя между
;кодами ASCII байты атрибута
        mov    CX,str_len ;(2)Длина обрабатываемой строки
        mov    SI,offset string;(3)Смещение исходной строки
        cld               ;(4)Вперед
        mov    AH,attr    ;(5)Атрибут
lodstor:lodsb             ;(6)Загрузили в AL очередной символ
        stosw             ;(7)Выгрузили символ+атрибут из AX
                          ;прямо в видеобуфер
        loop   lodstor    ;(8)Повторять str_len раз
;Остановим программу
        ...
;Поля данных
clrscr  db 27,'[2J$' ;Esc-последовательность очистки экрана
attr    db 1Eh ;Атрибут символов выходной строки
string  db 'Обращение к функциям DOS и BIOS осуществляется '
db 'с помощью механизма программных прерываний. Настроив нужным'
db ' образом регистры общего назначения процессора и выполнив '
db 'команду программного прерывания INT с соответствующим номером'
db ', пользователь активизирует требуемую функция DOS или BIOS.'
str_len=$-string
```

Преобразование исходной строки string выполняется в цикле по байтам с одновременным переносом формируемых слов в видеобуфер. После инициализации ES в регистр DI заносится начальное смещение на экране (предложение 1). В предложении 2 инициализируется счетчик цикла CX. Индексный регистр SI настраивается на исходную строку и устанавливается обработка строки вперед (предложения 3-4).

В младшем байте выводимого слова должен быть код ASCII, в старшем - атрибут символа. Эти слова формируются в регистре AX. Поскольку атрибут символов остается неизменным, он загружается в старший байт AX перед началом цикла (предложение 5).

Цикл формирования слов и переноса их в видеобуфер состоит всего из двух команд. Команда lodsb загружает в регистр AL содержимое ячейки по адресу, находящемуся в DS:SI. После выполнения операции SI получает положительное приращение и настраивается на следующий байт исходной строки. Старший байт AX уже заполнен. Следующая команда stosw переносит содержимое регистра AX в память видеобуфера по адресу, находящемуся в паре регистров ES:DI. После выполнения операции регистр DI получает положительное приращение, равное 2 и настраивается на следующее слово в видеобуфере. В конце программы предусмотрена остановка для наблюдения результатов ее выполнения.

Статья 40

Обработка символьных данных

Программа, описанная в предыдущей статье, выводит текст в видеобуфер байт за байтом, безотносительно к расположению в тексте слов или знаков препинания. В результате правая граница экрана может попадать в середину слова или числа, что затрудняет чтение. В настоящей статье мы рассмотрим программу, "вылавливающую" слова, пересекаемые границей экрана, и переносящую их на следующую строку. Такая программа является неотъемлемой частью любого текстового редактора. К сожалению, программа имеет значительную длину. Некоторым утешением может послужить наглядность получаемого результата. Функциональные фрагменты программы выделены в отдельные процедуры; это соответствует общепринятой методике написания сложных программ, существенно упрощает алгоритм основной процедуры и облегчает объяснения.

В примерах 39.2 и 39.3 было проиллюстрировано использование команд обработки строк movs, lods и stos. В настоящей статье эта группа команд будет дополнена командой сканирования строки scas, а также приведен пример обработки строки справа налево.

Пример 40.1. Обработка символьного массива

```
clear_screen   proc        ;(1)Начало процедуры
;Подпрограмма очистки экрана с помощью Esc-последовательности
         push   AX          ;(2)Сохраним используемые
         push   DX          ;(3)регистры
         mov    AH,09h      ;(4)Функция вывода на экран
         mov    DX,offset clrscr;(5)Адрес сообщения
         int    21h         ;(6)Вызов DOS
         pop    DX          ;(7)Восстановим используемые
         pop    AX          ;(8)регистры
         ret                ;(9)Возврат в вызывающую программу
clear_screen   endp        ;(10)Конец процедуры
search_blank   proc        ;(11)Начало процедуры
;Подпрограмма поиска последнего пробела в строке длиной 80
;символов. При вызове: ES:DI=адрес строки. При возврате:
;ES:DI=адрес байта за последним пробелом в строке, CX=число
;выводимых символов от начала строки до последнего пробела
         push   AX          ;(12)Сохраним используемые
         push   BX          ;(13)регистры
         mov    BX,DI       ;(14)Смещение начала строки
         add    DI,79       ;(15)DI->последний байт строки
```

```
        std                 ;(16)Обработка назад
        mov     CX,80       ;(17)Столько байтов просмотреть
        mov     AL,' '      ;(18)Ищем пробел
repne   scasb               ;(19)Сканирование строки от конца
                            ;DI->первый символ перед пробелом
        add     DI,2        ;(20)DI->символ справа от пробела
        mov     CX,DI       ;(21)Сохраним это выходное значение
        sub     CX,BX       ;(22)Получим длину выводимой строки
        pop     BX          ;(23)Восстановим используемые
        pop     AX          ;(24)регистры
        ret                 ;(25)Возврат в вызывающую программу
search_blank    endp        ;(26)Конец процедуры
mov_text proc               ;(27)Начало процедуры
;Подпрограмма включения атрибута между кодами ASCII и записи
;строки в видеобуфер. При вызове: DS:SI->начало выводимой строки,
;ES:DI->текущая позиция на экране, CX=число выводимых символов,
;AH=атрибут. При возврате: ES:DI->текущая позиция на экране за
;записанной строкой, CX тот же, что и при вызове (число выведенных
;символов, включая пробел)
        push    CX          ;(28)Сохраним число выводимых символов
        cld                 ;(29)Обрботка вперед
lodstor:lodsb               ;(30)Загрузим в AL очередной символ
        stosw               ;(31)Выгрузим из AX символ+атрибут
        loop    lodstor     ;(32)Повторять CX раз
        pop     CX          ;(33)Восстановим счетчик
        ret                 ;(34)Возврат в вызывающую программу
mov_text endp               ;(35)Конец процедуры
mov_blank proc              ;(36)Начало процедуры
;Подпрограмма дополнения оставшейся части строки экрана пробелами
;При вызове: ES:DI->текущая позиция на экране, CX=число выведенных
;символов в данной строке, AH=атрибут. При возврате:ES:DI->текущая
;позиция на экране за записанной строкой
        push    BX          ;(37)Сохраним используемый регистр
        mov     BX,80       ;(38)Число символов в строке экрана
        sub     BX,CX       ;(39)Столько надо вывести пробелов
        mov     CX,BX       ;(40)Отправим его в счетчик
        add     symbs,CX    ;(41)Добавим к общей длине текста
        mov     AL,' '      ;(42)Записываем пробел
        cld                 ;(43)Обработка вперед
rep     stosw               ;(44)Запись слова пробел+атрибут
        pop     BX          ;(45)Восстановим используемый регистр
        ret                 ;(46)Возврат в вызывающую программу
mov_blank endp              ;(47)Конец процедуры
;Главная процедура
main    proc                ;(48)
        mov     AX,data     ;(49)
        mov     DS,AX       ;(50)
        call    clear_screen;(51)Очистим экран
        mov     mem_seg,DS  ;(52)Сохраним сегмент данных
        mov     mem_offs,offset string;(53)Сохраним текущее
                            ;смещение нашего текста
;Настроим сегментный регистр ES на страницу 0 видеобуфера
        mov     AX,0B800h   ;(54)Сегментный адрес видеобуфера
        mov     ES,AX       ;(55)Загрузим его в ES
        mov     AX,80*2*5   ;(56)Начальное смещение на экране
        mov     scr_seg,ES  ;(57)Сохраним сегмент видеобуфера
        mov     scr_offs,AX;(58)Текущее смещение на экране
```

```
;Найдем последний пробел в строке
begin:  mov     ES,mem_seg ;(59)Настроим исходные параметры
        mov     DI,mem_offs;(60)для search_blank
        call    search_blank;(61)Вызов подпрограммы
;Выведем на экран строку до пробела
        mov     SI,mem_offs;(62)Параметр для mov_text
        mov     mem_offs,DI;(63)Новое значение текущего адреса
        mov     ES,scr_seg ;(64)Параметры для
        mov     DI,scr_offs;(65)подпрограмм
        mov     AH,attr    ;(66)mov_text и mov_blank
        call    mov_text   ;(67)Вызов подпрограммы
;Выведем пробелы
        call    mov_blank  ;(68)Вызов подпрограммы
        mov     scr_offs,DI;(68)Новое смещение на экране
;Учтем вывод 80 символов и будем повторять
;выполненные шаги до исчерпания текста
        sub     symbs,80   ;(70)Число оставшихся символов
        je      exit       ;(71)Если не осталось - на выход
        jmp     begin      ;(72)Если осталось - повторять
                           ;обработку и вывод строк
exit:   ...                ;(73)Остановим для наблюдения
;Поля данных
clrscr  db      27,'[2J$'  ;Очистка экрана
attr    db      1Eh        ;Атрибут символов
symbs   dw      str_len    ;Длина текста
scr_seg dw      0          ;Сегмент экрана
scr_offs dw     0          ;Смещение на экране
mem_seg dw      0          ;Сегмент текста
mem_offs dw     0          ;Смещение текста
string  db 'Обращение к функциям DOS и BIOS осуществляется '
db 'с помощью механизма программных прерываний. Настроив нужным'
db ' образом регистры общего назначения процессора и выполнив '
db 'команду программного прерывания INT с соответствующим номером'
db ', пользователь активизирует требуемую функция DOS или BIOS.'
db ' '                     ;Необходимый завершающий пробел
str_len=$-string
```

Программа 40.1, кроме главной процедуры main, включает четыре процедуры-подпрограммы: clear_screen, search_blank, mov_text и mov_blank.

Подпрограмма clear_screen очищает экран и переводит курсор в верхнюю левую позицию путем посылки на экран с помощью функции 09h DOS соответствующей Esc-последовательности.

Подпрограмма search_screen получив адрес входного текста, осуществляет поиск в 80-символьной строке, начинающейся с этого адреса, последнего пробела. Найдя последний пробел, подпрограмма возвращает адрес символа за этим пробелом, а также длину строки от указанного адреса до последнего пробела (включая этот пробел). В дальнейшем с помощью подпрограммы mov_text эта строка будет записана в видеобуфер и с помощью подпрограммы mov_blank дополнена пробелами до правого края экрана. В следующем шаге цикла поиск последнего пробела в очередной 80-символьной строке начнется от возвращен-

ного этой подпрограммой адреса, т.е. от начала первого невыведенного еще на экран слова текста.

В начале подпрограммы clear_screen сохраняются в стеке, а в конце подпрограммы извлекаются из стека два регистра (AX и BX), используемые подпрограммой. Полученный в качестве входного параметра относительный адрес анализируемой строки сохраняется на время в регистре BX (предложение 14). Прибавлением к содержимому DI числа 79 образуется адрес последнего байта 80-символьной строки. Командой std устанавливается флаг направления DF, определяя направление обработки строки справа налево (от больших адресов к меньшим). Счетчик цикла инициализируется числом байтов в строке экрана (предложение 17), а в регистр AL заносится код искомого символа - пробела. На этом заканчиваются подготовительные действия.

Команда scasb (scan string byte, сканирование строки байтов) осуществляет сравнение содержимого регистра AL и ячейки памяти по адресу, находящемуся в паре регистров ES:DI. Результат сравнения фиксируется в регистре флагов процессора. После каждой операции сравнения регистр DI получает отрицательное (поскольку установлен флаг DF) приращение, подготавливая операцию сравнения следующего байта. Поскольку команда предваряется префиксом повторения repne (repeat while not equal, повторение пока не равно), она выполняется многократно, но не более CX раз, а фактически - до нахождения первого пробела. При обнаружении равенства содержимого сканируемой строки и регистра AL цикл повторения команды прекращается. При этом регистр DI указывает на следующий (в сторону меньших адресов) байт за найденным, т.е. на последний байт слова, предшествующего пробелу.

При следующем вызове подпрограммы search_blank анализ текста должен начаться с первого байта слова после пробела. Чтобы перевести на этот байт указатель DI, к нему прибавляется 2. Полученное значение указателя копируется в регистр CX и вычитанием адреса начала строки определяется длина строки, выводимой на экран (в дальнейшем ее надо будет дополнить пробелами до 80 символов). После восстановления регистров BX и CX процедура заканчивается командой возврата из подпрограммы ret.

Подпрограмма mov_text служит для записи в видеобуфер указанного числа символов. Подпрограмме передается адрес строки символов (в паре регистров DS:SI), текущий адрес видеобуфера (в регистрах ES:DI) и число записываемых символов (в регистре CX). В процессе записи на экран коды ASCII символов перемежаются их атрибутами. Атрибут задается также в качестве параметра в регистра AH.

В подпрограмме сохраняется в стеке длина выводимой на экран строки, так как она в дальнейшем понадобится для определения числа

добавляемых пробелов. Сбрасывается флаг DF, задавая обработку строки вперед. Далее в цикле в регистр AL загружается очередной символ исходного текста, и из регистра AX этот символ уже с атрибутом записывается в видеобуфер. Команда lodsb забирает байты и выполняет инкремент указателя SI на единицу, а команда stsw заносит в видеобуфер слова и инкрементирует указатель DI на 2. После завершения цикла восстанавливается исходное содержимое CX и подпрограмма завершается.

Подпрограмма mov_blank обеспечивает заполнение оставшейся части строки экрана пробелами. Ей передается текущий адрес видеобуфера (в регистрах ES:DI), число уже записанных в эту строку экрана символов (в регистре CX) и атрибут символа (в регистре AH). Хотя подпрограмма выводит на экран не символы, а пробелы, ей нужно знать значение атрибута, так как пробел заполняет знакоместо цветом фона. Подпрограмма возвращает текущий адрес в видеобуфере (фактически всегда адрес начала следующей строки экрана).

После сохранения содержимого используемого регистра BX, подпрограмма определяет оставшееся до конца строки экрана число знакомест, вычитая из 80 число уже записанных символов, и инициализирует этим значением счетчик цикла CX (предложения 38..40). Полученное число добавляется к текущему значению переменной symbs, исходное значение которой равно длине исходного текста. Учет добавленных пробелов после вывода на экран очередной строки текста необходим для того, чтобы можно было определить, когда программа должна завершить свою работу. В регистр AL заносится код выводимого символа (пробела); атрибут символа уже находится в регистре AH. Далее командой stosw с префиксом rep в видеобуфер записываются CX пробелов, восстанавливается значение регистра BX и подпрограмма завершается.

Главная процедура main начинается с инициализации регистра DS и очистки экрана вызовом подпрограммы clear_screen. Далее записываются исходные значения характерных адресов в ячейки mem_seg, mem_offs, scr_seg и scr_offs. После настройки параметров вызывается подпрограмма поиска последнего пробела в строке search_blank. Полученный от нее относительный адрес начала следующей порции текста заносится в ячейку mem_ofss (предложение 63), но не ранее, чем текущее значение адреса переносится в регистр SI в качестве параметра для подпрограммы mov_text. После подготовки остальных параметров вызывается подпрограмма mov_text и сразу вслед за ней подпрограмма mov_blank, которые заполняют первую строку экрана текстом и добавочными пробелами. Полученный текущий адрес на экране сохраняется в переменной scr_offs (предложение 69).

В предложении 70 из текущего числа символов, ожидающих вывода, вычитается длина только что выведенной строки (80 байт). Как только

результат вычитания станет равен 0 (все символы текста выведены), команда условного перехода je передаст управление на завершение программы. Пока же этого не произошло, команда безусловного перехода jmp возвращает управление на начала цикла анализа и вывода строк текста.

Рассмотренная программа имеет неприятный дефект. Обрабатываемый текст должен заканчиаться символом пробела, иначе последняя строка (даже, точнее говоря, последнее слово последней строки) будет обрабатываться неверно.

Обратите внимание на конструкцию

```
begin:
        ...
        je      exit
        jmp     begin
exit:
```

В данном случае ее лучше было заменить одной командой jne begin, что позволило бы отказаться от команды jmp и несколько сократить объем программы. Однако использованная конструкция более универсальна. Дело в том, что команды условного перехода передают управление на ограниченное количество байтов, именно, не более, чем на 127 байт вперед или 128 байт назад. Это связано с тем, что все команды условных переходов имеют размер 2 байт. Первый байт команды занят кодом операции, а второй - смещением к точке перехода. Поскольку необходимо обеспечить переход и вперед, и назад, в этом байте записывается знаковое число, а диапазон изменения байтового числа со знаком как раз и составляет -128...+127. В нашей программе расстояние от точки расположения команды je до метки begin составляет CCh=-52 байт (вспомните материал статьи 36). Поэтому в данном случае можно было ограничиться одной командой условного перехода. Если бы, однако, в процессе модификации программы в нее добавилось более 76 байт программных кодов, пришлось бы воспользоваться парой команд je-jmp.

Для того, чтобы развлечься и заодно приобрести навыки работы с атрибутами символов на экране, вы можете включить в какое-нибудь место цикла команду inc attrib или что-нибудь вроде add attrib,16. Такая добавка позволит выводить на экран строки разного цвета.

Статья 41

Создание файла на диске

Одной из важнейших операций, используемых практически в любой программе, является работа с дисковыми файлами. Процедура обращения к файлу в общем случае распадается на следующие этапы:

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.

Кроме перечисленных, имеется еще целый ряд вспомогательных операций: удаление файла, получение или установка его атрибутов, получение или изменение даты и времени создания файла, поиск файла с заданным именем (либо файлов, удовлетворяющих условиям заданного шаблона групповой операции) и др.

При открытии имеющегося или создании нового файла DOS выделяет для хранения и модификации характеристик открытого файла (его имени и расширения, длины и т.д.) свободный элемент одной из своих таблиц. Для ссылки на этот элемент используется система указателей. Исходный указатель, называемый обычно дескриптором файла, DOS возвращает в прикладную программу. В дальнейшем любое обращение к файлу осуществляется уже не по имени файла, а по его дескриптору. Дескриптор освобождается при закрытии файла и может быть использован повторно.

Напишем простую программу, которая создает на диске файл с некоторой символьной (для простоты отладки) информацией.

Пример 41.1. Создание файла.

```
;Создадим файл
        mov     AH,3Ch      ;Функция создания файла
        mov     CX,0        ;Без атрибутов
        mov     DX,offset filename;Адрес имени файла
        int     21h         ;Вызов DOS
        mov     handle,AX   ;Сохраним дескриптор файла
;Запишем строку в файл
        mov     AH,40h      ;Функция записи
        mov     BX,handle   ;Дескриптор
        mov     CX,buflen   ;Число записываемых байтов
```

```
        mov     DX,offset buf;Адрес буфера
        int     21h           ;Вызов DOS
;Закроем файл
        mov     AH,3Eh        ;Функция закрытия
        mov     BX,handle     ;Дескриптор
        int     21h           ;Вызов DOS
;Завершим программу
        mov     AX,4C00h      ;Функция завершения
        int     21h           ;Вызов DOS
;Поля данных
buf     db '0123456789'       ;Данные, записываемые в файл
buflen  equ     $-buf         ;Их длина
handle  dw      ?             ;Ячейка для дескриптора
filename db     'D:\test\myfile.001',0;Спецификация файла в
                              ;формате ASCIIZ
```

Функция 3Ch позволяет создать файл с заданной спецификацией. Спецификация файла, т.е. путь к нему вместе с именем файла и расширением указывается в виде символьной строки, завершающейся двоичным нулем ("строки ASCIIZ") Адрес этой строки заносится в регистры DS:DX. При задании имени файла следует руководствоваться обычными правилами DOS: имя файла можно вводить как строчными, так и прописными буквами; в имени файла может быть от 1 до 8 символов, а в расширении - от 0 до 3 (т.е. расширение может отсутствовать); если не указан диск или каталог, по умолчанию используются текущие диск или текущий каталог. В примере 41.1 файл с именем MYFILE.001 создается на диске D: в каталоге test, который к моменту создания файла должен уже существовать.

При вызове функции 3Ch в регистре CX задается код атрибутов создаваемого файла: 0 - отсутствие атрибутов, 1 - только для чтения, 2 - скрытый, 4 - системный, 8 - метка тома, 20h - атрибут архивации. Таким образом, с помощью функции 3Ch можно создать как "настоящий" файл, так и метку тома (в корневом каталоге диска).

После выполнения всех необходимых действий по созданию файла, функция 3Ch возвращает в регистре AX дескриптор созданного файла, которым можно в дальнейшем пользоваться для записи в файл или чтения из него. Стоит заметить, что если файл с указанным именем уже существовал, функция 3Ch уничтожит имеющийся файл и создаст новый с тем же именем.

Для записи в файл данных используется функция 40h, которой мы уже пользовались для вывода символьных строк на экран. Перед вызовом функции в регистр BX помещается дескриптор, в регистр CX - число записываемых байтов, а в регистры DS:DX - адрес буфера в программе пользователя.

Следует обратить внимание на задание адреса буфера. Как уже неоднократно отмечалось, любой адрес является величиной двухсловной и включает два компонента: сегментный адрес и смещение, или относи-

тельный адрес. При использовании функции DOS, которая для своей работы требует указания адреса некоторого объекта, надо посмотреть в справочнике по функциям DOS, в каких регистрах функция ожидает найти этот адрес. Для функции 3Ch адрес имени файла задается в паре регистров DS:DX; в других случаях могут использоваться иные пары регистров, например, DS:SI, ES:DI, ES:BX и проч.

Функция 3Eh закрывает файл, заполняя при этом назначенную этому файлу запись в каталоге, куда заносится информация об имени файла, его длине, дате и времени создания, адресе на диске, атрибутах. В простых программах открытые файлы можно не закрывать, так функция завершения программы 4Ch закрывает все открытые программой файлы.

Готовя к выполнению программу 41.1, укажите в спецификации файла ваш рабочий диск и заранее создайте на нем каталог TEST. Выполнив программу, убедитесь, что в каталоге TEST появился файл с именем MYFILE.001. Выведите на экран его содержимое (например, командой TYPE).

Рассмотрим теперь последовательность работы с уже имеющимся файлом.

Пример 41.2. Чтение файла.

```
;Откроем файл
        mov     AH,3Dh      ;Функция открытия файла
        mov     AL,2        ;Доступ для чтения/записи
        mov     DX,offset filename;Адрес имени файла
        int     21h
        mov     handle,AX   ;Сохраним дескриптор
;Попытаемся прочитать 65535 байт
        mov     AH,3Fh      ;Функция чтения
        mov     BX,handle   ;Дескриптор
        mov     CX,65535    ;Столько читать
        mov     DX,offset bufin;Сюда
        int     21h
        mov     CX,AX       ;Столько реально прочитали
;Выведем прочитанное на экран
        mov     AH,40h      ;Функция записи
        mov     BX,1        ;Дескриптор стандартного вывода
        mov     DX,offset bufin;Отсюда выводить (CX байт)
        int     21h
;Поля данных
bufin   db      80 dup (' ');Буфер ввода
handle  dw      ?           ;Ячейка для дескриптора
filenamedb      'D:\test\myfile.001',0;Спецификация файла
```

Для того, чтобы получить доступ к имеющемуся файлу, его надо открыть функцией 3Dh. Функция требует указания адреса спецификации файла в регистрах DS:DX и кода доступа в регистре AL. Можно открыть доступ к файлу для чтения и записи (код 2), только для чтения (код 0) или только для записи (код 1). Функция открытия файла, так же, как и

функция создания, возвращает в регистре AX дескриптор, выделенный системой.

После открытия файла можно выполнить его чтение. Для этого предусмотрена функция 3Fh, которая, в зависимости от указанного дескриптора, может читать из файла или устройства (например, с клавиатуры). В регистр BX заносится полученный от DOS дескриптор, а в регистре CX - число читаемых байтов. Если требуется прочитать в программу весь файл, а его длина неизвестна, в регистре CX можно указать число, заведомо большее длины файла. В примере указано максимально возможное число 65535.

В регистрах DS:DX указывается адрес буфера, куда DOS скопирует содержимое файла. Как правило, указывается поле в сегменте данных программы, однако в принципе допустимо указывать любой законный адрес, например, адрес видеобуфера. В этом случае содержимое файла будет прочитано прямо в видеобуфер и появится на экране. Правда, не следует забывать, что в видеобуфере байты символов чередуются с байтами атрибутов, и для получения разумного изображения именно такая информация должна быть записана в читаемом файле.

В нашей программе в регистрах DS:DX указывается адрес буфера в программе. При этом предполагается, что размер читаемого файла не превышает 80 байт. В противном случае буфер следует увеличить.

Функция чтения 3Fh возвращает в регистре AX число реально прочитанных байтов. Это дает возможность работать с файлами неизвестной заранее длины. В примере 41.2 прочитанная информация выводится на экран функцией 40h, которая в данном случае удобнее используемой нами ранее функции 09h, так как позволяет обойтись без завершающего знака "$" (где мы его поставим, если не знаем длину читаемого текста?) и вывести на экран заданное число символов.

Статья 42

Анализ системных ошибок

При использовании в программе функций DOS или BIOS нередко возникает ситуация, когда программа, написанная формально правильно, тем не менее не выполняется в результате системного сбоя, называемого обычно системной ошибкой. Например, в программе делается

попытка открыть файл, а файл с таким именем в действительности отсутствует. Надо заметить, что в другой ситуации (при наличии на диске открываемого файла) программа выполнится без всяких сбоев. Именно из-за того, что системные ошибки не являются систематическими, а возникают в зависимости от конкретных условий работы программы, их фиксирование и анализ является почти обязательным элементом любой программы, обращающейся по ходу своего выполнения к DOS.

Большинство функций DOS и многие функции BIOS возвращают в флаге переноса CF код завершения. Если функция выполнилась успешно, CF=0, в случае же любой ошибки CF=1. В последнем случае в регистре 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      error       ;Флаг CF установлен?
;Нет, нормальное продолжение программы
        ...
error:
;Да, проанализируем код ошибки в AX
        cmp     AX,1        ;Код 1?
        je      err1        ;Да, на обработку этой ошибки
        cmp     AX,2        ;Код 2?
        je      err2        ;Да, на обработку этой ошибки
        ...
err1:
        ...
err2:
        ...
```

Усовершенствуем программу 41.2 с целью отработки возможных системных ошибок. При работе с файлом наиболее вероятными будут ошибки с кодами 2 (файл не найден) и 3 (путь не найден), хотя, конечно могут быть и другие ошибки.

Пример 42.1. Чтение файла с анализом системных ошибок.

```
        include mac.mac  ;Подсоединим файл с макрокомандами
;Откроем файл
        mov     AX,3D02h ;Функция открытия файла
        mov     DX,offset filename;Адрес имени файла
        int     21h      ;Вызов DOS
        jc      error    ;Системная ошибка?
        mov     handle,AX ;Нет, сохраним дескриптор
;Попытаемся прочитать 65535 байт
        mov     AH,3Fh   ;Функция чтения
        mov     BX,handle ;Дескриптор
        mov     CX,65535 ;Столько читать
```

```
        mov     DX,offset bufin;Сюда
        int     21h
        mov     CX,AX       ;Столько реально прочитали
;Выведем прочитанное на экран
        mov     AH,40h      ;Функция записи
        mov     BX,1        ;Дескриптор стандартного вывода
        mov     DX,offset bufin;Отсюда выводить (CX байт)
        int     21h
;Завершим программу
outprog:mov     AX,4C00h    ;Завершение программы
        int     21h
error:  cmp     AX,02       ;Код ошибки = 2?
        je      not_found   ;Да, не найден файл
        cmp     AX,03       ;Код ошибки = 3?
        je      no_path     ;Да, не найден каталог
        jmp     outprog     ;Другая ошибка, на выход
not_found:                  ;Отработка ошибки "не найден файл"
        write   mes1
        jmp     outprog     ;На выход
no_path:                    ;Отработка ошибки "не найден каталог"
        write   mes2
        jmp     outprog     ;На выход
;Поля данных
bufin   db      80 dup (' ');Буфер ввода
handle  dw      ?           ;Ячейка для дескриптора
filenamedb      'F:\test\myfile.001',0;Спецификация файла
mes1    db      'Файл не найден$'
mes2    db      'Не найден каталог$'
```

В примере 42.1 анализируется завершение только одной функции DOS, именно, функции 3Dh открытия файла. Если сразу после выполнения команды int 21h флаг CF оказывается установлен, произошел системный сбой. Управление передается на метку error, где последовательно анализируется код ошибки в регистре AX. Если код ошибки равен 2, выводится сообщение "Файл не найден"; если код ошибки равен 3, выводится сообщение "Не найден каталог". В любом случае осуществляется переход на завершение программы, которая уже не пытается читать из отсутствующего файла. То же происходит и при возникновении любой другой ошибки.

Подготовив программу, выполните ее в разных вариантах. Сначала убедитесь, что при наличии файла MYFILE.001 в каталоге TEST на указанном диске программа работает нормально (выводит на экран содержимое файла). Затем удалите файл и повторите прогон программы. Далее удалите и каталог TEST и снова запустите программу.

Как видно из приведенного примера, обработка системных ошибок заметно усложняет программу. Однако для ответственных программ, которые могут выполняться в изменяющихся условиях, она совершенно необходима. Следует иметь в виду, что DOS только сообщает об ошибках с помощью установки флага CF, но никак их не обрабатывает. Отсутствие анализа ошибок может привести к непредсказуемому

поведению программы. Пусть, например, в программе не открылся файл из-за отсутствия на диске указанного в спецификации файла каталога. В этом случае в регистре AX будет возвращен не дескриптор открытого файла, а код ошибки, равный 3. Если дальше в программе предусмотрен вывод в открытый файл через возвращенный в регистре AX дескриптор, то DOS выполнит вывод через дескриптор 3, который по умолчанию закреплен за последовательным портом. Ясно, что программа дальше будет работать совершенно неправильно.

Статья 43

Завершение программы и анализ кода возврата в командном файле

Как уже отмечалось ранее, выполнение программы, написанной на языке ассемблера, начинается с точки (имени процедуры или метки), указанной в качестве операнда завершающей директивы end. Это логическое начало программы (точка входа) может находиться как в начале текста программы, так и в любом другом месте. Программные строки выполняются процессором в точности в том порядке, в каком они следуют в исходном тексте программы. Поэтому недопустимо, например, перемежать программные строки строками данных: процессор не может отличить коды команд от кодов данных и будет "выполнять" строку данных, в предположении, что это команды. Точно также недопустимо располагать процедуры обработки прерываний в теле основной программы. И данные, и строки обработки прерываний надо либо предварять командами безусловных переходов с целью обхода этих участков, либо располагать их в таких местах программного модуля, которые не попадут в естественный ход выполнения программы, например, перед точкой входа в программу.

Как же завершить выполнение программы? Если не принять специальных мер, процессор, выполнив последнюю, с нашей точки зрения, строку программы, продолжит выполнение следующих за ней строк, в которых, возможно, расположены данные программы, или ее стек, или вообще это уже память за пределами программы. Такое часто происходит в процессе отладки программ и обычно приводит к разрушению системы. Однако завершение программы заключается не просто в оста-

новке процессора. Для того, чтобы после окончания выполнения нашей программы система осталась работоспособной, необходимо передать управление командному процессору COMMAND.COM, который выведет на экран системный запрос и будет ожидать следующих команд оператора. Это делается с помощью функции DOS 4Ch, которой, таким образом, должна заканчиваться практически любая программа.

Функция 4Ch требует единственного параметра - кода возврата, который помещается программой перед вызовом функции 4Ch в регистр AL. Хотя коды возврата можно трактовать как угодно, принято считать, что при успешном выполнении программы код возврата должен быть равен 0; отличное от нуля значение кода возврата говорит о возникновении при выполнении программы каких-либо ошибок.

Функция 4Ch закрывает файлы, открытые программой, модифицирует записи в каталогах, отражая текущее состояние этих файлов, освобождает память, которую занимала программа, и передает управление командному процессору. Содержимое регистра AL, т.е. код возврата, сохраняется в одной из системных ячеек, откуда его можно получить с помощью функции DOS 4Eh. Эту операцию выполняет внутренняя команда командных файлов if errorlevel, с помощью которой можно определить успешность выполнения закончившейся только что программы (если, конечно, эта программа возвращает код завершения). Для этого программу надо запустить не с клавиатуры, а из командного файла. В примере 43.1 использована программа из статьи 42, в которую включены строки передачи в DOS, в качестве кода возврата, кода системной ошибки при операции открытия файла. Этот код анализируется командным файлом (пример 43.2), который выводит на экран соответствующее сообщение, если запущенная из него программа не смогла открыть файл.

Пример 43.1. Чтение файла с анализом системных ошибок.

```
;Откроем файл
        mov     AH,3Dh      ;Функция открытия файла
        mov     AL,2        ;Доступ для чтения/записи
        mov     DX,offset filename;Адрес имени файла
        int     21h
        jnc     go
        mov     errcode,AL
        jmp     outprog
go:     mov     handle,AX   ;Сохраним дескриптор
;Попытаемся прочитать 65535 байт
        mov     AH,3Fh      ;Функция чтения
        mov     BX,handle   ;Дескриптор
        mov     CX,65535    ;Столько читать
        mov     DX,offset bufin;Сюда
        int     21h
        mov     CX,AX       ;Столько реально прочитали
```

```
;Выведем прочитанное на экран
        mov     AH,40h      ;Функция записи
        mov     BX,1        ;Дескриптор стандартного вывода
        mov     DX,offset bufin;Отсюда выводить (CX байт)
        int     21h
;Завершим программу
outprog:mov     AH,4Ch      ;Завершение программы
        mov     AL,errcode
        int     21h
;Поля данных
bufin   db      80 dup (' ');Буфер ввода
handle  dw      ?           ;Ячейка для дескриптора
filenamedb      'F:\test\myfile.001',0;Спецификация файла
errcode db      0
```

Пример 43.2. Командный файл для запуска программы 43.1 (с именем P.EXE)

```
@echo off
p.exe
if errorlevel 1 goto error
goto end
:error
echo Программа P.EXE не смогла открыть файл!
:end
```

Программа P.EXE, завершившись, возвращает управление командному файлу. Команда if errorlevel анализирует возвращенный функцией 4Ch код возврата и, если он равен или больше 1, осуществляет переход на метку error. На экран выводится сообщение об ошибке и командный файл завершается. Если код возврата равен 0, оператор goto error не выполняется, следующей строкой командного файла осуществляется переход на метку end и файл завершается без выдачи сообщения об ошибке.

Статья 44

Программирование портов. Звук

Управление разнообразной аппаратурой компьютера - контроллером клавиатуры, видеоадаптером, последовательными и параллельными портами и др., осуществляется через управляющие регистры этой аппаратуры. Каждый из регистров имеет закрепленный за ним номер в диапазоне от 0000h до FFFFh. Этот номер называется портом, и программиро-

вание аппаратуры путем непосредственного обращения к ее регистрам носит название программирования через порты. Обращение к портам осуществляется с помощью двух команд - in (input, ввод) для ввода и out (output, вывод) для вывода. Никакие другие команды для работы с портами не годятся, поэтому (к сожалению) непосредственно в регистрах аппаратуры нельзя выполнять арифметические или логические операции. При необходимости изменить содержимое аппаратного регистра, надо сначала с помощью команды in прочитать содержимое этого регистра в регистр процессора или ячейку памяти, там требуемым образом изменить его и затем с помощью команды out вывести новое данное назад в регистр аппаратуры.

В настоящей статье мы рассмотрим методику программирования ввода-вывода через порты на примере управления динамиком компьютера.

Программирование звука в простейшем случае осуществляется путем периодического включения и выключения тока, протекающего через динамик, по командам программы. Если частота переключения тока лежит в пределах приблизительно от 16 Гц до 16 кГц, мы слышим звуковой тон соответствующей частоты. Для управления током динамика служит бит 1 порта с номером 61h. Установка этого бита в 1 включает ток в динамике, установка в 0 - выключает. Таким образом, для получения звукового тона программа должна непрерывно переключать этот бит, что исключает выполнение программой какой-либо работы одновременно с выводом звука. Для того, чтобы организовать звуковое сопровождение, т.е. генерацию звука одновременно с работой программы, следует воспользоваться более сложным методом, который будет описан в следующей статье. Здесь приводится пример простейшей программы, генерирующей одиночный звуковой сигнал заданной высоты.

Пример 44.1. Программная генерация звука

```
        cli                 ;Запрет аппаратных прерываний
        in      AL,61h      ;Введем содержимое порта 61h
        mov     CX,4000     ;Установим длительность звукового
                            ;сигнала
begin:  push    CX          ;Сохраним счетчик цикла
        or      AL,2        ;Установим бит 1
        out     61h,AL      ;Выведем в порт, включив динамик
        mov     CX,1000     ;Организуем паузу, в течение
        loop    $           ;которой через динамик течет ток
        and     AL,11111101b;Сбросим бит 1
        out     61h,AL      ;Выведем в порт, выключив динамик
        mov     CX,1000     ;Организуем паузу, в течение которой
        loop    $           ;через динамик не течет ток
        pop     CX          ;Восстановим счетчик цикла
        loop    begin       ;Повторять CX раз
        sti                 ;Разрешение аппаратных прерываний
```

Фрагмент генерации звука начинается с запрещения аппаратных прерываний с помощью команды cli (clear interrupts, запрет прерываний). В действительности требуется запретить только прерывания от таймера, которые, возникая 18 раз в секунду, будут прерывать генерацию тона и искажать звук. Однако для выборочного запрещения прерываний надо программировать контроллер прерываний, что относительно сложно. Проще запретить все прерывания.

Следующей командой в регистр AL читается содержимое порта 61h. Команды чтения-записи через порты in и out имеют весьма жесткий формат. Читать из порта можно только в регистр AL (или, если данные в порте 16-разрядные, то в регистр AX). При этом в команде необходимо указывать обозначение регистра. Что касается задания номера порта, то его можно указать в виде числа (если его номер не превышает FFh) или поместить в регистр DX. В последнем случае команда ввода приобретает формат in AL,DX.

Чтение содержимого порта необходимо из-за того, что, как отмечалось выше, в регистре аппаратуры, адресуемом через порт, нельзя выборочно изменять отдельные биты. Прочитанное из порта данное будет сохраняться у нас в регистре AL. Далее в регистр CX заносится число шагов цикла, которое определяет длительность звучания тона. Поскольку регистр CX нам понадобится внутри цикла, его содержимое сразу же сохраняется в стеке.

Следующей командой or над содержимым регистра AL выполняется операция логического ИЛИ.

Команда логического сложения or устанавливает в 1 все биты первого операнда, соответствующие установленным битам второго операнда и оставляет без изменения остальные биты первого операнда (рис. 44.1).

Исходное число	1	2	3	4 h	(16-ричное представление)
Исходное число	0001	0010	0011	0100	(двоичное
Операнд-маска	1111	0000	0000	0000	представление)
Результат	1111	0010	0011	0000	

Рис. 44.1. Результат действия команды or.

В нашем случае в данном, которое мы получили из порта 61h, устанавливается дополнительно бит 1 (его вес составляет 2) и это модифицированное данное выводится в порт командой out. Формат команды out такой же, как и у команды in, только операнды поменялись местами. Установка в порте 61h бита 1 включает ток в динамике. Это состояние динамика должно длиться полпериода генерируемой нами частоты, поэтому дальше организована программная задержка, длительность

которой определяет частоту тона. Команда loop $, которая передает управление по адресу, находящемуся в счетчике текущего адреса $, т.е. на текущую команду, полностью аналогична команде

```
label1: loop  label1
```

но позволяет обойтись без символического обозначения метки. Конечно, это своего рода трюкачество, затрудняющее чтение программы, однако если в программе таких циклов много, для каждого пришлось бы вводить свою метку, что тоже не очень удобно.

После окончания паузы сначала в регистре AL, а затем и в порте 61h сбрасывается бит 1. Это делается командой and, причем для большей наглядности маска этой команды, задающая сбрасываемые биты, записана в двоичной форме. При записи в исходном тексте программы двоичного числа его следует заканчивать буквой b. Сброс бита 1 в порте выключает ток в динамике, образуя второй полупериод звукового колебания. Его длительность задается программной задержкой.

Далее содержимое счетчика цикла восстанавливается из стека и командой loop организуется цикл из CX шагов. В конце фрагмента необходимо выполнить команду sti (set interrupts, разрешить прерывания), иначе не только таймер, но и все остальные устройства компьютера, в частности, клавиатура, перестанут работать.

Описанная методика генерации звукового тона может быть использована в программах для подачи предупреждающих сигналов, однако их частота, а вместе с ней и длительность, будут зависеть от скорости работы процессора, что, конечно, чрезвычайно неудобно. Более совершенная методика генерации звука, основанная на использовании системного таймера, описывается в следующей статье.

Статья 45

Програмирование звукового каналя таймера

В программировании звука принимают участие два узла компьютера - таймер, обычно входящий в состав многофункциональной микросхемы периферийного контроллера, и один из портов контроллера клавиатуры (именно, 61h), управляющий клавиатурой и динамиком. На рис. 45.1 приведена упрощенная схема взаимодействия этих узлов.

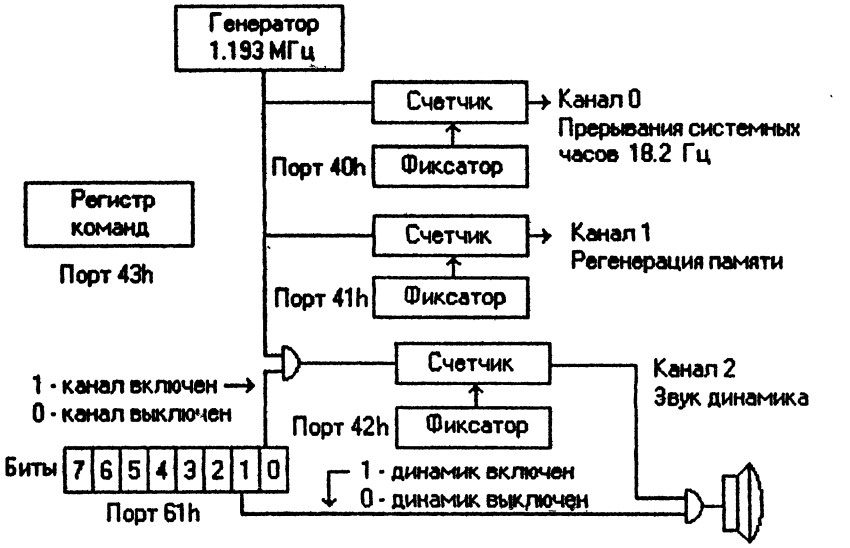


Рис.45.1. Элементы компьютера, принимающие участие в генерации звука.

Подсистема таймера работает независимо от процессора (и параллельно с ним) от собственного генератора, вырабатывающего сигналы с частотой 1,19318 МГц. Таймер имеет три независимых канала, каждый из которых можно независимо перепрограммировать. Для управления режимами программирования в подсистеме таймера имеется регистр команд, обращение к которому осуществляется через порт 43h. Каждый канал таймера включает счетчик, пересчитывающий сигналы от генератора и регистр-фиксатор, в который программно заносится число, определяющее коэффициент пересчета счетчика. Фиксаторы каналов таймера адресуются через порты 40h, 41h и 42h.

При включении компьютера в фиксатор канала 0 заносится максимально возможное число 65535 (FFFFh), в результате чего сигналы на выходе канала 0 имеют частоту 18,2 1/с. Эти сигналы возбуждают прерывания с вектором 08h, которые обрабатываются программой BIOS, осуществляющей отсчет текущего времени. Прерывания от канала 0 можно использовать для временной синхронизации программы, например, для периодического вывода на экран некоторой информации.

Канал 1 таймера используется схемами регенерации памяти и его лучше в программах не использовать.

Выход канала 2 связан с динамиком и используется для генерации звука. Изменяя содержимое фиксатора этого канала, можно изменять частоту сигналов, поступающих на динамик, от 18,2 Гц до 1,19 МГц.

Реально для возбуждения звука можно использовать частоты не выше приблизительно 10 кГц.

Управление каналом 2 таймера и его подключение к динамику осуществляется через порт 61h контроллера клавиатуры.

Бит 0 порта 61h управляет включением и выключением канала 2 таймера. Пока бит 0 установлен, на выходе канала действуют периодические сигналы заданной фиксатором частоты; при сбросе бита 0 колебанию прекращаются. Бит 1 того же порта, как уже описывалось в предыдущей статье, управляет посылкой в динамик тока. Таким образом, при использовании для возбуждения звука таймера, имеются две возможности прекратить звучание тона: запретить работу канала 2 путем сброса бита 0 порта 61h или выключить прохождение через динамик тока путем сброса бита 1 того же порта.

Программирование любого канала таймера осуществляется одинаково. Прежде всего в регистр команд (порт 43h) засылается управляющее слово, характеризующее режим работы таймера. Формат управляющего слова приведен на рис. 45.2.

Рис. 45.2. Формат управляющего слова таймера.

Бит 0 управляющего слова определяет способ задания константы пересчета. При нулевом значении бита константа задается в двоичной форме, при единичном - в двоично-десятичной (BCD). Чаще используется двоичный способ задания константы. В биты 1...3 засылается режим работы таймера (разовый или периодический, с различной скважностью и проч.). Для возбуждения звука используется режим 011 - периодическая генерация прямоугольных сигналов со скважностью 2. В битах 4...5 записывается способ загрузки в фиксатор константы. Обычно используемый код 11 позволяет загрузить в фиксатор 16-разрядную константу пересчета двумя последовательными командами out. Сначала загружается младший байт константы, затем старший. Наконец, в биты 6...7 засылается номер программируемого канала таймера. Как видно из рис. 45.1, для программирования звука используется канал 2 (код 10). Таким образом, управляющее слово оказывается равным 10110110b, или B6h.

Детали программирования таймера будут ясны из приведенных ниже примеров.

Пример 45.1. Управление звуком от таймера. Генерация тона

```
        include mac.mac  ;Объявление макробиблиотеки
;Установим режим таймера
        mov    AL,0B6h   ;(1)Управляющее слово
```

```
        out     43h,AL      ;(2)Выведем его в регистр команд
;Установим частоту канала 2 таймера
        mov     AX,11930    ;(3)1193000 Гц/11930=100 Гц
        out     42h,AL      ;(4)Младший байт константы в порт
        mov     AL,AH       ;(5)AL=старший байт константы
        out     42h,AL      ;(6)Старший байт константы в порт
;Включим динамик и разрешим таймер
        in      AL,61h      ;(7)Введем содержимое порта 61h
        or      AL,3        ;(8)Установим биты 0 и 1
        out     61h,AL      ;(9)Выведем в порт
;После задержки выключим динамик и запретим таймер
        delay   50          ;(10)Задержка
        and     AL,11111100bh;(11)Сбросим в AL биты 0 и 1
        out     61h,AL      ;(12)Выведем в порт
```

Программа 45.1 возбуждает тон заданной частоты и продолжительности. В предложениях 1-2 в регистр команд таймера засылается управляющее слово, описанное выше. После получения управляющего слова микросхема таймера ждет получения двух байтов данных, которые она будет рассматривать, как младший и старший байты константы пересчета. Поэтому после посылки управляющего слова необходимо выполнить две команды out в порт 42h. Константа пересчета, определяющая частоту тона, загружается нами в регистр AX (предложение 3), откуда ее младший байт засылается в порт. В предложении 5 командой mov старший байт константы отправляется в младший байт регистра AX, после чего вторая команда out загружает старшую половину фиксатора. Все это время звука нет.

В предложениях 7...9 к содержимому порта 61h (неизвестному нам) добавляются два младшие бита (число 3). Тем самым включается ток динамика и его периодическое переключение от таймера.

После программной задержки длительностью несколько секунд в сохраненном содержимом регистра AL снова сбрасываются два младшие бита, и это данное пересылается в порт, выключая звук. При выполнении примеров настоящей статьи следует помнить, что величина программной задержки зависит от скорости работы процессора. Приведенные в программе численные параметры макрокоманды delay были подобраны на компьютере с процессором 80486DX2, работающим на частоте 50 МГц.

В следующем примере показано, как генерируются периодические звуковые посылки с паузами между ними (гудки).

Пример 45.2. Управление звуком от таймера. Звук/пауза

```
;Установим режим таймера
        mov     AL,0B6h
        out     43h,AL
;Установим частоту канала 2 таймера
        mov     AX,1193*10
        out     42h,AL
```

```
        mov     AL,AH
        out     42h,AL
;Организуем цикл
        mov     CX,10       ;Подготовим счетчик цикла
;Включим динамик и таймер
        in      AL,61h
        or      AL,00000001b;Включим таймер
repeat: or      AL,00000010b;Включим динамик
        out     61h,AL
        delay   10          ;Длительность звука
        and     AL,11111101b;Выключим динамик
        out     61h,AL
        delay   5           ;Длительность паузы
        loop    repeat
        and     AL,11111110b;Выключим и таймер
        out     61h,AL
```

Начало программы не отличается от предыдущей. После настройки таймера и установки в нем требуемой частоты инициализируется счетчик цикла (в примере на 10 шагов) и в содержимом порта 61h устанавливается младший бит (включение таймера). Далее выполняется тело цикла, состоящее из двух частей. В первой части в порте 61h устанавливаются биты 0 и 1, чем включается звук. В второй части бит 1 сбрасывается, чем выключается динамик (таймер продолжает работать). В цикл включены две задержки, определяющие длительность тона и паузы (в примере они разные). После окончания цикла выполняется выключение таймера.

На любом этапе вывода звука можно изменять его частоту. В примере 45.3 изменение частоты звука выполняется без его прекращения, чем создается эффект летящей бомбы.

Пример 45.3. Управление звуком от таймера. Непрерывное изменение частоты

```
;Установим режим таймера
        mov     AL,0B6h
        out     43h,AL
;Включим динамик и разрешим таймер
        in      AL,61h
        or      AL,3
        out     61h,AL
;Будем изменять частоту тона в процессе звучания
        mov     CX,200      ;Число шагов
music:  mov     AX,tone     ;Константа
        out     42h,AL      ;Младший байт в порт
        mov     AL,AH
        out     42h,AL      ;Старший байт в порт
        sub     tone,30     ;Изменим константу пересчета
        delay   3           ;Задержка
        loop    music       ;На повторение CX раз
;Выключим динамик и запретим таймер
        in      AL,61h
        and     AL,0FCh
        out     61h,AL
```

```
;Поля данных
tone    dw      8000
```

В первых предложениях программы задается режим программирования и работы таймера, однако константа пересчета, определяющая частоту звука, не загружается. Установкой соответствующих битов в порте 61h включаются динамик и таймер. Далее выполняется цикл, в каждом шаге которого в порт 42h загружается текущее значение константы из ячейки tone, после чего значение константы в ячейке уменьшается и после небольшой задержки цикл повторяется. В конце программы выключаются таймер и динамик. Кстати, если позабыть выполнить эти действия, после завершения программы звук, генерируемый независимо от работы процессора, будет продолжать звучать до перезагрузки компьютера.

В примере 45.3 имеется любопытный нюанс. После засылки в порт 43h управляющего слова выполняется многократное заполнение фиксатора различными значениями константы пересчета. Таймер допускает такую повторную загрузку фиксатора после того, как однажды установлен режим его работы (два первых предложения примера 45.3).

Статья 46

Обработчик прерываний от таймера

Структура обработчика прерываний и его взаимодействие с остальными компонентами программного комплекса определяются рядом факторов, из которых важнейшими являются следующие:

- прерывания, инициализирующие обработчик, могут быть аппаратными (от периферийных устройств) или программными (команда int);
- обработчик может быть входить в состав прикладной программы или представлять собой самостоятельную программную единицу. В последнем случае он относится к специальному классу резидентных программ;
- вектор обрабатываемого прерывания может быть свободным или использоваться системой или какой-либо резидентной прикладной программой;
- если вектор уже используется системой, т.е. в составе DOS имеется системный или прикладной обработчик прерываний с данным номе-

ром, то новый обработчик может полностью заменять уже загруженный (превращая его тем самым фактически в бесполезную программу) или "сцепляться" с ним;

- в случае сцепления с загруженым ранее обработчиком новый обработчик может выполнять свои функции до уже имеющегося в системе или после него.

В настоящей статье будут рассмотрены общие вопросы обработки прерываний на примере простого обработчика прерывания 1Ch. Другие типы обработчиков и проблемы, возникающие при их использовании, будут описаны в последующих статьях.

Для того, чтобы прикладные программы могли использовать сигналы таймера, не нарушая при этом работу системных часов, в программу BIOS, обслуживающую аппаратные прерывания от таймера, поступающие через вектор 08, включен вызов int 1Ch, передающий управление на программу-заглушку BIOS, которая содержит единственную команду iret (рис. 46.1). 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 1Ch.

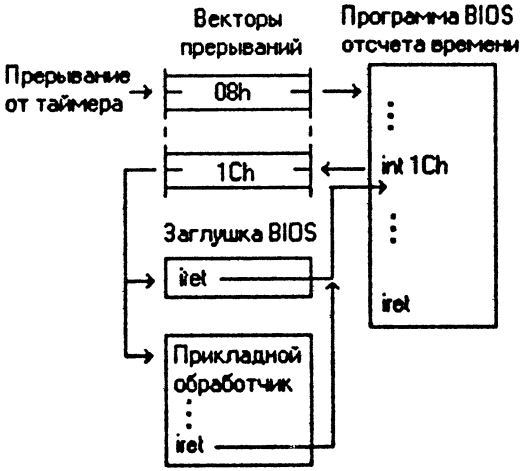


Рис. 46.1. Прикладная обработка прерываний от таймера.

При рассмотрении методики включения в программу процедур-подпрограмм мы отмечали, что порядок расположения процедур в программе не влияет на ход ее выполнения. Важно лишь так скомпоновать текст программы, чтобы подпрограммы никогда не активизировались "сами по себе", иначе, чем в результате выполнения команды call в вызывающей программе. Это правило относится и к обработчикам прерываний, включаемым в состав программы. Текст обработчика можно расположить в любом месте программы, обеспечив лишь невозможность случайного перехода на его строки не в результате прерывания, а по ходу выполнения основной программы. Обычно обработчики располагаются либо в начале, либо в

конце текста программы. В примере 46.1 текст обработчика идет вслед за текстом основной программы.

Другое замечание относится к оформлению программы обработчика. Так же, как и в случае подпрограмм, обработчик прерывания может образовывать процедуру (что наглядно), но может начинаться просто с метки. Важно лишь, чтобы последней выполняемой командой обработчика была команда iret.

Пример 46.1. Обработчик прерываний от таймера.

```
text    segment 'code'    ;(1)
        include mac.mac   ;(2)
        assume CS:text,DS:data;(3)
main    proc              ;(4)Главная процедура
        mov   AX,data     ;(5)Сделаем наши данные
        mov   DS,AX       ;(6)адресуемыми
;Сохраним вектор 1Ch
        mov   AX,351Ch    ;(7)Функция 35h, вектор 1Ch
        int   21h         ;(8)Вызов DOS
        mov   word ptr old_1ch,BX;(9)Сохраним смещение системного
                          ;обработчика
        mov   word ptr old_1ch+2,ES;(10)Сохраним сегмент
                          ;системного обработчика
;Заполним вектор 1Ch
        mov   AX,251Ch    ;(11)Функция 25h, вектор 1Ch
        mov   DX,offset new_1ch;(12)Смещение нашего обработчика
        push  DS          ;(13)Сохраним наш DS
        push  CS          ;(14)Настроим DS на сегмент обработ-
        pop   DS          ;(15)чика (т.е. на сегмент команд)
        int   21h         ;(16)Вызов DOS
        pop   DS          ;(17)Восстановим адресуемость данных
;Организуем контрольный вывод на экран строк текста в цикле с
;задержкой
        mov   CX,20       ;(18)Число повторений вывода строк
wri:    write string      ;(19)Макрокоманда вывода на экран
        delay 50          ;(20)Макрокоманда задержки
        loop  wri         ;(21)
;Перед завершением программы восстановим содержимое вектора 1Ch
        lds   DX,old_1ch  ;(22)Отправим в DS:DX сохраненный
                          ;вектор 1Ch
        mov   AX,251Ch    ;(23)Функция 25h, вектор 1Ch
        int   21h         ;(24)Вызов DOS
;Завершим программу обычным образом
        outprog           ;(25)Завершение программы
main    endp              ;(26)Конец главной процедуры
new_1ch proc              ;(27)Процедура нашего обработчика
;Наш обработчик от таймера. Его функция - вывод на экран мигающего
;символа, свидетельствующего об активности программы
        push  AX          ;(28)Сохраним используемые в нем
        push  ES          ;(29)регистры
        mov   AX,0B800h   ;(30)Настроим ES
        mov   ES,AX       ;(31)на видеобуфер
        mov   AX,CS:sym1  ;(32)Получим символ с атрибутом из
                          ;ячейки sym1
        mov   ES:3998,AX  ;(33)Выведем в последнюю позицию экрана
```

```
        xchg    AX,CS:sym2 ;(34)Обменяем содержимое
        mov     CS:sym1,AX ;(35)ячеек sym1 и sym2
        pop     ES         ;(36)Восстановим
        pop     AX         ;(37)сохраненные регистры
        iret               ;(38)Выход из прерывания
;Поля данных обработчика в сегменте команд
sym1    dw      421Eh      ;(39)Символы с атрибутами
sym2    dw      241Eh      ;(40)для вывода на экран
new_1ch endp               ;(41)Конец процедуры обработчика
text    ends               ;(42)
data    segment            ;(43)
old_1ch dd      0          ;(44)Двухсловная ячейка для
                           ;хранения исходного вектора
string  db '***************0123456789*************',10,13,'$'
;(45)
data    ends               ;(46)
        end     main       ;(47)
```

Главная процедура начинается, как обычно, с инициализации сегментного регистра DS. Перед тем, как устанавливать собственный обработчик какого-либо прерывания, следует сохранить его исходный (системный) вектор, чтобы перед завершением программы вернуть систему в исходное состояние. Для получения содержимого вектора 1Ch используется функция DOS 35h, которая возвращает содержимое указанного вектора в регистрах ES:BX. Для сокращения объема исходного текста программы и номер функции, и номер требуемого вектора заносятся в регистры AH и AL одной командой (предложение 7). Исходный вектор сохраняется в двухсловной ячейке old_1ch, объявленной директивой dd (define double, определить двойное слово) в сегменте данных программы. Однако команды пересылки не могут работать с двойными словами, поэтому сохраняемый вектор засылается из регистров ES:BX в память пословно, сначала в младшую половину ячейки old_1ch (предложение 9), затем в старшую, адрес которой, естественно, равен old_1ch+2 (предложение 10). Поскольку ячейка old_1ch объявлена с помощью директивы dd, для обращения к ее составляющим (словам) необходимо включать в команду описатели word ptr (word pointer, указатель на слово), которые как бы отменяют на время трансляции команды первоначальное описание ячейки.

Сохранив вектор, мы можем приступить к заполнению его адресом нашего обработчика. Для этого используется функция DOS 25h, которая, как уже отмечалось, требует указания в регистре AL номера заполняемого вектора, а в регистрах DS:DX полного адреса обработчика, который и будет записан в указанный нами вектор. Однако регистр DS настроен на сегмент данных программы. Кстати, если бы это было не так, мы не могли бы выполнить предложения 9 и 10, так как поля данных программы адресуются через регистр DS. Поэтому на время выполнения функции 25h нам придется изменить содержимое DS, настро-

ив его на тот сегмент, в котором находится процедура обработчика, т.е. на сегмент команд. Это и выполняется в предложениях 13...15. Содержимое DS сохраняется в стеке, а затем в него через стек заносится содержимое регистра CS, который, очевидно, указывает на сегмент команд. После возврата из DOS в программу исходное содержимое DS восстанавливается (предложение 17).

Начиная с этого момента, прерывания от таймера, приводящие к выполнению в системной программе BIOS команды int 1Ch, будут активизировать 18,2 раз в секунду программу нашего обработчика. При этом вся наша программа должна находиться в памяти и что-то делать, так как если она завершится, то и она, и обработчик уйдут из памяти. Для задержки программы в ней предусмотрен многократный, в цикле вывод на экран строки текста (предложения 18...21).

Перед завершением программы необходимо с помощью той же функции 25h восстановить исходное содержимое вектора 1Ch. Для загрузки регистров DS:DX требуемым адресом в примере 46.1 используется удобная команда lds (load pointer using DS, загрузка указателя с использованием DS). В качестве операндов для этой команды указывается один из регистров общего назначения и двухсловное поле памяти с искомым адресом. Следует иметь в виду, что после выполнения этой команды старое содержимое регистра DS теряется. В нашем примере оно больше не нужно, так как выполнив восстановления вектора, программа завершается (предложение 25). Вообще же перед выполнением команды lds исходное содержимое регистра DS следует сохранить в стеке.

Рассмотрим теперь программу обработчика прерывания от таймера. Программа начинается с сохранения в стеке регистров, которые будут использоваться в обработчике. Это чрезвычайно важное действие, так как переход на программу обработчика осуществляется по команде int 1ch из системной программы обработки прерываний от таймера. При выполнении процедуры прерывания процессор настраивает должным образом только регистры CS и IP. Содержимое всех остальных регистров (в том числе сегментных) отражает состояние системной программы, и если оно будет изменено, то после возврата из нашего обработчика в вызвавшую его системную программу она перестанет функционировать. В нашем обработчике используются лишь регистры AX и ES, которые и сохраняются в стеке (предложения 28-29).

Далее регистр ES настраивается на адрес видеобуфера (предложения 30-31), а в регистр AX помещается код ASCII выводимого на экран символа вместе с его атрибутом (предложение 32). В последнем предложении используется важная возможность замены сегмента. Как уже отмечалось, при обращении к памяти по умолчанию используется сегментный регистр DS, т.е. предполагается, что адресуемая ячейка находится в том сегменте, на который в настоящий момент указывает DS. В

нашем же случае в момент выполнения этой команды DS почти наверное указывает на какой-то сегмент программы BIOS. Для адресации к нашим данным можно сохранить содержимое DS и настроить его на наши данные. Однако можно поступить проще, именно, ввести в команду префикс замены сегмента CS: и тем самым указать транслятору, чтобы он в данной команде использовал адресацию через регистр CS. Но и данные в этом случае следует разместить не в сегменте данных, к которому нет доступа, а в сегменте команд. У нас так и сделано. Ячейки sym1 и sym2, к которым обращается обработчик, расположены в конце процедуры new_1ch в пределах сегмента команд (предложения 39-40).

Вывод одного и того же символа в одно и то же место экрана приведет к тому, что мы не будем знать, работает ли наш обработчик. В нашем примере предусмотрена периодическая смена атрибута символа, что делает символ мерцающим. Для этого при каждом проходе программы обработчика ячейки sym1 и sym2 взаимно обмениваются своим содержимым. В результате на экран выводится то один, то другой код. Для обмена содержимого регистров или ячеек предусмотрена специальная команда xchg (exchange, обмен). Поскольку в микропроцессорах 80x86 запрещены команды с адресацией к памяти в обоих операндах, обмен приходится осуществлять через регистр AX.

После восстановления сохраненных в стеке регистров работа обработчика завершается командой iret, которая передает управление назад в вызвавшую наш обработчик программу BIOS. Когда эта программа дойдет до своего завершения, она выполнит команду iret и управление вернется в нашу программу в ту (неопределенную) точку, в которой она была прервана сигналом таймера.

В примере 46.1 смена символов на экране осуществляется 18,2 раза в секунду, что приводит к неприятному мельканию. Для снижения частоты вывода нам придется в обработчике отсчитывать прерывания и выводить символ на экран не при каждом вызове обработчика, а, скажем, на каждый 4-й вызов. Этот прием проиллюстрирован в примере 46.2.

Пример 46.2. Снижение частоты прерываний

```
new_1ch proc                ;(1)Процедура обработчика прерывания
                            ;1Ch
        push   AX           ;(2)Сохраним используемые в нем
        push   ES           ;(3)регистры
        inc    CS:count     ;(4)Инкремент счетчика
        test   byte ptr CS:count,03h;(5)Пересчет на 4
        jnz    exit         ;(6)Если не 4-е прерывание, на выход
        mov    AX,0B800h    ;(7)Настроим ES
        mov    ES,AX        ;(8)на видеобуфер
        mov    AX,CS:sym1   ;(9)Получим символ с атрибутом из sym1
        xchg   AX,CS:sym2   ;(10)Обменяем содержимое
        mov    CS:sym1,AX   ;(11)ячеек sym1 и sym2
        mov    ES:3998,AX   ;(12)Выведем в последнюю позицию экрана
```

```
exit:   pop    ES          ;(13)Восстановим
        pop    AX          ;(14)сохраненные регистры
        iret               ;(15)Выход из прерывания
;Поля данных обработчика
sym1    dw     421Eh       ;(16)Символы с атрибутами
sym2    dw     241Eh       ;(17)для вывода на экран
count   db     0           ;(18)Счетчик прерываний
```

В обработчике предусматривается ячейка count для отсчета прерываний. При каждой активизации обработчика содержимое этой ячейки увеличивается на 1 (предложение 4). Для выделения каждого четвертого запуска обработчика в программе используется команда test.

Команда test byte ptr CS:count,03h в сочетании с командой jnz exit осуществляет переход на метку exit, т.е. на завершение программы, если установлен хотя бы один из битов 0 или 1 байта count. Таким образом, содержательная часть обработчика будет выполняться лишь в тех случаях, когда оба эти бита сброшены. При последовательном наращивании счетчика count такая ситуация будет возникать при каждом четвертом вызове обработчика.

Следует обратить внимание на необходимость использования в рассматриваемой команде описателя byte ptr (byte pointer, указатель на байт). При указании второго операнда в виде "безразмерного" непосредственного значения транслятор не может решить, анализировать ли ему байт или слово. Необходимость в описателе отпадает, если маска проверки задается в регистре.

Статья 47

Резидентный обработчик прерываний от таймера

Большой класс программ, обеспечивающих функционирование вычислительной системы (драйверы устройств, программы шифрации и защиты данных, русификаторы, обслуживающие программы типа электронных блокнотов или калькуляторов и др.), должны постоянно находиться в памяти и быстро 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 ввиду того, что резидентная программа должна быть максимально компактной, чаще всего в качестве резидентных используют программы типа .COM.

Рассмотрим типичную структуру резидентной программы и системные средства оставления ее в памяти после инициализации (рис. 47.1).

```
text    segment 'code'
        assume CS:text,DS:text
        org    100h
main    proc
        jmp    init        ;Переход на секцию инициализации
        ;Данные резидентной секции программы
        ...
entry:  ;Текст резидентной секции программы
        ...
main    endp
init    proc               ;Секция инициализации
        ...
        mov    DX,(init-main+10Fh)/16;Размер в параграфах
        mov    AH,3100h    ;Функция "Завершить и оставить в
        int    21h         ;памяти"
init    endp
text    ends
        end    main
```

Рис. 47.1. Типичная структура резидентной программы.

Программа пишется в формате .COM, поэтому в ней предусматривается только один сегмент, с котором связываются сегментные регистры CS и DS; в начале сегмента резервируется 100h байт для PSP.

При запуске программы с клавиатуры управление передается (в соответствии с параметром директивы end) на начало процедуры main. Командой jmp сразу же осуществляется переход на секцию инициализации, которая может быть оформлена в виде отдельной процедуры или входить в состав процедуры main. В секции инициализации, в частности, подготавливаются условия для работы программы уже в резидентном состоянии. Последними строками секции инициализации вызывается функция DOS 31h, которая выполняет завершение программы с оставлением в памяти указанной ее части. Размер резидентной части программы (в параграфах) передается DOS в регистре DX. Определить размер резидентной секции можно, например, следующим образом. К разности смещений init-main, которая равна длине резидентной части программы в байтах, прибавляется размер PSP (100h) и еще число 15 (Fh) для того, чтобы после целочисленного деления на 16 результат был округлен в большую сторону.

С целью экономии памяти секция инициализации располагается в конце программы и отбрасывается при ее завершении.

Функция 31h, закрепив за резидентной программой необходимую для ее функционирования память, передает управление командному процессору и вычислительная система переходит в исходное состояние. Наличие программы, резидентной в памяти, никак не отражается на ходе вычислительного процесса, за исключением того, что уменьшается объем свободной памяти. Одновременно в память может быть загружено любое число резидентных программ.

На рис. 47.2 показаны элементы резидентной программы и их взаимодействие.

Рис. 47.2. Взаимодействие элементов резидентной программы.

Любая резидентная программа имеет по крайней мере две точки входа. При запуске с клавиатуры программы типа .COM управление всегда передается на первый байт после PSP (IP=100h). Поэтому практически всегда первой командой резидентной программы является команда jmp, передающая управление на начало секции инициализации.

После отработки функции DOS 31h программа остается в памяти в пассивном состоянии. Для того, чтобы активизировать резидентную программу, ей надо как-то передать управление и, возможно, параметры. Вызвать к жизни резидентную программу можно разными способами, но наиболее употребительным является механизм аппаратных или программных прерываний. В этом случае в секции инициализации необходимо заполнить соответствующий вектор адресом резидентной части программы (точка entry на рис. 47.2). Адрес entry образует вторую точку входа в программу, через которую осуществляется ее активизация. Очевидно, что резидентная секция программы должна заканчиваться командой выхода из прерывания iret.

В статье 46 был рассмотрен обработчик прерываний от таймера, который активизировался прерыванием 1Ch и выводил в угол экрана приблизительно 4 раза в секунду некоторый символ, изменяя при каждом выводе его атрибут. Возьмем эту программу за основу и сделаем ее резидентной в памяти.

Пример 47.1. Резидентный обработчик прерываний от таймера

```
text    segment 'code'
        assume cs:text,ds:text
        org    256
main    proc
        jmp    init         ;Переход на инициализацию
;Поля данных резидентной секции
sym1    dw     421Eh
sym2    dw     241Eh
count   db     0
;Обработчик прерываний от таймера (взят из примера 46.2)
new_1ch proc                ;Точка входа по прерыванию 1Ch
        push   AX           ;Сохраним используемые в нем
        push   ES           ;регистры
        inc    CS:count     ;Инкремент счетчика
        test   byte ptr CS:count,03h;Пересчет на 4
        jnz    exit         ;Если не 4-е прерывание, на выход
        mov    AX,0B800h    ;Настроим ES
        mov    ES,AX        ;на видеобуфер
        mov    AX,CS:sym1   ;Получим символ с атрибутом из sym1
        xchg   AX,CS:sym2   ;Обменяем содержимое
        mov    CS:sym1,AX   ;ячеек sym1 и sym2
        mov    ES:3998,AX   ;Выведем в последнюю позицию экрана
exit:   pop    ES           ;Восстановим
        pop    AX           ;сохраненные регистры
        iret
new_1ch endp
main    endp                ;Конец резидентной части
init    proc                ;Начало секции инициализации
;Заполним вектор 1Ch
        mov    AH,25h       ;Функция заполнения вектора
        mov    AL,1Ch       ;Номер вектора
        mov    DX,offset new_1ch;Смещение резидентного обработчика
        int    21h          ;Вызов DOS
;Выведем на экран информационное сообщение
        mov    AH,09h
        mov    DX,offset mes
        int    21h
;Завершим программу, оставив ее резидентной в памяти
        mov    AX,3100h     ;Функция "Завершить и оставить в
                            ;памяти"
        mov    DX,(init-main+10Fh)/16 ;Размер резидентной части в
                            ;параграфах
        int    21h          ;Вызов DOS
init    endp
mes     db     'Резидентный обработчик загружен$'
text    ends
        end    main
```

Практически все элементы программы уже были рассмотрены; отметим только отличия от примера 45.2. Поля данных резидентной части программы переместились в начало программы после команды jmp. Это довольно естественное место для резидентных данных, потому что и при первом запуске, и при активизации сюда никогда не будет передано управление. При заполнении в секции инициализации вектора 1Ch не возникает проблем с перенастройкой регистра DS, так как в программе типа .COM все регистры указывают на единственный сегмент программы. В секции инициализации предусмотрен, как это обычно делается, вывод на экран сообщения о загрузке программы в память.

После запуска программы она остается в памяти и, активизируясь фактически аппаратными прерываниями от таймера (а более точно - программой BIOS, активизируемой аппаратными прерываниями от таймера), постоянно выводит в угол экрана мерцающий символ, который может служить признаком нормального функционирования компьютера (а не текущей программы, как это было в случае примеров 46.1 и 46.2).

Рассмотренная программа вполне работоспособна, однако она обладает весьма существенным недостатком. После того, как она загружена и осталась резидентной, удалить ее из памяти уже нельзя никакими силами. Чтобы привести компьютер в обычное состояние придется перезагружать систему. В DOS вообще не предусмотрены средства удаления резидентных программ; резидентная программа должна иметь такие средства внутри себя. Трудность здесь заключается в том, что резидентные программы практически всегда заполняют какие-то векторы прерываний, и перед выгрузкой такой программы из памяти эти векторы следует восстановить, чтобы не нарушить работоспособность системы. Однако никто, кроме самой программы, не знает, какие векторы были разрушены, и что в них было до вмешательства резидентной программы. Поэтому почти невозможно корректно выгрузить резидентную программу без ее активного участия в этом процессе. Включение в резидентную программу средств ее выгрузки (обычно по команде пользователя) требует существенного ее усложнения. Этот вопрос будет рассмотрен в статье 58.

Статья 48

Будильник

В статье 37 было рассмотрено чтение текущего времени из КМОП-микросхемы с помощью функции 02h прерывания BIOS 1Ah. Это прерывание 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. Если пользователь при этом 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. Рассмотрим программу (пример 48.1), которая считывает показания часов реального времени и устанавливает будильник на заданное время (например, через 5 секунд после текущего, чтобы было легко проверить его работу).

Для того, чтобы проверить работоспособность программы, мы должны написать программу обработчика прерывания будильника и загрузить его 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, прервав 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 видеобуферу. Именно такой обработчик и используется в примере 48.1.

Пример 48.1. Установка будильника реального времени

```
new_4ah proc            ;Наш 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;Выведем в центр экрана символ
        pop    CX           ;Восстановим
        pop    AX           ;используемые
        pop    ES           ;регистры
        iret                ;Выход из обработчика прерывания
new_4ah endp
;Главная процедура
;Сохраним вектор 4Ah в двухсловной ячейке old_4ah
        mov    AX,354Ah
        int    21h
        mov    word ptr old_4ah,BX
        mov    word ptr old_4ah+2,ES
;Установим собственный обработчик new_4ah прерывания 4Ah
        mov    AX,254Ah
        mov    DX,offset new_4ah
        push   DS           ;Сохраним DS
        push   CS           ;Настроим DS на сегмент
        pop    DS           ;команд
        int    21h
        pop    DS           ;Восстановим DS
;Получим текущее время
        mov    AH,02h       ;Фукнция получения текущего времени
        int    1Ah
;Прибавим время из поля time
        mov    SI,offset hour
        call   add_time
;Установим будильник (регистры CH, CL и DH уже настроены)
        mov    AH,06h       ;Функция установки будильника
        int    1Ah
;Остановим программу
        stop
;Сбросим будильник
        mov    AH,07h
        int    1Ah
;Восстановим системный обработчик 4Ah
        mov    AX,254Ah
        push   DS           ;Сохраним DS
        lds    DX,old_4ah
        int    21h          ;Восстановим DS
        pop    DS
;Поля данных программы
hour    db     0
min     db     0
sec     db     5h
old_4Ah dd     0
```

Из сказанного следует, что возможности обработчиков 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". Правда, возникает вопрос, что же делать, если DOS окажется занята? Однако обсуждение этого вопроса выходит за рамки настоящей статьи, так как в нашем обработчике мы к функциям DOS обращаться не будем.

Как уже отмечалось, местоположение процедуры обработчика прерывания среди других процедур программы особой роли не играет. Мы расположили обработчик в начале программы. Его программа весьма проста. Прежде всего в стеке сохраняются все используемые в нем регистры. Далее в сегментный регистр ES загружатся сегментный адрес видеобуфера и в центр экрана (со смещением 2000 байтов, или 1000 знакомест) выводится символ с кодом ASCII Fh (большая звездочка) и с атрибутом F4h (красный мерцающий по белому). После восстановления регистров программа обработчика завершается командой возврата из прерывания iret.

В главной процедуре сохраняется в двухсловной ячейке old_4ah системное содержимое вектора 4Ah, после чего вектор заполняется адресом нашего обработчика. Функция DOS 25h требует, чтобы адрес устанавливаемого обработчика прерывания находился в регистрах DS:DX, поэтому помимо занесения в регистр DX относительного адреса обработчика, сегментный регистр данных DS на время выполнения функции DOS настраивается на сегмент команд, в котором находится процедура обработчика. Далее с помощью функции 02h прерывания BIOS 1Ah из КМОП-микросхемы читается текущее время. Функция 02h возвращает число часов в регистре CH, число минут - в CL и число секунд - в DH. Все возвращаемые числа записаны в двоично-десятичном упакованном формате. С помощью разработанной ранее подпрограммы add_time к текущему времени прибавляется время, записанное в байтах массива, начинающегося с имени hour (в примере 48.1 прибавляемый интервал составляет 0 часов 0 минут 5 секунд). Таким образом, будильник должен дать сигнал через 5 секунд после запуска программы. Для установки будильника в BIOS предусмотрена функция 06h прерывания 1Ah. Она требует указания устанавливаемого времени в регистрах CH, CL и DH (часы, минуты и секунды) в двоично-десятичном формате. Поскольку подпрограмма add_time возвращает результирующее время именно в этих регистрах, они к моменту вызова функции 06h уже настроены должным образом, и остается только вызвать прерывание BIOS 1Ah.

Для того, чтобы можно было наблюдать работу будильника, в программе предусмотрена остановка до нажатия клавиши (с помощью макрокоманды stop). После прохождения через точку останова функцией 07h прерывания BIOS 1Ah будильник возвращается в исходное выключенное состояние. Если будильник не сбросить, он будет давать сигнал прерывания каждые сутки в установленное время. В этом вроде нет

ничего плохого, однако чтобы сигнал будильника отрабатывался нужным нам образом, к моменту его срабатывания в памяти должен находиться наш обработчик, а его адрес должен быть записан в вектор 4Ah.

Наконец, перед завершением программы в векторе 4Ah восстанавливается адрес исходной системной программы.

В программе 48.1 используются макрокоманда stop и подпрограмма add_time. Поэтому необходимо, во-первых, с помощью оператора include объявить транслятору имя файла с макробиблиотекой и, во-вторых, на этапе компоновки присоединить к основной программе подпрограмму add_time.

Статья 49

Контроллер прерываний и его программирование

До сих пор все наши обработчики прерываний обрабатывали, в сущности, программные прерывания. Действительно, прерывания с векторами 1Ch (таймер) и 4Ah (часы реального времени) возбуждаются не непосредственно аппаратурой компьютера, а программами BIOS с помощью команд int. В этом случае все обслуживание аппаратуры берут на себя программы BIOS. Если же прикладная программа обрабатывает аппаратные прерывания, то она должна выполнить все необходимые действия по обслуживанию, во-первых, той аппаратуры, от которой поступают прерывания, и, во-вторых, контроллера прерываний. Поэтому для составления программ обработчиков аппаратных прерываний необходимо быть знакомым с особенностями функционирования и программирования контроллера прерываний.

Сигналы аппаратных прерываний, возникающие в устройствах, входящих в состав компьютера (таймер, клавиатура, диски и проч.), поступают в процессор не непосредственно, а через два контроллера прерываний, один из которых называется ведущим, а второй - ведомым (рис. 49.1). В прежних моделях машин контроллеры представляли собой отдельные микросхемы; в современных компьютерах они входят в состав многофункциональной микросхемы периферийного контроллера.

Два контроллера используются для увеличения допустимого количества внешних устройств. Дело в том, что каждый контроллер прерываний может обслуживать сигналы лишь от 8 устройств. Для обслу-

живания большего количества устройств контроллеры можно объединять, образуя из них всеобразную структуру. В современных машинах устанавливают два контроллера, увеличивая тем самым возможное число входных устройств до 15 (7 у ведущего и 8 у ведомого контроллеров).

Рис. 49.1. Организация аппаратных прерываний.

Ко входным выводам IRQ1...IRQ7 и IRQ8...IRQ15 (IRQ - это сокращение от Interrupt Request, запрос прерывания) подключаются выводы устройств, на которых возникают сигналы прерываний. Выход INT ведущего контроллера подключается к одноименному входу микропроцессора, а выход INT ведомого - к входу IRQ2 ведущего. Основная функция контроллеров - передача сигналов запросов прерываний от внешних устройств на единственный вход прерываний микропроцессора. При этом, кроме сигнала INT, контроллеры передают в микропроцессор по линиям данных номер вектора, который образуется в контроллере путем сложения базового номера, записанного в одном из его регистров, с номером входной линии, по которой поступил запрос прерывания. Получив сигнал INT и номер вектора, процесс извлекает из вектора адрес обработчика прерывания и передает управление на этот обработчик.

Номера базовых векторов заносятся в контроллеры автоматически в процессе начальной загрузки компьютера. Ведущий контроллер программируется через порты 20h и 21h, ведомый - A0h и A1h.

Поскольку базовый вектор ведущего контроллера всегда равен 8, а базовый вектор ведомого - 70h, номера векторов, закрепленных за аппаратными прерываниями, лежат в диапазонах 8h...Fh и 70h...77h. Очевидно, что номера векторов аппаратных прерываний однозначно связаны с номерами линий, или уровнями IRQ, а через них - с конкретными устройствами компьютера. В таблице 49.1 приведен перечень аппаратных векторов и закрепленных за ними устройств.

Таблица 49.1. Соответствие векторов прерываний устройствам компьютера.

Уровень прерывания	Вектор прерывания	Устройство
IRQ0	08h	Таймер
IRQ1	09h	Клавиатура
IRQ2	0Ah	Вход от ведомого
IRQ8	70h	КМОП-микросхема
IRQ9	71h	Перенаправлено на int 0Ah
IRQ10	72h	Зарезервировано
IRQ11	73h	Зарезервировано
IRQ12	74h	Мышь (PS/2)
IRQ13	75h	Исключение сопроцессора
IRQ14	76h	Жесткий диск
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 (для примера рассмотрим ведущий контроллер). Логически в ней можно выделить четыре основных узла: регистр входных запросов, регистр маски, схему приоритетов и регистр обслуживаемых запросов (рис. 49.2). Все эти узлы восьмибитовые, по одному биту на каждый входной сигнал.

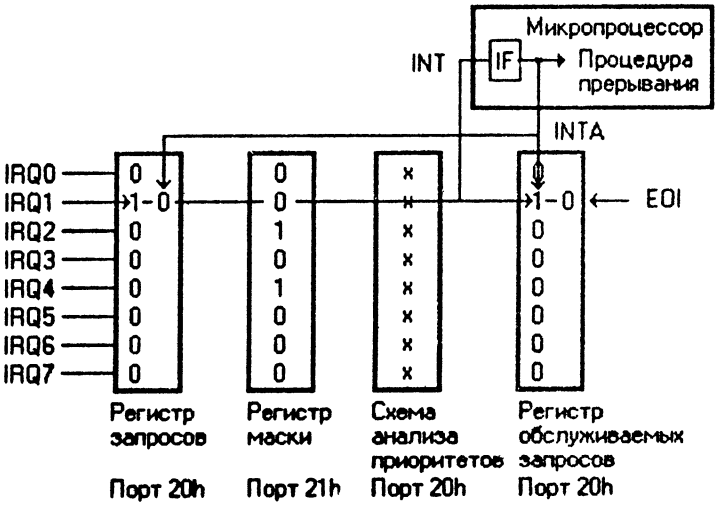


Рис. 49.2. Логическая структура контроллера прерываний.

Сигнал запроса прерывания IRQ от устройства (на рис. 49.2 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, программируемый через порт 21h. 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 - IRQ7.

Ведомый контроллер всегда подключается к входу IRQ2 ведущего, поэтому все приоритеты ведомого контроллера располагаются между приоритетами уровней IRQ1 и IRQ3 ведущего и образуется такая цепочка приоритетов:

```
IRQ0 ... IRQ1 ── IRQ8 ... IRQ15 ──── IRQ3 ... IRQ7
  Наивысший          Приоритеты          Низший
  приоритет          ведомого            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 обработки таких наложенных прерываний определяет программист путем соответствующего построения обработчика прерывания.

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, сброс флага IF командой cli запрещает все аппаратные прерывания.

Микропроцессор, получив сигнал INT, отвечает на него выходным сигналом INTA (INTerrupt Acknoledge, подтверждение прерывания), который поступает в контроллер прерываний и выполняет там два действия: устанавливает бит регистра обслуживаемых запросов, разрешенный сигналом запроса прерывания, и сбрасывает бит регистра запросов. Таким образом, 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 запом-

нится 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. 49.3 для конкретного случая прихода запроса прерывания уровня 1.

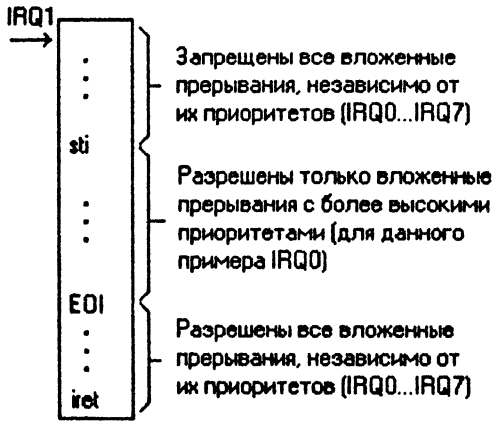


Рис. 49.3. Обобщенная структура программы обработки прерывания.

Поскольку процессор, получив сигнал INT, сбрасывает флаг IF, при входе в обработчик все прерывания оказываются запрещенными и программа не может быть прервана внешними сигналами. Команда sti, выполненная в программе, устанавливает флаг IF и разрешает прохож-дение запросов прерываний в процессор. Однако все уровни преры-ваний, начиная с текущего, остаются заблокированными в контроллере. В результате работа обработчика может быть прервана только при поступлении запроса прерывания более высокого приоритета (в рас-сматриваемом 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 реентерабельность. Лучше, однако избегать возникновения такой ситуации.

Практически структуру программы обработки прерывания выбирают исходя из конкретных условий (рис. 49.4).

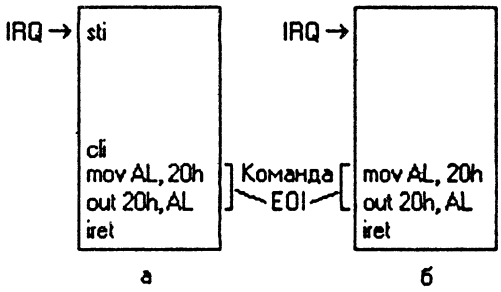


Рис. 49.4. Типичные структуры программы обработки прерываний: при разрешенных вложенных прерываниях (а) и при запрещенных вложенных прерываниях (б).

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. 49.4,а). Однако сигнал прерывания того же (или более низкого) уровня может придти 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 выполняют команду 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 восстанав-ливает 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. 49.4,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 аппаратных прерываний уровней 8...15 следует предусматривать посылку команд EOI в оба контроллера:

```
mov     AL,20h
out     20h,AL
out     A0h,AL
```

Рассмотрим несколько примеров программирования контроллера прерываний.

Пример 49.1. Запрещение прерываний от таймера

```
in      AL,21h      ;Прочитаем текущую маску
or      AL,1        ;Установим добавочно бит 0
out     21h,AL      ;Пошлем в регистр маски
```

Выполнение приведенного примера приведет к остановке системного таймера. Это легко проконтролировать с помощью часов программы Norton Commander или с помощью команды DOS TIME.

Чтобы не нарушать работу машины, после программы 49.1 следует выполнить программу 49.2.

Пример 49.2. Разрешение прерываний от таймера

```
in      AL,21h      ;Прочитаем текущую маску
and     AL,0FEh     ;Сбросим выборочно бит 0
out     21h,AL      ;Пошлем в регистр маски
```

Таким же образом можно, например, запретить прерывания от контроллера гибких дисков и для контроля попробовать обратиться к дискете, хотя бы с помощью программы Norton Commander.

Рассмотрим на конкретном примере некоторые возможности программирования контроллера прерываний. Как мы уже выяснили, при переходе на программу обработки аппаратного прерывания бит регистра обслуживаемых запросов, соответствующий пришедшему запросу, установлен в 1, что приводит к блокированию данного и всех нижележащих уровней прерываний. Для снятия блокировки используется команда конца прерывания EOI, обычно посылаемая в контроллер перед самым завершением обработчика прерываний. В примере 49.3 с помощью

программы обработки аппаратного прерывания от таймера (вектор 08h) наглядно демонстрируется состояние регистра обслуживаемых запросов до и после команды EOI. 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).

Пример 49.3. Изучение контроллера прерываний

```
text    segment 'code' public
        assume cs:text,ds:text
main    proc
;Настроим DS на сегмент команд, чтобы можно было пользоваться
;обозначениями полей данных без префикса замены сегмента
        push   CS
        pop    DS
;Сохраним вектор 08h
        mov    AX,3508h
        int    21h
        mov    word ptr old_08h,BX
        mov    word ptr old_08h+2,ES
;Заполним вектор 08h
        mov    AH,25h
        mov    AL,08h
        mov    DX,offset new_08h
        int    21h
        pop    DS
;Остановим программу до нажатия клавиши ,
        mov    AH,01h
        int    21h
;Перед завершением программы восстановим содержимое вектора 08h
        lds    DX,old_08h
        mov    AX,2508h
        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    AL,0Bh      ;Пошлем в контроллер прерываний
        out    20h,AL      ;команду 0Bh - разрешение чтения
                           ;регистра обслуживаемых запросов
        jmp    $+2         ;Небольшая
        jmp    $+2         ;задержка
        in     AL,20h      ;Прочитаем регистр обслуживаемых
                           ;запросов
        add    AL,'0'      ;Преобразуем в символьную форму
        mov    AH,1Eh      ;Атрибут
        mov    ES:1680,AX  ;Выведем на экран
        mov    AL,20h      ;Пошлем в ведущий контроллер
        out    20h,AL      ;команду EOI
```

```
        jmp     $+2             ;Небольшая
        jmp     $+2             ;задержка
        in      AL,20h          ;Снова прочитаем регистр
                                ;обслуживаемых запросов
        add     AL,'0'          ;Преобразуем в символьную форму
        mov     AH,4Eh          ;Атрибут для наглядности другой
        mov     ES:1690,AX      ;Выведем на экран в другое место
        pop     ES              ;Восстановим сохраненные
        pop     AX              ;регистры
        iret                    ;Возврат в прерванную программу
new_08h endp
text    ends
end     main
```

В программе на этапе инициализации выполняются стандартные действия по сохранению системного содержимого вектора 08h и занесению в вектор адреса нашего обработчика new_08h, после чего программа останавливается до нажатия любой клавиши. Тем временем приходящие от таймера прерывания активизируют наш обработчик. После сохранения регистров AX и ES регистр ES настраивается на адрес видеобуфера. В порт 20h контроллера прерываний посылается код команды 0Bh, разрешающей чтение регистра обслуживаемых запросов. После команды 0Bh чтение (в том числе неоднократное) из порта 20h будет вводить в программу содержимое этого регистра. Между посылкой в порт команды и чтением из порта предусмотрена небольшая задержка (две команды jmp) для того, 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.

После чтения в регистр AL содержимого регистра обслуживаемых запросов (в нашем случае прерываний по уровню IRQ0 в нем должен быть код 01h) к содержимому AL добавляется код ASCII символа "0", чтобы на экран выводились не малонаглядные пиктограммы, а символы "0" или "1". В регистр AH засылается произвольный атрибут символа и получившийся двухбайтовый код выводится на экран с некоторым (1680/2=840 знакомест) смещением от начала экрана.

Далее в порт 20h контроллера посылается код команды EOI, после чего снова читается и выводится на экран (с другим атрибутом и в другое место) содержимое регистра обслуживаемых запросов. После восстановления сохраненных ранее регистров обработчик завершается командой iret, возвращающей управление в прерванную программу.

Следует заметить, что наш обработчик, активизируясь аппаратным прерыванием и возвращая управление в прерванную программу (нашу же) "похищает" такты системного таймера. Пока программа находится в памяти (до нажатия клавиши), отсчет времени не производится.

Статья 50

Взаимодействие прикладных и системных обработчиков прерываний

До сих пор мы сталкивались с обработкой прерываний, специально предназначенных для использования в прикладных программах. Таковы прерывания с векторами 1Ch (системный таймер) или 4Ah (будильник). Команды int, инициирующие эти прерывания, включены в системные программы BIOS, обслуживающие соответствующие аппаратные средства компьютера, и структура прикладного обработчика такого прерывания оказывается весьма простой:

```
new_4ah proc
        ...
        iret
new_4ah endp
```

Алгоритм прикладного обработчика прерывания в этом случае определяется исключительно нуждами пользователя, поскольку все действия по обслуживанию инициирующей данное прерывание аппаратуры и по реализации стандартной реакции системы выполняются системными программами BIOS. Завершающая обработчик команда iret передает управление системной программе, которая продолжает свою работу.

В тех случаях, когда пользователь хочет вмешаться в обработку прерывания, обслуживаемого системой, и не имеющего "выхода на пользователя", структура и алгоритм работы прикладного обработчика усложняются. Часто пользователю требуется лишь незначительно изменить или дополнить системный алгоритм. В этих случаях используется методика сцепления прикладного обработчика с системным, когда программа пользователя выполняет свою часть обработки 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
        ...    ;Прикладная обработка
        iret
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. 50.1). Именно такая структура данных должна быть на верху стека, чтобы команда iret, которой завершается любая программа обработки прерываний, могла вернуть управление в прерванный процесс.

Рис. 50.1. Стек прерванной программы в процессе выполнения прикладного обработчика прерываний. CS1 - сегментный адрес прерванного процесса; IP1 - смещение точки возврата в прерванный процесс; CS2 - сегментный адрес обработчика; IP2 - смещение точки возврата в прикладной обработчик.

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.

Системный обработчик, закончив обработку данного прерывания, завершается командой iret. Эта команда забирает из стека три верхних слова и осуществляет переход по адресу CS2:IP2, т.е. на продолжение прикладного обработчика. Завершающая команда нашего обработчика iret снимает со стека три верхних слова и передает управление по адресу CS1:IP1.

Если прикладная обработка должна выполняться до системной, структура прикладного обработчика будет иной:

```
new_int proc
...     ;Прикладная обработка
        jmp    CS:old_int ;В системный обработчик без возврата
new_int endp
```

Завершающая команда перехода передает управление в системный обработчик (не затрагивая стек), который далее выполняется 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
        call    CS:old_int ;В системный обработчик 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 нужна:
        jmp sys
                ;Если не нужна:
        ...     ;Прикладная обработка
        iret
sys:    jmp     CS:old_int
new_int endp
```

В последующих статьях будут приведены примеры обработчиков прерываний разного рода.

Статья 51

Резидентный обработчик прерываний от клавиатуры с подключением до системного обработчика

Практически любая программа, в которой предусмотрено управление ходом ее выполнения с помощью команд, подаваемых с клавиатуры, имеет в своем составе обработчик прерываний от клавиатуры. В зависимости от стоящих перед ним задач, обработчик может подключаться до системного, выполняя обработку скен-кодов нажимаемых

клавиш, или после системного, работая в этом случае с кодами ASCII, возникающими на выходе системного обработчика. Нередки случаи, когда прикладной обработчик выполняет часть своих функций до системного, а часть - после. Настоящая и несколько последующих статей посвящены этому важному для прикладного программиста вопросу.

Для того, чтобы написать обработчик прерываний от клавиатуры, необходимо хорошо представлять, каким образом вводятся, куда попадают и как обрабатываются символы, вводимые с клавиатуры. Процесс взаимодействия системы с клавиатурой показан на рис. 51.1.

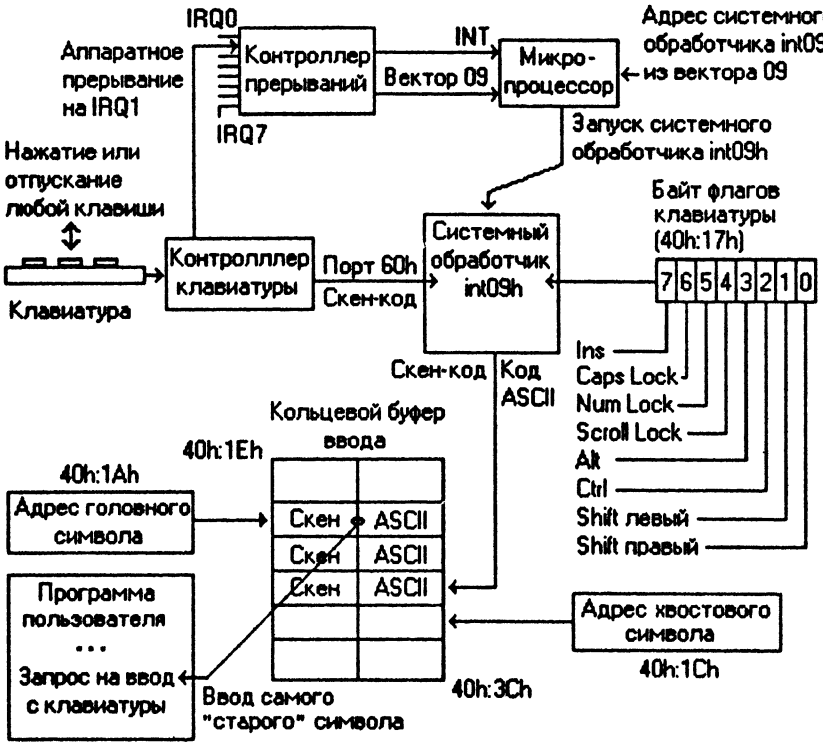


Рис. 51.1. Процесс взаимодействия системы с клавиатурой.

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) с номером 60h. Код клавиши, поступающий в порт, называется скен-кодом и является, по существу, порядковым номером кла-

виши. При этом каждой клавише присвоены два скен-кода, отличающиеся друг от друга на 80h. Один скен-код (меньший, код нажатия) засылается контроллером в порт 60h при нажатии клавиши, другой (больший, код отпускания) - при ее отпускании.

Скен-код однозначно указывает на нажатую клавишу, однако по нему нельзя определить, работает ли пользователь на нижнем или верхнем регистре. 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.

Как нажатие, так и отпускание любой клавиши вызывает сигнал аппаратного прерывания, заставляющий процессор прервать выполняемую программу и перейти на программу системного обработчика прерываний от клавиатуры, входящего в систему BIOS. Поскольку обработчик вызывается через вектор 09h, его иногда называют программой int09h.

Программа int09h, помимо порта 60h, работает еще с двумя областями оперативной памяти: кольцевым буфером ввода, располагаемым по адресам от 40h:1Eh до 40h:3Dh, куда в конце концов помещаются коды ASCII нажатых клавиш, и байтом флагов клавиатуры, находящимся по адресу 40h:17h, где фиксируется состояние управляющих клавиш (<Shift>, <Caps Lock>, <Num Lock> и др.).

Программа int09h, получив управление в результате прерывания от клавиатуры, считывает из порта 60h скен-код и анализирует его значение. Если скен-код принадлежит одной из управляющих клавиш, и, к тому же, представляет собой код нажатия, в байте флагов клавиатуры устанавливается бит (флаг), соответствующий нажатой клавише. Например, при нажатии правой клавиши <Shift> в байте флагов устанавливается бит 0, при нажатии левой клавиши <Shift> - бит 1, при нажатии любой клавиши <Ctrl> - бит 2 и т.д. Биты флагов сохраняют свое состояние, пока клавиши (по одиночке или в любых комбинациях) остаются нажатыми. Если управляющая клавиша отпускается, программа int09h получает скен-код отпускания и сбрасывает соответствующий бит в байте флагов. Кроме состояния указанных клавиш, в байте флагов фиксируются еще режимы <Scroll Lock>, <Num Lock>, <Caps Lock> и <Insert> (см. рис. 51.1).

Компьютеры PC/AT имеют второй байт флагов клавиатуры, находящийся по адресу 40h:18h, и отражающий состояние управляющих клавиш на расширенной (101-клавишной) клавиатуре. Назначение битов этого байта приведено на рис. 51.2.

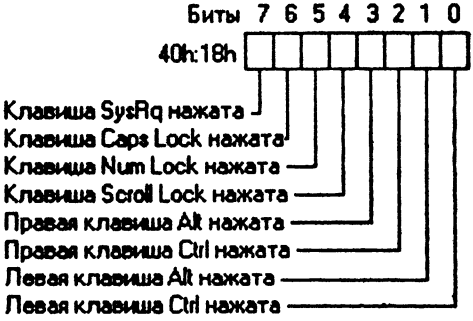


Рис. 51.2. Второй байт флагов клавиатуры.

При нажатии обычной, не управляющей клавиши, программа int09h 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", "1" и "!", "2" и "@" и т.д.), то каждому скен-коду соответствуют, как минимум, два кода ASCII. В процессе трансляции программа int09h 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 клавише <Shift> образуется код 1071h, поскольку в такой ситуации клавиша <Shift> на время нажатия переводит клавиатуру в режим нижнего регистра (строчные буквы).

Полученный в результате трансляции двухбайтовый код засылается программой int09h в кольцевой буфер ввода, который служит для синхронизации процессов ввода данных с клавиатуры и приема их выполняемой компьютером программой. Объем буфера составляет 16 слов, при этом 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. Оба адреса представляют собой смещения относительно начала области данных BIOS, т.е. числа от 1Eh до 3Ch. В начале работы, когда буфер пуст, оба указателя - и хвостовой, и головной, указывают на первую ячейку буфера.

Программа int09h, сформировав двухбайтовый код, помещает его в буфер по адресу, находящемуся в хвостовом указателе, после чего 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 обратиться для этого к каким-либо системным средствам - функциям ввода с клавиатуры DOS (прерывание 21h) или BIOS (прерывание 16h). Системные программы с помощью драйвера клавиатуры (точнее говоря, объединенного драйвера клавиатуры и экрана, так называемого драйвера консоли с именем CON) считывают 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, затем 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 поставила запрос на ввод символа с клавиатуры, то драйвер консоли будет ждать поступления кода в буфер, после чего этот код будет передан в программу. Если же хвостовой указатель, перемещаясь по буферу в процессе его заполнения, подошел к головному указателю "с обратной стороны" (это произойдет, если оператор нажимает на клавиши, а выполняемая в настоящий момент программа не обращается к клавиатуре), прием новых кодов блокируется, а нажатие на клавиши возбуждает предупреждающие звуковые сигналы.

Если компьютер не выполняет никакой программы, то активной является программа командного процессора COMMAND.COM. Активность COMMAND.COM заключается в том, что он, поставив запрос к DOS на ввод с клавиатуры (с помощью функции 0Ah прерывания 21h) ожидает ввода с клавиатуры очередной команды пользователя. Как только в кольцевом буфере ввода появляется код символа, функция 0Ah переносит его в внутренний буфер DOS, очищая при этом кольцевой буфер ввода, а также выводит символ на экран. При получении кода клавиши <Enter> (0Dh) функция 0Ah завершает свою работу, а командный процессор предполагает, что ввод команды закончен, анализирует содержимое буфера DOS и приступает к выполнению введенной команды. При этом командный процессор работает практически лишь с младшими половинами двухбайтовых кодов символов, именно, с кодами 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 эхоконтроля 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 int09h) 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 отображаемые на экране символы. Это, например, функциональные клавиши <F1>, <F2>...<F10>; клавиши управления курсором <Home>, <End>, <PgUp>, <PgDn>, <Стрелка вправо>, <Стрелка вниз> и др. При нажатии этих клавиш в кольцевой буфер ввода засылается расширенный код ASCII, в котором младший байт равен нулю, а старший является скен-кодом нажатой клавиши. Расширенный коды ASCII поступают в буфер ввода и в случае нажатия комбинаций управляющих и функциональных клавиш, например, <Shift>/<F1>, <Ctrl>/<Home> (на дополнительной цифровой клавиатуре), <Alt>/<Insert> и др. 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     <Ctrl>/<F1>  5E00h
<Alt>/<F1> 6800h     <Shift>/<F1> 5400h
```

Таблицы расширенных кодов ASCII были приведены в статье 13.

Итак, прерывание, возникающее при нажатии или отпускании любой клавиши, обрабатывается по относительно сложному алгоритму системным обработчиком, содержащимся в BIOS. Рассмотрим примеры вмешательства в этот процесс. Ниже приведен пример прикладной программы, выполняющей некоторую обработку поступающих с клавиатуры данных еще до активизации системного обработчика.

Пример 51.1. Перехват и уничтожение скен-кода клавиши <F10>

```
text    segment 'code'
        assume cs:text,ds:text
        org   256
main    proc
        jmp   init
;Поля данных резидентной секции
old_09h dd    0;Ячейка для сохранения системного вектора 09h
;Наш обработчик от клавиатуры
new_09h proc
        push  AX          ;Сохраним используемый регистр
        in    AL,60h      ;Введем скен-код
        cmp   AL,44h      ;Скен-код <F10>?
        je    hotkey      ;Да, на уничтожение
        pop   AX          ;Нет, восстановим регистр
        jmp   CS:old_09h ;и в системный обработчик
hotkey:
;Разрешим дальнейшую работу клавиатуры
        in    AL,61h      ;Введем содержимое порта B
        or    AL,80h      ;Подтвердим прием кода, добавив
        out   61h,AL      ;бит 80h к содержимому порта B
        and   AL,7Fh      ;Снова разрешим работу клавиатуры,
        out   61h,AL      ;сбросив в порте B бит 80h
;Пошлем в контроллер прерываний команду EOI
        mov   AL,20h
        out   20h,AL
        pop   AX          ;Восстановим регистр
        iret              ;Выход из прерывания
new_09h endp
end_res=$       ;Смещение конца резидентной части программы
main    endp
;Процедура инициализации
init    proc
;Сохраним вектор 09h
        mov   AX,3509h
        int   21h
        mov   word ptr CS:old_09h,BX
        mov   word ptr CS:old_09h+2,ES
;Заполним вектор 09h
        mov   AX,2509h
        mov   DX,offset new_09h
        int   21h
;Выведем на экран информационное сообщение
        write mes
;Завершим программу, оставив ее резидентной в памяти
        mov   AX,3100h
        mov   DX,(end_res-main+10Fh)/16
        int   21h
init    endp
mes     db    'Резидентный обработчик прерывания 09h загружен$'
text    ends
        end   main
```

Структура примера 51.1 традиционна для резидентных обработчиков аппаратных прерываний. Программа написана в формате минимальной

модели памяти, поэтому после трансляции и компоновки обычным образом ее следует преобразовать в формат .COM с помощью системной утилиты EXE2BIN.

В процедуре инициализации, на которую передается управление при запуске программы, сохраняется системное содержимое вектора прерываний от клавиатуры 09h, вектор заполняется адресом прикладного обработчика new_09h, на экран выводится информационное сообщение о загрузке программы и программа завершается функцией DOS 31h с оставлением в памяти ее резидентной части.

При нажатии любой клавиши (а также и при ее отпускании) процессор, сохранив в стеке текущей программы вектор возврата, передает управление нашему обработчику new_09h. В нем прежде всего сохраняется в стеке единственный используемый регистр AX, после чего из порта контроллера клавиатуры 60h вводится скен-код нажатой клавиши, который далее сравнивается с кодом 44h клавиши <F10>. Если была нажата клавиша, отличная от <F10>, восстанавливается регистр AX и командой дальнего косвенного перехода jmp CS:old_09h управление передается той программе, адрес которой находился в векторе 09h в момент загрузки нашей программы. В реальной ситуации в векторе 09h скорее всего будет содержаться адрес русификатора, который, естественно, перехватывает все прерывания от клавиатуры с целью замены кодов ASCII латинских букв на коды ASCII русских, если в данный момент включен режим кириллицы.

Если наш обработчик получил управление в результате нажатия клавиши <F10>, осуществляется переход на метку hotkey. Здесь прежде всего следует на короткое время установить и затем сбросить старший бит порта B контроллера клавиатуры. Этим действием мы сообщаем контроллеру о приеме скен-кода и разрешаем дальнейшую работу клавиатуры. Далее в контроллер клавиатуры посылается команда конца прерывания EOI и обработчик, после восстановления сохраненного регистра, завершается командой iret. Управление передается не в системный обработчик, а в ту программу, которая была прервана нажатием клавиши (скорее всего, в командный процессор COMMAND.COM или в Norton Commander), код же клавиши <F10> не обрабатывается системными средствами, не поступает в буфер ввода и теряется. Результат работы нашего обработчика особенно нагляден при запуске его из программы Norton Commander. Все будет работать, как обычно, за исключением того, что вы не сможете никаким образом выгрузить из памяти программу Norton Commander клавишей F10.

Поскольку в программе обработчика отсутствует команда sti, она работает при запрещенных прерываниях. Программа слишком коротка, чтобы это могло иметь какие-либо неприятные последствия; в то же время мы не нарушаем работу системного обработчика, который в

обычном случае получает управление при запрещенных (процессором) прерываниях и, возможно, использует это обстоятельство.

Усложним пример 51.1, задав в качестве горячей клавиши сочетание <Alt>/<F10>. Для того, чтобы обнаружить нажатую клавишу <Alt>, следует обратиться к слову флагов клавиатуры. Этот пример отличается от предыдущего только программой обработчика прерывания.

Пример 51.2. Перехват <Alt>/<F10>

```
;Наш обработчик от клавиатуры
new_09h proc
        push   AX          ;Сохраним используемый регистр
        in     AL,60h      ;Введем скен-код
        cmp    AL,44h      ;Скен-код <F10>?
        je     go1         ;Да, на продолжение анализа
exit1:  pop    AX          ;Нет, восстановим регистр
        jmp    CS:old_09h  ;и в системный обработчик
go1:    push   ES          ;Сохраним используемый регистр
        mov    AX,40h      ;Настроим ES на начало
        mov    ES,AX       ;области данных BIOS
        mov    AL,ES:[17h];Получим слово флагов клавиатуры
        pop    ES          ;Восстановим ES - он больше не нужен
        cmp    AL,08h      ;Уже нажата <Alt>?
        je     hotkey      ;Да, на выход в прерванную программу
        jmp    exit1       ;Нет, в системный обработчик
hotkey:
;Разрешим дальнейшую работу клавиатуры
        ...
;Пошлем в контроллер прерываний команду EOI
        ...
        pop    AX          ;Восстановим регистр
        iret               ;Выход из прерывания
```

Серьезным недостатком нашего обработчика является невозможность его выгрузки из памяти. Восстановить исходное состояние компьютера можно только выполнив операцию его перезагрузки. В последующих статьях мы рассмотрим процедуры выгрузки из памяти резидентной программы по команде пользователя.

Статья 52

Резидентный обработчик прерываний от клавиатуры с подключением после системного обработчика

Во многих случаях прикладная обработка прерывания (аппаратного или программного) должна выполняться в качестве дополнения к системной. Такая постановка задачи типична, например, для русификаторов клавиатуры. Системная программа BIOS обработки прерываний от клавиатуры, основное назначение которой - преобразование поступающих с клавиатуры скен-кодов в коды ASCII латинских и международных символов, выполняет, кроме этой основной задачи, массу дополнительных операций: следит за указателями кольцевого буфера клавиатуры и модифицирует их; отслеживает нажатие сочетания <Ctrl>/<Break> и выполняет при его вводе специфические действия; позволяет вводить в кольцевой буфер коды непосредственно в цифровом виде с помощью нажатия клавиши <Alt> и цифр на цифровой клавиатуре и т.д. При установке на компьютере русификатора можно, конечно, полностью отказаться от использования системной программы обслуживания клавиатуры, включив все упомянутые функции в русификатор, однако это заметно усложнит его. Более продуктивно выполнить программу русификатора, как дополнение к системному обработчику, возложив на него лишь корректировку кодов ASCII в кольцевом буфере, если включен режим кириллицы. В этом случае русификатор должен подключаться к выходу системного обработчика.

Рассмотрим простой пример программы такого рода. Предположим, что пользователь, работающий с простым текстовым редактором, часто пользуется символами штриховки, включая их в создаваемые им документы. В этом случае целесообразно закрепить коды ASCII этих символов (▒ - 176, ▓ -177, █ -178) за какими-то клавишами или сочетаниями клавиш, заменив стандартные коды ASCII этих клавиш на приведенные выше. Лучше, конечно, использовать именно сочетания клавиш, чтобы не лишаться обычных символов. Рассмотрим резидентную программу, выполняющую эту операцию. Для символов штриховки в ней используются сочетания клавиши <Alt> с тремя нижними левыми клавишами клавиатуры: <Alt>/<Z>,<Alt>/<X> и <Alt>/<C>.

Поскольку процедура установки прикладного обработчика и оставление его резидентным в памяти уже неоднократно описывалась, в

примере 52.1 приведена только программа собственно обработчика (его резидентная часть).

Пример 52.1. Замена сочетаний <Alt>/<Z>,<Alt>/<X> и <Alt>/<C> кодами символов штриховки

```
;Поля данных резидентной секции
old_09h dd      0 ;Ячейка для сохранения системного вектора 09h
;Наш обработчик от клавиатуры
new_09h proc
        pushf               ;(1)Создадим в стеке структуру для iret
        call    CS:old_09h  ;(2)Вызов системного обработчика
        push    AX          ;(3)Сохраним
        push    BX          ;(4)используемые
        push    ES          ;(5)регистры
        mov     AX,40h      ;(6)Настроим ES на сегментный
        mov     ES,AX       ;(7)адрес области данных BIOS
        mov     BX,ES:[1Ch];(8)Адрес нового хвоста
        dec     BX          ;(9)Сместимся назад к последнему
        dec     BX          ;(10)введенному символу
        cmp     BX,1Eh      ;(11)Хвост не вышел за пределы буфера?
        jae     go          ;(12)Нет, значит, он был внутри буфера
        mov     BX,3Ch      ;(13)Хвост после вычитания 2 вышел за
                            ;пределы буфера, значит, он был в его
                            ;начале, а последний введенный символ
                            ;находится в самом конце буфера
go:     mov     AX,ES:[BX] ;(14)Получим последний символ из буфера
        cmp     AX,2C00h    ;(15)Был введен расширенный код
                            ;ASCII сочетания <Alt>/<Z>?
        jne     go1         ;(16)Нет, будем проверять еще
        mov     word ptr ES:[BX],00B0h ;(17)Да, заменим код в
                            ;буфере на код редкой штриховки
        jmp     outout      ;(18)И на выход из обработчика
go1:    cmp     AX,2D00h    ;(19)Был введен расширенный код
                            ;ASCII сочетания <Alt>/<X>?
        jne     go2         ;(20)Нет, будем проверять еще
        mov     word ptr ES:[BX],00B1h;(21)Да, заменим код в
                            ;буфере на код средней штриховки
        jmp     outout      ;(22)И на выход из обработчика
go2:    cmp     AX,2E00h    ;(23)Был введен расширенный код
                            ;ASCII сочетания <Alt>/<C>?
        jne     outout      ;(24)Нет, на выход
        mov     word ptr ES:[BX],00B2h;(25)Да, заменим код в буфере
                            ;на код густой штриховки
outout: pop     ES          ;(26)Восстановим
        pop     BX          ;(27)используемые
        pop     AX          ;(28)регистры
        iret                ;(29)Выход из прерывания
new_09h endp
end_res=$                   ;Смещение конца резидентной части
                            ;программы
main    endp
```

Первыми же командами нашей программы (предложения 1-2) управление передается в системный обработчик. При этом в стеке создается структура (содержимое регистра флагов от команды pushf и двух-

словный адрес возврата на предложение 3 от команды дальнего косвенного вызова call CS:old_09h), которая заставит завершающую команду iret системного обработчика вернуть управление нашей программе. Далее в стеке сохраняются используемые в программе регистры, а сегментный регистр ES настраивается на начало области данных BIOS (физический адрес 400h, сегментный адрес 40h) и начинается работа со входным буфером клавиатуры.

Будем считать, что перед вводом данного символа буфер был пуст (рис. 52.1).

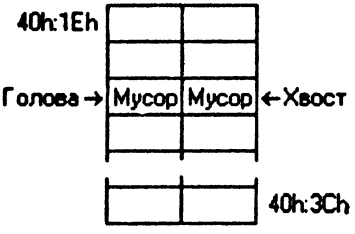


Рис. 52.1. Исходное состояние кольцевого буфера.

Фактически в слове, на которое указывают оба указателя, находится код символа, введенного ранее и уже изъятого программой из буфера. Очередное нажатие клавиши вызывает нашу программу, а из нее - системный обработчик, который заносит двухбайтовый код нажатой клавиши в хвостовой элемент, перемещая хвостовой указатель на следующее слово буфера (рис. 52.2).

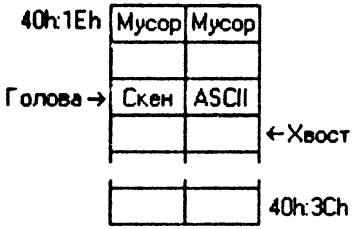


Рис. 52.2. В кольцевой буфер занесен один символ.

Казалось бы, получить код нажатой клавиши можно либо по адресу в головном указателе, либо вычтя 2 из адреса хвостового указателя. Однако ни то, ни другое неверно.

Головной указатель действительно указывает на последний сохраненный в буфере код, лишь если при запуске программы буфер был пуст. Если же в процессе загрузки программы пользователь нажимал какие-либо клавиши, их коды были помещены в буфер клавиатуры, и к моменту активизации нашей программы головной указатель будет указывать не на пустую ячейку буфера, а на самый старый из введенных символов. При этом по мере занесения в буфер новых символов, обработке будет подвергаться все тот же самый старый символ. Поэтому более надежно определять местонахождение последнего символа в буфере по адресу в хвостовом указателе. Однако, учитывая кольцевой характер буфера, не всегда этот адрес будет на 2 больше адреса последнего символа. В том случае, когда очередная свободная ячейка буфера находилась в самом его конце, по адресу 40h:3Ch, после занесения туда очередного кода хвостовой указатель переместится в начало буфера, указывая на адрес 1Eh. Поэтому в

нашей программе после вычитания 2 из адреса хвостового элемента (предложения 9-10), проверяется, находится ли полученное значение в пределах буфера. Если оно равно или больше 1Eh, командой jae выполняется переход на продолжение программы. Если же полученное значение оказалось меньше 1Eh, это значит, что хвостовой указатель указывает на самое начало буфера, а занесенный только что в буфер код находится в самом его конце (рис. 52.3). В этом случае в BX заносится адрес последнего символа в явном виде (предложение 13).

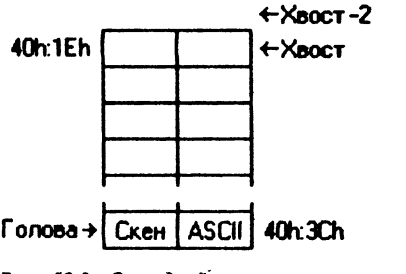


Рис. 52.3. Очередной символ попал в конец буфера..

Дальнейший алгоритм программы вполне очевиден. Двухбайтовый код нажатой клавиши извлекается из буфера (предложение 14) и последовательно сравнивается с двухбайтовыми кодами сочетаний <Alt>/<Z>, <Alt>/<X> и <Alt>/<C>. Если нажато одно из указанных сочетаний, код в кольцевом буфере заменяется на код ASCII соответствующего символа псевдографики (естественно, без смещения указателей) и осуществляется переход на метку outout для завершения программы. Если все три операции сравнения не дали положительного результата, код в буфере остается без изменения.

По метке outout восстанавливаются сохраненные регистры и командой iret управление передается в прерванную программу.

Статья 53

Резидентный обработчик прерываний от клавиатуры с подключением как до, так и после системного

Иногда при подцеплении нашего обработчика к системному бывает удобно некоторую часть прикладной обработки выполнить до системного обработчика, а некоторую часть после. Модифицируем программу из предыдущей статьи, использовав другой способ определения фактического адреса последнего символа в буфере. В примере 53.1 приве-

дено только начало резидентной части программы. Весь остальной текст не претерпел никаких изменений.

Пример 53.1. Назначение сочетаниям <Alt>/<Z>, <Alt>/<X> и <Alt>/<C> кодов штриховок

```
;Поля данных резидентной секции
old_09h dd     0;Ячейка для сохранения вектора 09h
tail    dw     0;Ячейка для сохранения хвостового указателя
;Наш обработчик от клавиатуры
new_09h proc
        push   AX          ;Сохраним используемые
        push   ES          ;регистры
        mov    AX,40h      ;Настроим ES
        mov    ES,AX       ;на область данных BIOS
        mov    AX,ES:[1Ch];Адрес хвоста перед обработкой данного
                           ;нажатия
        mov    CS:tail,AX ;Сохраним его для дальнейшего анализа
        pop    ES          ;Восстановим
        pop    AX          ;регистры
        pushf              ;и перейдем в системный обработчик
        call   CS:old_09h ;с возвратом в нашу программу
        push   AX          ;Вернулись в нашу программу.
        push   BX          ;Сохраним используемые
        push   ES          ;регистры
        mov    AX,40h      ;Настроим ES
        mov    ES,AX       ;на область данных BIOS
        mov    BX,CS:tail ;Адрес старого хвоста
        cmp    BX,ES:[1Ch]     ;Хвост сместился?
        je     outout      ;Нет, на выход
        mov    AX,ES:[BX] ;Получим занесенный код
        cmp    AX,2C00h    ;Был введен расширенный код ASCII
                           ;сочетания <Alt>/<Z>?
        jne    go1         ;Нет, будем проверять еще
        ;и т.д.
```

В полях данных резидентной части программы, кроме ячейки для адреса системного обработчика, предусмотрена ячейка для хранения "предыдущего" содержимого хвостового указателя. После активизации по нажатию клавиши и еще до передачи управления системному обработчику содержимое хвостового указателя запоминается в ячейке tail. Естественно, что предварительно, во-первых, сохраняются используемые регистры, и во-вторых, один из сегментных регистров (в данном случае ES) настраивается на адрес области данных BIOS. После восстановления регистров и подготовки стека (команда pushf) управление передается системному обработчику.

После возврата из системного обработчика сохраняются используемые регистры и настраивается ES. Сохраненное содержимое хвостового указателя сравнивается с текущим значением, полученным из ячейки 40h:1Ch. Если системный обработчик не сместил хвостовой указатель, это значит, что была нажата какая-то управляющая клавиша (<Shift>,

<Alt> и т.д.). В этом случае необходимость в дальнейшем анализе отпадает. Если же указатель сместился, это значит, что в буфер введен очередной код, который, очевидно, помещен в буфер по адресу, сохраненному в ячейке tail. Таким образом, мы сразу получаем адрес символа.

В свете изложенного становится ясно, что программа из предыдущей статьи была написана не совсем корректно. Мы не рассматривали возможности нажатия управляющих клавиш, которые ничего не заносят в буфер и не смещают хвостовой указатель. Практически, однако, нажатие управляющей клавиши приведет к тому, что программа проанализирует предыдущий символ в буфере, который, скорее всего, был уже проанализирован (и, возможно, заменен) ранее. Его повторный анализ ничего плохого не сделает.

Если сравнить листинги примеров 52.1 и 53.1, то окажется, что первый вариант на 14 байт короче второго. Однако мы не ставили целью создание наиболее эффективной программы, а лишь демонстрировали методики написания обработчиков прерываний и работы с входным буфером клавиатуры.

В программе 53.1 (как и 52.1) для адресации входного буфера используется сегментный регистр ES. Это довольно естественно, так как регистр DS обычно служит для адресации полей данных программы. Однако в резидентной программе типа.COM сегмент данных отсутствует, и сегментный регистр DS свободен. В этом случае использование регистра DS для обращения по абсолютным адресам имеет то преимущество, что в программных строках отсутствует префикс замены регистра, и каждая команда с адресацией к памяти получается на один байт короче. Для всей программы 53.1 это даст экономию 7 байт. Ниже приведено несколько предложений программы 53.1 в новом варианте.

```
new_09h proc
        push   AX           ;Сохраним используемые
        push   DS           ;регистры
        mov    AX,40h       ;Настроим DS
        mov    DS,AX        ;на область данных BIOS
        mov    AX,DS:[1Ch];По умолчанию используется DS
        ...
        cmp    BX,DS:[1Ch];По умолчанию используется DS
        je     outout
        mov    AX,[BX]      ;По умолчанию используется DS
```

Обратите внимание, что в тех командах, где выполняется обращение по абсолютному адресу, необходимо указывать обозначение сегментного регистра. Так, команда

```
        mov    AX,[1Ch]
```

засылает в AX число 1Ch (квадратные скобки не определяют косвенную адресацию), в то время, как команда

```
mov   AX,DS:[1Ch]
```

загружает в AX содержимое слова памяти, расположенного по адресу DS:1Ch. Однако указание регистра DS в программной строке не приводит к добавлению к коду команды префикса замены сегментного регистра, так как все команды с обращением к памяти в простейшем варианте (без префикса) используют для адресации именно регистр DS.

Статья 54

Динамический дамп

При изучении системных средств и отладке сложных программ часто возникает необходимость получить "мгновенный снимок" состояния программы в той или иной точке. Изучение содержимого регистров позволяет установить, в какой области оперативной памяти находится и к какой области обращается программа, какой стек она использует, какие флаги установлены и т.д. Далеко не всегда эту информацию можно получить с помощью отладчика, хотя бы потому, что программа, выполняемая под управлением отладчика, загружается не по тем адресам, где она будет работать в "самостоятельном" режиме. Кроме того, отладчик перехватывает некоторые прерывания и тем самым искажает ту операционную среду, в которой работает программа. Практически невозможно с помощью отладчика отлаживать обработчики аппаратных прерываний.

Конечно, в программу всегда можно встроить фрагменты вывода на экран требуемой информации. Однако, как мы уже видели, программа вывода на экран содержимого регистров или ячеек памяти в шестнадцатеричной форме довольно сложна, и включение такого длинного фрагмента в отлаживаемую программу, да еще в нескольких точках, не всегда возможно. Проблему можно решить с помощью резидентной программы динамического дампа, загружаемой в память независимо от отлаживаемой программы на весь сеанс работы с компьютером. Активизация этой программы может выполняться из отлаживаемой программы с помощью команды int, которая занимает в памяти всего лишь 2 или даже 1 байт, и которую можно без труда поместить в любое место отлаживаемой программы. При этом программу динамического дампа мож-

но с помощью команды DOS LOADHIGH загрузить не в обычную, а в расширенную память, что позволит отлаживать программу в точности в той же области памяти, где она будет работать.

Для того, чтобы резидентная программа активизировалась извне командой int, надо просто поместить в выбранный вектор с номером n адрес ее рабочей части, которая должна завершаться командой iret. Тогда команда int n в отлаживаемой программе будет передавать управление резидентной, а команда iret, завершающая рабочую часть программы дампа, будет возвращать управление в отлаживаемую программу. Естественно, выбранный вектор не должен использоваться системой и загруженными или запускаемыми прикладными программами.

Если с помощью какой-либо инструментальной программы (например, Manifest фирмы Quarterdeck) просмотреть таблицу векторов, то можно заметить, что значительная их часть не используется. Сюда относятся, например, векторы 60h...66h и F1h...FFh, которые специально отведены для использования в прикладных программах, а также векторы 32h, 34h...3Fh, 42h, 44h, 45h, 48...49h и ряд других, зарезервированных для дальнейшего использования системой. Любым из этих векторов в принципе можно воспользоваться "в личных целях". При этом, однако, следует иметь в виду, что коммерческие программы, устанавливаемые на компьютере, часто используют свободные векторы. Например, резидентная антивирусная программа PCCSTSR.COM фирмы Trend Micro Devices использует вектор пользователя 60h. Поэтому, создавая свою резидентную программу, активизируемую через вектор прерывания, надо предварительно убедиться, что выбранный вектор действительно свободен.

Команда int n преобразуется в результате трансляции в код CD n и занимает два байта. Специально для отладочных целей в системе команд микропроцессора предусмотрена команда int 3, которая занимает не два, а всего один байт. В ее машинный код (CCh) не входит номер вектора, однако она всегда вызывает программу, адрес которой находится в векторе с номером 3. Таким образом, для программы динамического дампа лучше всего использовать именно этот вектор, который, кстати, с меньшей вероятностью будет занят какой-то другой программой (с другой стороны, отладчики перехватывают этот вектор, поэтому отлаживать нашу программу динамического дампа с помощью отладчика может оказаться затруднительным).

Ранее мы обсуждали программы, позволяющие вывести на экран содержимое регистров или ячеек памяти. Эти программы были оформлены нами в виде подпрограмм и помещены в объектную библиотеку. Однако воспользоваться ими в таком качестве не очень просто. Написанные нами программы использовали (возможно, и не очень обоснованно) не только сегмент команд, но и сегмент данных. Следовательно,

нам придется создавать резидентную программу в формате .EXE, что сопряжено с некоторыми сложностями. Далее, при компоновке загрузочного модуля компоновщик поместит библиотечные подпрограммы в конец сегмента команд, и при оставлении в памяти резидентной программы (после ее установки) придется сохранить в памяти и ненужную уже секцию инициализации, которая теперь окажется в середине модуля. Конечно, все эти проблемы можно так или иначе решить, однако все же предпочтительнее создать резидентную программу в формате .COM и включить разработанные нами ранее подпрограммы прямо в ее текст.

Приведенная ниже резидентная программа динамического дампа (пример 54.1) активизируется через вектор 3 и выводит на экран текущее содержимое всех регистров процессора, включая указатель команд IP и регистр флагов. После загрузки в память и инициализации программа дампа вызывается однобайтовой командой int, включаемой в требуемое место отлаживаемой программы. Поскольку вывод на экран содержимого регистров осуществляется с помощью функции DOS, наша программа дампа разрушит систему, если ее вызвать из обработчика аппаратного прерывания, который прервал выполнение каких-либо функций DOS. Однако написание обработчиков аппаратных прерываний, которые могут без последствий прерывать выполнение функций DOS и BIOS, вообще представляет собой довольно тонкую материю, отдельные элементы которой будут рассмотрены в дальнейшем.

Пример 54.1. Резидентная программа динамического дампа регистров

```
text    segment 'code'
        assume CS:text, DS:text
        org    256
main    proc
        jmp    init         ;Переход на инициализацию
binasc  proc
;Подпрограмма преобразования двоичного слова в 16-ричную
;символьную форму и вывода его на экран. При вызове:
;AX=преобразуемое число, DS:SI=адрес строки с результатом
        push   CX           ;Сохраним используемый регистр
        push   AX           ;Сохраним наше число в стеке
        and    AX,0F000h    ;Выделим старшую четверку битов
        mov    CL,12        ;Счетчик сдвига
        shr    AX,CL        ;Сдвиг вправо на 12 бит
        call   hexasc       ;Преобразуем в символ
        pop    AX           ;Вернем в AX исходное число
        push   AX           ;И отправим его обратно в стек
        and    AX,0F00h     ;Выделим вторую четверку битов
        mov    CL,8         ;Счетчик сдвига
        shr    AX,CL        ;Сдвиг вправо на 8 бит
        inc    SI           ;Сдвинемся к следующему байту
        call   hexasc       ;Преобразуем в символ
        pop    AX           ;Вернем в AX исходное число
        push   AX           ;И снова сохраним в стеке
```

```
        and     AX,0F0h     ;Выделим третью четверку битов
        mov     CL,4        ;Счетчик сдвига
        shr     AX,CL       ;Сдвиг вправо на 4 бита
        inc     SI          ;Сдвинемся к следующему байту
        call    hexasc      ;Преобразуем в символ
        pop     AX          ;Вернем в AX исходное число
        push    AX          ;И снова сохраним в стеке
        and     AX,0Fh      ;Выделим младшую четверку битов
        inc     SI          ;Сдвинемся к следующему байту
        call    hexasc      ;Преобразуем в символ
        pop     AX          ;Восстановим стек
        pop     CX          ;Восстановим регистр
        ret
binasc  endp
hexasc  proc
;Подпрограмма получения символьного представления четырехбитового
;числа. На входе: число в младшей половине AL; адрес строки, куда
;надо поместить результат, в DS:SI
        push    BX          ;Сохраним используемый регистр
        mov     BX,offset tblhex;BX=адрес таблицы трансляции
        xlat                ;Команда табличной трансляции
        mov     [SI],AL     ;Сохраним полученный символ
        pop     BX          ;Восстановим регистр
        ret
hexasc  endp
new_03h proc
;Сохраним в стеке все регистры для последующего вывода их
;содержимого на экран и восстановления перед завершением
        push    SP
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    BP
        push    DS
        push    ES
        push    SS
        mov     BP,SP       ;Настроим базовый регистр BP на
                            ;текущую вершину стека
;Настроим DS на наш сегмент команд для простоты программирования и
;для команды xlat
        push    CS
        pop     DS
;Будем извлекать из стека значения регистров и заполнять ими поле
;вывода
        mov     AX,[BP]     ;SS
        mov     SI,offset regs1+43
        call    binasc
        mov     AX,[BP]+2   ;ES
        mov     SI,offset regs1+35
        call    binasc
        mov     AX,[BP]+4   ;DS
        mov     SI,offset regs1+27
        call    binasc
        mov     AX,[BP]+6   ;BP
```

```
        mov   SI,offset regs+51
        call  binasc
        mov   AX,[BP]+8  ;DI
        mov   SI,offset regs+43
        call  binasc
        mov   AX,[BP]+10 ;SI
        mov   SI,offset regs+35
        call  binasc
        mov   AX,[BP]+12 ;DX
        mov   SI,offset regs+27
        call  binasc
        mov   AX,[BP]+14 ;CX
        mov   SI,offset regs+19
        call  binasc
        mov   AX,[BP]+16;BX
        mov   SI,offset regs+11
        call  binasc
        mov   AX,[BP]+18 ;AX
        mov   SI,offset regs+3
        call  binasc
        mov   AX,[BP]+20 ;SP
        add   AX,6       ;Скорректируем SP
        mov   SI,offset regs1+3
        call  binasc
        mov   AX,[BP]+22 ;IP вызвавшей задачи
        mov   SI,offset regs1+11
        call  binasc
        mov   AX,[BP]+24 ;CS вызвавшей задачи
        mov   SI,offset regs1+19
        call  binasc
        mov   AX,[BP]+26 ;Флаги
        mov   SI,offset regs1+54
        call  binasc
;Выведем на экран
        mov   AH,09h
        mov   DX,offset lfcr
        int   21h
;Восстановим все регистры
        pop   SS
        pop   ES
        pop   DS
        pop   BP
        pop   DI
        pop   SI
        pop   DX
        pop   CX
        pop   BX
        pop   AX
        pop   SP
        iret
tblhex  db    '0123456789ABCDEF'
lfcr    db    10,13
regs    db 'AX=**** BX=**** CX=**** DX=**** SI=**** DI=**** '
        db 'BP=****   ',10,13
regs1   db 'SP=**** IP=**** CS=**** DS=**** ES=**** SS=**** '
        db 'FLAGS=****'
lfcr2   db    10,13,'$'
```

```
new_03h endp
main    endp
end_res=$
init    proc
;Процедура инициализации
;Заполним вектор 3
        mov     AX,2503h
        mov     DX,offset new_03h
        int     21h
;Выведем на экран информационное сообщение
        mov     AH,09h
        mov     DX,offset mes
        int     21h
;Завершим программу, оставив ее резидентной в памяти
        mov     AX,3100h
        mov     DX,(end_res-main+10fh)/16
        int     21h
mes     db      'Программа динамического дампа загружена',10,13,'$'
init    endp
text    ends
        end     main
```

Для преобразования 16-битовых двоичных чисел в 16-ричную символьную форму используются описанные ранее подпрограммы binasc и hexasc (см. статью 31). Программа соответствует минимальной (односегментной) модели памяти, поэтому все ее поля данных располагаются в сегменте команд. Для того, чтобы оставить возможность обращаться к полям данных через сегментный регистр DS, что, кстати, однозначно требуется для правильного выполнения команды xlat, в резидентной процедуре new_03h после сохранения в стеке регистров вызывающей программы содержимое CS заносится через стек в регистр DS.

В процедуре new_03h использованы стандартные для программ такого рода приемы передачи параметров. Все программно-адресуемые регистры вызывающей программы, за исключением, разумеется, CS, сохраняются в стеке. Это, во-первых, дает возможность восстановить их перед выходом из резидентной программы, а во-вторых, позволяет воспользоваться стеком для передачи необходимых параметров, каковыми эти регистры и являются.

Поле данных, куда помещаются результаты преобразования, для удобства программирования разбито на несколько подполей. Выводимая строка начинается с обозначения lfcr и включает коды возврата каретки и перевода строки, шаблоны для содержимого регистров в 16-ричной форме и завершающие коды возврата каретки и перевода строки. Эта позволяет вызывать наш отладчик многократно с достаточно аккуратным размещением на экране его вывода. Заканчивается поле результатов символом "$" для функции DOS 09h.

После сохранения всех регистров содержимое указателя стека SP копируется в базовый регистр BP, архитектурно предназначенный для

работы со стеком. На рис. 54.1 приведено состояние стека на этот момент.

Рис. 54.1. Содержимое стека в отладчике.

Перед каждым вызовом подпрограммы binasc в регистр AX заносится один из параметров из стека, а в регистр SI - смещение к его расположению в выходной строке. Для выборки из стека параметров с сохранением содержимого стека используется базовая адресация со смещением через регистр BP. В этом случае по умолчанию адресуется стек, т.е. команда

```
mov     AX, [BP]
```

выполняется фактически, как команда

```
mov     AX, SS:[BP]
```

Сопоставляя текст программы с рисунком, можно убедиться, что каждый параметр после преобразования его в символьную форму попадает на предназначенное для него место в выходной строке.

Значение SP, сохраненное в стеке по адресу BP+20, относится к состоянию стека на момент входа в отладчик. Поскольку при отладке программы нас, скорее всего, интересует состояние указателя стека на момент вызова прерывания 3, значение SP, полученное из стека, корректируется добавлением 6 (три двухбайтовых слова для CS, IP и флагов). Далее из стека извлекаются составляющие вектора вызывающей программы. Заметим, что получение значения IP из стека после команд

прерывания или вызова подпрограммы - едва ли не единственная возможность прочитать указатель команд IP, который, как известно, не является адресуемым регистром.

После преобразования в символьную форму всех параметров образованная строка выводится на экран функцией DOS 09h, восстанавливаются все регистры и командой iret управление возвращается в вызвавшую программу.

Процедура инициализации init построена традиционно. В вектор 3 заносится адрес нашего отладчика, на экран выводится информационное сообщение и программа завершается функцией DOS 31h.

Программу отладчика следует оттранслировать, скомпоновать и перевести в формат .COM утилитой EXE2BIN. После этого ее можно загрузить в обычную память или в блоки UMB, если читатель работает на компьютере с расширенной памятью и процессором 386 или выше.

Для проверки работоспособности отладчика достаточно запустить программу, приведенную в примере 54.2.

Пример 54.2. Тестовая программа для проверки работоспособности резидентной программы динамического дампа

```
text    segment 'code'
        assume CS:text
main:   mov    AX,1111h   ;Произвольные
        mov    BX,2222h   ;проверочные
        mov    CX,3333h   ;числа
        mov    DX,4444h
        mov    SI,5555h
        mov    DI,offset endpr
        mov    BP,SP
        int    3
endpr:  mov    AX,4C00h
        int    21h
text    ends
        end    main
```

В выводе на экран программы динамического дампа содержимое регистров AX, BX, CX, DX и SI должно совпадать с числами, загруженными в них в тестовой программе (1111h, 2222h и т.д.); содержимое регистра DI (смещение точки endpr) должно совпадать с содержимым указателя команд IP в точке вызова прерывания int; должно также совпадать содержимое регистров BP и SP.

Статья 55

Переключение стека в обработчике прерываний

Описанная в предыдущей статье программа динамического дампа является весьма полезным инструментальным средством, которое может найти применение как при исследовании функционирования операционной системы, так и при отладке конкретных программ. Созданная нами программа выводит содержимое всех регистров процессора; читатель без особого труда сможет модифицировать программу, чтобы она выводила иную информацию, например, содержимое определенных полей данных, стека или сегмента команд. Правда, в приведенном простейшем варианте у программы больше недостатков, чем достоинств. Ее нельзя выгрузить из памяти; даже в процессе отладки самой программы вам, возможно, придется многократно перезагружать компьютер, чтобы изгнать из памяти старый неправильный вариант программы. Как уже отмечалось, программу нельзя вызывать из обработчиков аппаратных прерываний, поскольку она использует для вывода на экран функцию DOS, а вызывать функции DOS в обработчиках аппаратных прерываний допустимо только при условии выполнения определенных процедур проверки текущего состояния DOS. Наконец, в программе отсутствует собственный стек, хотя по ходу своей работы она активно использует стек для сохранения регистров и при переходе на подпрограммы. В последующих статьях будут постепенно рассматриваться и устраняться принципиальные недостатки этой программы. Начнем с вопроса о переключении стека.

При реализации процедуры прерывания (аппаратного или программного) процессор сохраняет в стеке прерванной задачи флаги, CS и IP и загружает из вектора прерывания в регистры CS и IP двухсловный адрес обработчика прерывания. Все остальные регистры хранят ту информацию, которая в них была на момент прерывания. Для того, чтобы не разрушить прерванную задачу, в самом начале программы обработчика следует сохранить все используемые в нем регистры, а перед завершающей командой iret восстановить их. Сказанное относится ко всем регистрам, в том числе к сегментным. Не следует забывать, что все команды обращения к памяти используют по умолчанию в качестве сегментного регистр DS. Если при переходе в обработчик не выполнить настройку DS на сегмент обработчика, то команды вида

```
        mov     AX,mem
        mov     BX,[SI]
```

будут обращаться к случайным ячейкам прерванной задачи, что, возможно, приведет к разрушению и этой задачи, и обработчика. Если по каким-то причинам нежелательно перенастраивать регистр DS, то в командах такого рода следует использовать замену сегмента:

```
        mov     AX,CS:mem
        mov     BX,CS:[SI]
```

Особая ситуация возникает со стеком. При переходе в обработчик регистры SS и SP настроены на стек прерванной задачи. Если этот стек имеет достаточный объем, то не будет большой беды в том, что он будет использоваться обработчиком. Если, однако, для работы обработчика требуется стек большого объема, или стек прерванной задачи назначен без запаса, то легко может произойти переполнение стека с фатальными для прерванной задачи последствиями. Поэтому безопаснее в обработчике прерывания иметь собственный стек.

Смена стека требует запоминания кадра стека (содержимого SS и SP) в ячейках памяти, отведенной обработчику, занесения в SS и SP новых значений и восстановления старого кадра перед выходом из обработчика. Естественно, в области данных обработчика должен быть выделен достаточный объем памяти для использования к качестве стека.

Выполним операцию замены стека в резидентной программе динамического дампа, описанной в предыдущей статье. Применительно к этой программе смена стека имеет свои особенности. Поскольку программа дампа предназначена для изучения состояния прерываемой задачи, желательно, чтобы она как можно меньше влияла на это состояние и, в частности, по возможности не нарушала содержимое стека задачи. Приводимый ниже вариант программы учитывает это пожелание.

Пример 55.1. Переключение стека

```
text    segment 'code' public byte
        assume CS:text, DS:text
        org     256
main    proc
        jmp     init
ss_seg  dw      0           ;Ячейка для содержимого SS
sp_offs dw      0           ;Ячейка для содержимого SP
mem     dw      0           ;Ячейка для временного хранения CS
binasc  proc
...
binasc  endp
hexasc  proc
...
hexasc  endp
new_03h proc
;Сохраним в стеке все регистры для последующего вывода их
```

```
;содержимого на экран
        mov     CS:ss_seg,SS;Сохраним кадр стека
        mov     CS:sp_offs,SP;вызывающей задачи
        cli                 ;Запрет аппаратных прерываний
        mov     CS:cs_seg,CS;Настроим SS на наш
        mov     SS,CS:cs_seg;сегмент (единственный)
        mov     SP,offset end_res;Настроим SP
        sti                 ;Разрешение аппаратных прерываний
;Теперь мы работаем на стеке резидентной программы
        push    AX          ;Сохраним регистры
        push    BX          ;вызывающей программы
        push    CX          ;с целью их последующей
        push    DX          ;распечатки на экране
        push    SI          ;Не надо сохранять
        push    DI          ;SS и SP, содержимое которых
        push    BP          ;уже находится в ячейках
        push    DS          ;ss_seg и sp_offs
        push    ES          ;
        mov     BP,SP       ;Настроим базовый регистр на текущую
                            ;вершину стека
;Настроим DS на наш сегмент для простоты программирования и для
;команды XLAT
        push    CS
        pop     DS
;Будем извлекать из стека значения регистров и заполнять ими поле
;вывода. Содержимое SS и SP извлечем из ячеек памяти
        mov     AX,CS:ss_seg;SS
        mov     SI,offset regs1+43
        call    binasc
        mov     AX,[BP]+0  ;ES
        mov     SI,offset regs1+35
        call    binasc
        mov     AX,[BP]+2  ;DS
        mov     SI,offset regs1+27
        call    binasc
        mov     AX,[BP]+4  ;BP
        mov     SI,offset regs+51
        call    binasc
        mov     AX,[BP]+6  ;DI
        mov     SI,offset regs+43
        call    binasc
        mov     AX,[BP]+8  ;SI
        mov     SI,offset regs+35
        call    binasc
        mov     AX,[BP]+10 ;DX
        mov     SI,offset regs+27
        call    binasc
        mov     AX,[BP]+12 ;CX
        mov     SI,offset regs+19
        call    binasc
        mov     AX,[BP]+14;BX
        mov     SI,offset regs+11
        call    binasc
        mov     AX,[BP]+16 ;AX
        mov     SI,offset regs+3
        call    binasc
        mov     AX,CS:sp_offs;SP
```

```
        add     AX,6        ;Скорректируем SP
        mov     SI,offset regs1+3
        call    binasc
;Настроим ES:DI на кадр стека вызвавшей задачи, чтобы получить
;сохраненные там процессором при выполнении команды int значения
;CS, IP и флагов
        mov     AX,CS:ss_seg;SS вызвавшей задачи
        mov     ES,AX       ;Отправим его в ES
        mov     DI,CS:sp_offs;SP вызвавшей задачи
        mov     AX,ES:[DI] ;IP вызвавшей задачи
        mov     SI,offset regs1+11
        call    binasc
        mov     AX,ES:[DI]+2;CS вызвавшей задачи
        mov     SI,offset regs1+19
        call    binasc
        mov     AX,ES:[DI]+4;Флаги
        mov     SI,offset regs1+54
        call    binasc
;Выведем на экран
        mov     AH,09h
        mov     DX,offset lfcr
        int     21h
;Восстановим все регистры
        pop     ES
        pop     DS
        pop     BP
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        cli
        mov     SP,CS:sp_offs  ;Восстановим кадр стека
        mov     SS,CS:ss_seg   ;вызвавшей задачи
        iret
tblhex  db      '0123456789ABCDEF'
lfcr    db      10,13
regs    db 'AX=**** BX=**** CX=**** DX=**** SI=**** DI=**** '
        db 'BP=****   ',10,13
regs1   db 'SP=**** IP=**** CS=**** DS=**** ES=**** SS=**** '
        db 'FLAGS=****'
lfcr2   db      10,13,'$'
stack_area      db          60 dup (?)
new_03h endp
main    endp
end_res=$
;Процедура инициализации
init    proc
...
init    endp
text    ends    ;Конец сегмента команд
        end     main
```

Тексты процедур binasc, hexasc и init остались без изменений и в примере 55.1 опущены. Процедура обработчика прерывания 03h начи-

нается теперь с сохранения кадра стека вызвавшей задачи в предусмотренных для этого ячейках в резидентной части обработчика. Адресация этих ячеек выполняется через сегментный регистр CS, поскольку регистр DS пока еще указывает на сегмент данных вызвавшей задачи.

Установка нового кадра стека всегда выполняется при запрещенных прерываниях, так как если аппаратное прерывание от какого-либо источника придет между командами заполнения SS и SP, вектор прерванного процесса будет сохранен в случайной ячейке памяти. Поскольку наш обработчик представляет собой программу типа .COM, состоящую из одного сегмента, в регистр SS надо занести сегментный адрес этого сегмента, который сейчас находится в регистре CS. В обычном случае мы выполнили бы это с помощью пары команд

```
push   cs
pop    ss
```

Эти команды, однако, будут выполняться на стеке прерванной задачи, который мы договорились по возможности не затрагивать. В то же время команда

```
mov    ss,cs
```

запрещена. Поэтому содержимое CS копируется в ячейку памяти mem, откуда уже переносится в SS.

Командой

```
mov    sp,offset end_res
```

в регистр SP заносится адрес дна нового стека, место под который выделено в самом конце резидентной части обработчика. Под стек отведено 60 байт, т.е. приблизительно столько, сколько будет использоваться программой дампа (с учетом вызываемой в нем функции DOS).

После переключения стека программа выполняется в принципе так же, как и в предыдущем варианте. Сохраняются регистры прерванной задачи (кроме SS и SP, которые уже сохранены в памяти), настраивается регистр DS и, наконец, содержимое регистров последовательно извлекается из стека или памяти, преобразуется в символьную форму и заносится в поле для вывода на экран. Сохранение SS и SP в ячейках памяти привело к некоторому порядка данных, сохраняемых в стеке, что нашло отражение в изменении смещений в командах извлечения данных из стека.

Вывод полученной символьной строки на экран осуществляется попрежнему функцией DOS 09h. Перед выходом из прерывания восстанавливается (естественно, при запрещенных аппаратных прерываниях) кадр стека прерванной задачи. Команду sti в данном случае выполнять нет необходимости, так как команда iret, восстановив слово флагов, установит и флаг управления прерываниями IF.

Для того, чтобы испытать новый вариант программы дампа, можно воспользоваться тестовой программой, приведенной в примере 52.2.

Статья 56

Функция DOS или прерывание BIOS?

Следующим серьезным недостатком программы динамического дампа является использование в ней для вывода на экран функции DOS. Для того, чтобы убедиться в неприемлемости такого решения, поставьте эксперимент, в котором программа дампа будет вызываться по какому-либо аппаратному прерыванию. Нагляднее всего использовать для этого прерывание от клавиатуры. В примере 56.1 приведена простая резидентная программа, перехватывающая прерывание 09h от клавиатуры и анализирующая скен-код нажатой клавиши. Все коды, кроме кода 4Eh ("серый плюс") передаются в системный обработчик прерываний от клавиатуры. Нажатие клавиши "серый плюс" приводит к выдаче сигнала конца прерывания EOI и вызове прерывания 03h, активизирующего нашу программу дампа.

Пример 56.1. Резидентная программа для активизации программы динамического дампа нажатием клавиши

```
text    segment 'code' public
        assume cs:text,ds:text
        org    256
main    proc
        jmp    init
;Поля данных резидентной секции
old_09h dd     0           ;Ячейка для сохранения системного
                           ;вектора 09h
;Наш обработчик от клавиатуры
new_09h proc
        push   AX          ;Сохраним используемый регистр
        in     AL,60h      ;Введем скен-код
        cmp    AL,4Eh      ;Скен-код серого плюса?
        je     hotkey      ;Да, на активизацию дампа
        pop    AX          ;Нет, восстановим регистр
        jmp    CS:old_09h  ;и в системный обработчик
hotkey:
;Разрешим дальнейшую работу клавиатуры
        in     AL,61h      ;Введем содержимое порта B
        or     AL,80h      ;Подтвердим прием кода, добавив
```

```
        out     61h,AL      ;бит 80h к содержимому порта B
        and     AL,7Fh      ;Снова разрешим работу клавиатуры,
        out     61h,AL      ;сбросив в порте B бит 80h
;Пошлем в контроллер прерываний команду EOI
        mov     AL,20h      ;Дадим в контроллер прерываний
        out     20h,AL      ;команду EOI
        pop     AX          ;Восстановим регистр
        int     3
        iret                ;Выход из прерывания
new_09h endp
end_res=$                   ;Смещение конца резидентной части
                            ;программы
main    endp
;Процедура инициализации
init    proc
;Сохраним вектор 09h
        mov     AX,3509h
        int     21h
        mov     word ptr CS:old_09h,BX
        mov     word ptr CS:old_09h+2,ES
;Заполним вектор 09h
        mov     AX,2509h
        mov     DX,offset new_09h
        int     .21h
;Выведем на экран информационное сообщение
        mov     AH,09h
        mov     DX,offset mes
        int     21h
;Завершим программу, оставив ее резидентной в памяти
        mov     AX,3100h
        mov     DX,(end_res-main+10Fh)/16
        int     21h
init    endp
mes     db      'Резидентный обработчик прерывания 09h загружен$'
text    ends
        end     main
```

Оттранслировав, скомпоновав и преобразовав в формат .COM эту программу, выгрузите программу Norton Commander или другую оболочку DOS, в которой вы работаете. Загрузите в память (в любом порядке) обе программы - динамического дампа и перехвата прерываний от клавиатуры. Убедитесь, что компьютер работает обычным образом - выполняет команды DOS, выводит на экран требуемую информацию и т.д. Нажмите клавишу "серый плюс". Программа динамического дампа выведет на экран содержимое регистров, после чего система неминуемо зависнет. Почему так происходит?

Дело в том, что если компьютер "ничего не делает", в действительности выполняется программа командного процессора, который ожидает ввода с клавиатуры команд оператора. Ожидание заключается в том, что процессор выполняет в цикле некоторый фрагмент DOS, реализующий функцию DOS 0Ah ввода строки с клавиатуры. Эта часть программы опрашивает (с помощью драйвера CON) буфер клавиатуры и, не

найдя в нем символов, бесконечно повторяет цикл опроса. Таким образом, на бездействующем компьютере непрерывно выполняется функция DOS. В какой бы момент времени мы не нажали клавишу, этим действием прервется выполнение функции DOS. Само по себе это не страшно. Если программа обработки прерывания от клавиатуры написана корректно - в ней сохраняются и восстанавливаются все используемые регистры и не нарушается (к моменту завершения) состояние стека, то такая приостановка выполнения функции DOS не приведет ни к каким особым последствиям. Если, однако, в программе обработки прерывания вызвать какую-либо функцию DOS, то сама она выполнится правильно, но при возврате управления командой iret в прерванную функцию DOS та разрушит систему, что мы и наблюдали в нашем эксперименте. Невозможность вызвать функцию DOS "изнутри" другой функции DOS носит название нереентерабельности DOS.

Разрушение системы связано с тем, что DOS выполняет свои функции на собственных стеках. Всего в DOS имеется три стека - стек ввода-вывода, дисковый стек и вспомогательный стек. Диспетчер DOS, получив управление по команде int 21h и сохранив на стеке вызывающей задачи ее регистры (на что требуется 9 слов), выполняет переключение на один из стеков DOS в зависимости от номера вызванной функции. Все функции DOS делятся в этом отношении на две группы. В первую группу входят функции ввода с клавиатуры и вывода на экран с номерами от 01h до 0Ch. Они выполняются на стеке ввода-вывода. Вторую группу образуют практически все остальные функции DOS (файловые, службы времени, обслуживания памяти и процессов и проч.) с номерами 00h и от 0Dh до 6Ch. Все эти функции выполняются на дисковом стеке. Наконец, имеется ограниченное количество функций, не использующих стеки DOS - они работают на стеке пользователя.

Перед началом выполнения функции DOS диспетчер DOS заносит в указатель стека SP адрес дна соответствующего стека DOS. Функция DOS, выполняясь, активно использует этот стек для вызова своих внутренних подпрограмм и передачи в них параметров. Если прервать этот процесс и активизировать какую-либо функцию DOS той же группы, то диспетчер DOS снова установит регистр SP на начало того же стека, и вторая функция DOS, выполняясь, начнет затирать содержимое стека, оставшееся от первой функции. Ясно, что при возврате в первую функцию та выполняться не сможет. Если, однако, прервана функция одной группы, а в обработчике прерывания вызывается функция другой группы, то поскольку они работают на разных стеках, затирания стека не произойдет и система будет работать нормально.

В нашем экспериментальном программном комплексе функция DOS 0Ah, выполняемая в COMMAND.COM, прерывается функцией 09h программы динамического дампа, принадлежащей к той же группе

функций ввода-вывода. Этим и объясняется зависание системы. Выйти из положения в принципе можно, использовав в программе дампа вместо функции 09h функцию вывода 40h, входящую в группу "дисковых" функций. Как мы уже видели, эта функция, выводя информацию через дескриптор стандартного вывода (01h), посылает ее на экран. Однако, работая с дескриптором 01h, эта функция уподобляется функциям ввода-вывода (выполняющим вывод через тот же дескриптор) и ее использование в обработчике прерываний по-прежнему приведет к зависанию системы. Выход заключается в том, чтобы открыть экран, как файл, получив для него другой дескриптор. Соответствующий фрагмент программы приведен в примере 56.2. Замените им в программе динамического дампа строки вывода с помощью функции 09h и повторите эксперимент с обработчиком прерываний от клавиатуры. Теперь все будет работать правильно.

Пример 56.2. Вывод на экран функцией 40h через нестандартный дескриптор

```
        mov     AH,3Dh      ;Функция открытия файла
        mov     AL,1        ;Доступ для записи
        mov     DX,offset consol;Адрес имени "файла"
        int     21h         ;Вызов DOS
        mov     BX,AX       ;Отправим дескриптор в BX
        mov     AH,40h      ;Файловая функция вывода
        mov     CX,meslen   ;Длина выводимой строки
        mov     DX,offset lfcr;Адрес выводимой строки
        int     21h         ;Вызов DOS
;Поля данных
consol  db      'CON',0
tblhex  db      '0123456789ABCDEF'
lfcr    db      10,13
regs    db 'AX=**** BX=**** CX=**** DX=**** SI=**** DI=**** '
        db 'BP=****   ',10,13
regs1   db 'SP=**** IP=**** CS=**** DS=**** ES=**** SS=**** '
        db 'FLAGS=****'
lfcr2   db      10,13
meslen=$-lfcr
stack_area      db 128 dup ('q')
```

Функция 3Dh служит для открытия уже имеющегося файла. В качестве файла может выступать и любое стандартное устройство компьютера - принтер, клавиатура, экран, последовательный порт. Указав в качестве имени файла обозначение CON (см. поля данных), мы открываем консоль (экран). Выделенный для консоли дескриптор DOS возвращает в регистре AX. При вызове функции 3Dh в регистре AL следует указать режим доступа к открываемому файлу. Код 0 обозначает доступ только для чтения, код 1 - только для записи, код 2 - для чтения и записи.

Для вывода информации на экран используется функция 40h, которая требует указания в регистре BX дескриптора файла или устройства,

в регистре CX - числа выводимых байтов, а в регистре DX - адреса буфера с выводимой информацией. Длина строки meslen определяется путем вычитания из значения счетчика текущего адреса $ в конце строки адреса ее начала lfcr. Символ '$', которым мы завершали строку для функции 09h, здесь удален.

Хотя мы и избавились от зависания системы, однако ценность новой программы динамического дампа по-прежнему невелика. Теперь ее можно включать в состав обработчиков аппаратных прерываний с целью их отладки, однако при этом основная (прерываемая) программа имеет право выполнять лишь функции DOS из диапазона 01h...0Ch. Если же в основной программе встретится какая-либо "дисковая" функция (из диапазона 0Dh...6Ch), прерывание ее функцией 3Dh программы дампа немедленно приведет к разрушению системы.

Радикальным способом преодоления нереентерабельности DOS в нашем случае является полный отказ от использования функций DOS в программе динамического дампа. Если для вывода информации на экран вместо тех или иных функций DOS использовать прерывание BIOS 10h, то никаких проблем со стеками DOS не возникнет, так как программы BIOS не используют системные стеки, а работают на стеке программы, вызвавшей прерывание 10h. В примере 56.3 для вывода на экран использована функция Eh "записи символа в режиме телетайпа". Прерывание 10h включает целый ряд функций вывода символов на экран, отличающихся своими возможностями. Например, функция 09h записывает в позицию курсора символ и его атрибут, что позволяет выводить на экран цветные символы, но вывод очередного символа не сопровождается перемещением курсора. При использовании функции 09h перед выводом каждого очередного символа приходится "вручную" устанавливать новую позицию курсора специально предназначенной для этого функцией 02h. Функция 13h выводит сразу строку символов (в том числе цветных), перемещая курсор по мере вывода каждого символа, но требует указания исходной позиции строки на экране. Это в нашем случае не очень удобно, так как не позволит вызывать программу дампа повторно (она будет каждый раз выводить информацию на одно и то же место, затирая предыдущий вывод). Выбранная нами функция Eh не позволяет задавать цвета символов, но, во-первых, перемещает курсор после вывода очередного символа и, во-вторых, отрабатывает управляющие коды возврата каретки и перевода строки. Таким образом, при многократном вызове нашей программы дампа каждый очередной вывод будет располагаться на экране после предыдущего.

Пример 56.3. Вывод на экран средствами BIOS

```
mov     AH,0Eh
mov     BX,offset lfcr
mov     CX,meslen
```

```
outscr: mov   AL,[BX]
        int   10h
        inc   BX
        loop  outscr
```

Функция Eh выводит на экран один символ (из регистра AL), и для вывода строки функцию надо вызывать в цикле, "подкладывая" ей каждый раз очередной символ. В приведенном фрагменте в регистр BX сначала засылается адрес выводимой строки lfcr, а затем в каждом шаге цикла содержимое BX увеличивается на 1, выступая в качестве указателя на очередной выводимый символ.

При замене в программе функций DOS на функции BIOS необходимо увеличить размер стека, поскольку функции BIOS работают на стеке вызвавшей их программы. Функция Eh прерывания 10h использует в процессе своего выполнения около 30 слов стека, поэтому с учетом потребностей самой программы динамического дампа, размер стека надо сделать не менее 45...50 слов.

Замените в программе динамического дампа фрагмент вывода на экран строками, приведенными в примере 56.3. Загрузите в память обе резидентные программы - дампа и обработки прерываний от клавиатуры. Нажимая клавишу "серый плюс", убедитесь, что программа динамического дампа работает правильно. Запишите ее вывод (или, если у вас есть принтер, распечатайте содержимое экрана с помощью клавиши <Print Screen>).

Запустите программу Manifest (или команду DOS MEM). Внимательно рассмотрите содержимое памяти и сопоставьте его с выводом программы дампа. Обратите особое внимание на содержимое сегментных регистров, указателя команд и указателя стека. Не забудьте еще раз взглянуть на исходный текст (или, еще лучше, листинг трансляции) тестового обработчика прерываний от клавиатуры. Какие регистры он использует и что в них находится к моменту вызова прерывания 03h? Вся эта работа позволит вам освежить в памяти процессы, происходящие в системе при обработке аппаратных и программных прерываний, и, в частности, вспомнить, что происходит при этом с регистрами процессора (общего назначения, сегментными и прочими).

Статья 57

Защита резидентной программы от повторной установки

Все резидентные программы, которые мы рассматривали в предыдущих статьях, имели серьезный недостаток: они не были защищены от повторной загрузки. Вспомним, что для работы с резидентной программой ее следует прежде всего запустить с клавиатуры, как обычную программу. В этом случае управление передается (в соответствии с параметром директивы end) на секцию инициализации, в которой подготавливаются условия для дальнейшей активизации программы уже в резидентном состоянии. Как правило, в секции инициализации загружаются векторы прерываний, через которые будет активизироваться программа. Последними строками секции инициализации вызывается функция DOS 31h, которая выполняет завершение программы с оставлением в памяти ее резидентной части.

Если такую программу запустить с клавиатуры повторно, в память будет загружена и останется резидентной ее вторая копия. Это плохо не только потому, что понапрасну расходуется память; более неприятным является вторичный перехват тех же векторов. Если резидентная программа после ее активизации не обращается к старому содержимому перехваченных ею векторов, то вторая копия полностью лишит первую работоспособности, и тогда повторная загрузка приведет только к расходованию памяти. Если, однако, как это обычно и имеет место, резидентная программа в процессе своей работы передает управление старому обработчику перехваченного ею прерывания, то новая копия резидентной программы, сохранившая в процессе инициализации адрес первой копии в качестве содержимого перехватываемого вектора, будет при каждой активизации вызывать и первую копию. В результате резидентная программа будет фактически выполняться при каждом вызове дважды. Во многих случаях такое повторное выполнение нарушит правильную работу программы. Поэтому обязательным элементом любой резидентной программы является процедура защиты ее от повторной загрузки, или, как говорят, установки.

Наиболее распространенным методом защиты резидентной программы от повторной установки является использование прерывания 2Fh, специально предназначенного для связи с резидентными програм-

мами. При вызове этого прерывания в регистре AH задается номер функции (от 00h до FFh), а в регистре AL - номер подфункции (в том же диапазоне). Функции с 00h по BFh зарезервированы для использования системой (например, функции 00h и 01h закреплены за резидентной программой DOS PRINT.COM, а 10h - за программой SHARE.EXE), а функции с C0h по FFh могут использоваться прикладными программами.

Для того, чтобы резидентная программа могла отозваться на вызов прерывания int 2Fh, в ней должен иметься обработчик этого прерывания. Фактически все резидентные программы, как системные, так и прикладные, имеют такие обработчики, через которые осуществляется не только проверка на повторную установку, но и вообще связь с резидентной программой: смена режима ее работы или получение от нее в транзитную программу каких-то параметров. Задание действия, которое надлежит выполнить обработчику прерывания 2Fh конкретной резидентной программы, осуществляется с помощью номера подфункции, помещаемого перед вызовом прерывания в регистр AL.

Таким образом, обработчик прерывания 2Fh резидентной программы должен прежде всего проверить номер функции в регистре AH; при обнаружении "своей" функции обработчик анализирует содержимое регистра AL и выполняет затребованные действия, после чего командой iret передает управление вызвавшей его программе. Если, однако, обработчик обнаружил в регистре AH "чужую" функцию, он должен командой jmp CS:old_2fh передать управление по цепочке тому обработчику, адрес которого был ранее в векторе 2Fh. В результате вызов int 2Fh из любой программы будет проходить по цепочке через все загруженные резидентные программы, пока не достигнет "своей" программы или не вернет управление в вызвавшую программу через обработчик DOS (который, очевидно, всегда будет самым последним в цепочке).

Естественно, для коммуникации с резидентной программой должен быть установлен некоторый интерфейс. Обычно при проверке на повторную установку резидентная программа, если она уже находится в памяти, возвращает в регистре AL значение FFh, которое является признаком запрета вторичной загрузки. Иногда для большей надежности идентификации "своей" функции резидентная программа, помимо значения FFh в регистре AL, возвращает еще какие-то обусловленные заранее коды в других регистрах. Часто через дополнительные регистры передается символьная информация, например, имя программы. В этом случае, если вызвавшая программа с именем DUMP.COM (т.е. вторая копия резидентной программы, выясняющая, можно ли ей остаться резидентной в памяти) получает после вызова int 2Fh в регистре AL значение FFh, а в регистрах CX и DX символьные коды 'DU' и 'MP', она может быть уверена, что ее первая копия уже находится в памяти.

Если же в регистре AL вернулся код FFh, а в регистрах CX и DX - коды, например, 'OK' и 'RB', это, скорее всего означает, что закрепленная за нашей программой функция мультиплексного прерывания уже используется другой резидентной программой. В этом случае стоит сменить функцию, чтобы на возбуждать конфликтных ситуаций.

Встроим в какую-либо из описанных ранее резидентных программ механизм защиты от повторной загрузки. Это можно выполнить с любой программой; в приведенном ниже примере мы использовали программу динамического дампа из предыдущих статей. Защита от повторной загрузки требует внесения определенных изменений как в резидентную часть программы, так и в секцию инициализации. В резидентную часть следует включить обработчик прерывания 2Fh. Его расположение в пределах текста программы не имеет особого значения; мы поместили его в начале резидентной части. Секция инициализации претерпела большие изменения. Теперь она должна начинаться с вызова прерывания 2Fh с соответствующей функций для проверки на повторную установку. Если первая копия программы уже загружена, текущую программу следует завершить не функцией 31h (завершить и оставить в памяти), а обычной функцией завершения 4Ch. Если же нашей программы в памяти нет, то в секции инициализации, помимо заполнения ее "рабочего" вектора, в данном случае 03h, следует также установить наш обработчик мультиплексного прерывания.

В примере 57.1 полностью приведена секция инициализации и описана процедура new_2fh, включенная в резидентную часть программы.

Пример 57.1. Проверка резидентной программы на повторную установку

```
text    segment 'code' public byte
        assume CS:text,DS:text
        org    256
main    proc
        jmp    init
ss_seg  dw     0            ;Ячейка для содержимого SS
sp_offs dw     0            ;Ячейка для содержимого SP
mem     dw     0            ;Ячейка для временного хранения CS
old_2fh dd     0            ;Ячейка для старого содержимого
                            ;вектора 2Fh
;Резидентный обработчик мультиплексного прерывания 2Fh
new_2fh proc
        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      ;Нет, неизвестная подфункция, на выход
        mov    AL,0FFh      ;Программа уже установлена
        iret
out_2fh:jmp    CS:old_2fh   ;Переход в следующий по цепочке
                            ;обработчик прерывания 2Fh
new_2fh endp
```

```
binascl proc
        ...
binasc  endp
hexasc  proc
        ...
hexasc  endp
new_03h proc
        ...
new_03h endp
;Процедура инициализации
init    proc
;Проверим, не установлена ли уже данная программа
        mov     AH,0C8h     ;Вызов int 2Fh с функцией C8h
        mov     AL,0        ;Подфункция проверки на установку
        int     2Fh         ;Вызов мультиплексного прерывания
        cmp     AL,0FFh     ;Обработчик 2Fh вернул FFh?
        je      installed   ;Да, наша программа уже установлена
;Сохраним вектор мультиплексного прерывания 2Fh
        mov     AX,352Fh
        int     21h
        mov     word ptr old_2fh,BX
        mov     word ptr old_2fh+2,ES
;Заполним вектор мультиплексного прерывания 2Fh
        mov     AX,252Fh
        mov     DX,offset new_2fh
        int     21h
;Заполним вектор 03h
        mov     AX,2503h
        mov     DX,offset new_03h
        int     21h
;Выведем на экран информационное сообщение
        mov     AH,09h
        mov     DX,offset mes
        int     21h
;Завершим программу, оставив ее резидентной в памяти
        mov     AX,3100h
        mov     DX,(end_res-main+10fh)/16
        int     21h
installed:
;Действия в случае наличия в памяти первой копии
;Выведем на экран информационное сообщение
        mov     AH,09h
        mov     DX,offset mes1
        int     21h
;Завершим программу обычным образом
        mov     AX,4C01h
        int     21h
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    ;Конец сегмента команд
        end     main
```

Среди функций мультиплексного прерывания, предназначенных для прикладных программ, мы произвольно выбрали для программы

динамического дампа функцию C8h, а для проверки на повторную установку подфункцию 00h. Резидентный обработчик прерывания 2Fh, включенный в нашу программу, проверяет номера функции и подфункции и при обнаружении каких-либо других кодов передает управление следующему обработчику этого прерывания. Если же вызвана функция C0h с подфункцией 00h, обработчик устанавливает в регистре AL значение FFh ("я уже загружен") и возвращает управление в вызвавшую программу командой iret.

Секция инициализации начинается с проверки на повторную установку. После загрузки в регистр AH номера функции (C8h), а в регистр AL - номера подфункции (00h), вызывается прерывание 2Fh. После возврата из прерывания анализируется содержимое регистра AL. Если обработчик вернул значение FFh, программа должна завершиться без оставления в памяти. Эти действия выполняются по метке installed. Если возвращено другое значение, инициализация продолжается (для надежности стоило проверить, возвращен ли именно 0). Сохраняется старое содержимое вектора 2Fh, устанавливается наш обработчик этого прерывания, после чего выполняются все действия по установке, предусмотренные в старом варианте программы динамического дампа.
При переходе на метку installed на экран выводится сообщение о невозможности повторной установки и выполняется функция завершения 4Ch с кодом возврата 01h. Последнее, конечно, имеет символический характер, поскольку этот код в дальнейшем не анализируется. Если, однако, программа динамического дампа запускается из командного файла, например, из файла AUTOEXEC.BAT, то код возврата можно проанализировать с помощью команды if errorlevel.

Статья 58

Выгрузка резидентной программы из памяти

Еще один серьезный недостаток, присущий рассмотренным ранее резидентным программам, заключается в невозможности выгрузить их из памяти, если отпала необходимость в их использовании. Следует заметить, что в DOS вообще отсутствуют средства 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. Чаще всего в качестве "выгружающей" используют саму резидентную программу, точнее, ее вторую копию, которая, если ее запустить в определенным 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.

Рассмотрим сначала пример выгрузки из памяти резидентной программы с помощью специально запускаемой программы выгрузки. Воспользуемся для этого программой динамического дампа, в которую в предыдущей статье мы уже включили обработчик мультиплексного прерывания. Примем, что подфункция 00h прерывания 2Fh служит для проверки на повторную установку, а подфункция 01h - для выгрузки.

В секцию инициализации необходимо добавить строки сохранения старого содержимого вектора 03h. Это выполняется точно так же, как и для вектора 2Fh - с помощью функции DOS 35h. Старый вектор сохраняется в ячейке old_03h, размещаемой в резидентной части программы. Поскольку выгрузка программы выполняется с помощью прерывания 2Fh, текст обработчика этого прерывания усложняется. Новый текст обработчика приведен в примере 58.1.

;Пример 58.1. Выгрузка резидентной программы

```
main    proc
        jmp     init
ss_seg  dw      0               ;Ячейка для содержимого SS
sp_offs dw      0               ;Ячейка для содержимого SP
cs_seg  dw      0               ;Ячейка для временного хранения CS
old_2fh dd      0               ;Ячейка для старого содержимого
                                ;вектора 2Fh
old_03h dd      0               ;Ячейка для старого содержимого
                                ;вектора 03h
new_2fh proc
        cmp     AH,0C8h         ;Наша функция прерывания 2Fh?
        jne     out_2fh         ;Не наша, на выход
        cmp     AL,00h          ;Подфункция проверки на повторную
                                ;установку?
        je      ins             ;Да, сообщим о невозможности
                                ;повторной установки
        cmp     AL,01h          ;Подфункция выгрузки?
        je      uninstall       ;Да, на выгрузку
        jmp     short out_2fh;Неизвестная подфункция, на выход
ins:    mov     AL,0FFh         ;Программа уже установлена
        iret
out_2fh:jmp     CS:old_2fh ;Переход в следующий по цепочке
                                ;обработчик прерывания 2Fh
uninstall:
;Выгрузим программу из памяти, предварительно восстановив все
;перехваченные ею векторы
        push    DS              ;Сохраним
        push    ES              ;используемые
        push    DX              ;регистры
;Восстановим вектор 03h
        mov     AX,2503h        ;Функция установки вектора
        lds     DX,CS:old_03h;Заполним DS:DX
        int     21h
;Восстановим вектор 2Fh
        mov     AX,252Fh        ;Функция установки вектора
        lds     DX,CS:old_2Fh;Заполним DS:DX
        int     21h
;Получим из PSP адрес собственного окружения и выгрузим его
        mov     ES,CS:2Ch  ;ES -> окружение
        mov     AH,49h          ;Функция освобождения блока памяти
        int     21h
;Выгрузим теперь саму программу
        push    CS              ;Загрузим в ES содержимое CS,
        pop     ES              ;т.е. сегментный адрес PSP
        mov     AH,49h          ;Функция освобождения блока памяти
```

```
        int     21h
        pop     DX          ;Восстановим
        pop     ES          ;используемые
        pop     DS          ;регистры
        iret                ;Возврат в вызвавшую программу
new_2fh endp
binascl proc
...
```

Резидентный обработчик прерывания 2Fh прежде всего проверяет номер функции, поступивший в регистре AH, Если этот номер отличается от C8h, управление передается следующему обработчику по цепочке. Далее анализируется содержимое регистра AL. Если AL=00h, выполняются действия по защите от повторной загрузки. Если AL=01h, осуществляется переход на метку uninstall для выполнения действий по выгрузке программы. При любом другом номере подфункции управление передается следующему обработчику по цепочке.

По метке uninstall осуществляется сохранение используемых далее регистров (что делается скорее для красоты, чем по необходимости) и функцией DOS 25h восстанавливается из ячеек old_03h и old_2Fh исходное содержимое соответствующих векторов. Далее из ячейки с смещением 2Ch относительно начала PSP в регистр ES загружается адрес окружения программы. Сегментный адрес освобождаемого блока памяти - единственный параметр, требуемый для выполнения функции DOS 49h. Размер освобождаемого блока DOS известен, он хранится в блоке управления памятью (MCB). Далее освобождается блок памяти с самой программой. Сегментный адрес этого блока (адрес PSP) находится в регистре CS. Наконец, командой iret управление передается в программу, вызвавшую прерывание 2Fh.

Функция 49h оповещает DOS о том, что данный блок памяти свободен и может впредь использоваться DOS. Это, однако, не мешает выполняться завершающим строкам программы (в данном случае - команде iret), поскольку освобождение памяти не разрушает ее содержимого. Наша резидентная программа физически сотрется лишь после того, как в память будет загружена очередная выполняемая программа.

Для того, чтобы удалить из памяти резидентную программу динамического дампа, надо в какой-то программе вызвать прерывание 2Fh с функцией C8h и подфункцией 01h. Проще всего создать для этого специальную "выгружающую" программу.

Пример 58.2. Выгружающая программа

```
text    segment 'code'
        assume CS:text
main:   mov     AH,0C8h     ;Наша функция прерывания 2Fh
        mov     AL,01h      ;Подфункция выгрузки
        int     2fh         ;Вызов нашей резидентной программы
        mov     AX,4C00h
```

```
        int     21h
text    ends
        end     main
```

Очевидно, что рассмотренная выше методика выгрузки резидентной программы с помощью специально предназначенной для этого "выгружающей" 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 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
```

"дезактивировала" программу и выгружала ее из памяти.

Если программа запускается с клавиатуры с указанием каких-либо 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
```

Модифицируем программу динамического дампа так, чтобы ее можно было выгрузить с клавиатуры. По сравнению с примером 58.2 изменяется только секция инициализации, которая и приведена в примере 58.3.

Пример 58.3. Выгрузка резидентной программы с клавиатуры

```
;Процедура инициализации
tail    db      'off'           ;Ожидаемый хвост команды
flag    db      0               ;Флаг требования выгрузки
```

```
init    proc
;Получим хвост команды из PSP
        mov     CL,ES:80h  ;Получим длину хвоста в PSP
        cmp     CL,0       ;Длина хвоста=0?
        je      ahead      ;Да, программа запущена без параметров
        xor     CH,CH      ;Теперь CX=CL=длина хвоста
        mov     DI,81h     ;ES:DI->хвост в PSP
        mov     SI,offset tail;DS:SI->поле tail
        mov     AL,' '     ;Уберем пробелы из начала хвоста
repe    scasb              ;Сканируем хвост, пока пробелы
        dec     DI         ;DI -> первый символ после пробелов
        mov     CX,3       ;Ожидаемая длина параметра
repe    cmpsb              ;Сравниваем введенный хвост с ожидаемым
        jne     ahead      ;Введена неизвестная команда
        inc     flag       ;Введено 'off', установим флаг
                           ;запроса на выгрузку
;Проверим, не установлена ли уже данная программа
ahead:  mov     AH,0C8h
        mov     AL,0
        int     2Fh
        cmp     AL,0FFh
        je      installed  ;Программа уже установлена, при наличии
                           ;запроса на выгрузку ее можно выгрузить
;Сохраним вектор мультиплексного прерывания 2Fh
        ...
;Заполним вектор мультиплексного прерывания 2Fh
        ...
;Сохраним вектор отладочного прерывания 03h
        ...
;Заполним вектор 3
        ...
;Выведем на экран информационное сообщение
        ...
;Завершим программу, оставив ее резидентной в памяти
        ...
installed:
        cmp     flag,1     ;Запрос на выгрузку установлен?
        je      unins      ;Да, на выгрузку
;Выведем на экран информационное сообщение
        mov     AH,09h
        mov     DX,offset mes1
        int     21h
;Завершим программу обычным образом
        mov     AX,4C01h
        int     21h
unins:
;Перешлем в первую(резидентную) копию программы запрос на выгрузку
        mov     AX,0C801h  ;Функция C8h, подфункция 01h
        int     2Fh        ;Мультиплексное прерывание
;Выведем сообщение о выгрузке программы
        mov     AH,09h
        mov     DX,offset mes2
        int     21h
;Первая копия программы выгружена, завершим и эту
        mov     AX,4C00h
        int     21h
mes     db      'Программа динамического дампа загружена',10,13,'$'
```

```
mes1    db      'Программа динамического диска уже загружена',10,13
        db      'Повторная загрузка запрещена',10,13,'$'
mes2    db      'Программа выгружена из памяти',10,13,'$'
init    endp
```

К данным секции инициализации добавилась строка с ожидаемым хвостом команды и байтовый флаг запроса на выгрузку.

Поскольку действия программы при ее запуске зависят от того, введена ли команда запуска с параметром или нет, наличие хвоста в PSP анализируется в самом начале секции инициализации. При запуске программы типа .COM все сегментные регистры указывают на начало PSP. Байт с длиной хвоста (возможно, нулевой) помещается в регистр CL и сравнивается с нулем. Если в нем 0, команда запуска была введена без параметров и инициализация программы продолжается обычным образом. Если хвост имеет ненулевую длину, начинается его анализ.

Обнулением регистра CH длина хвоста "расширяется" на весь регистр CX, что нужно для организации цикла. Регистр DI настраивается на первый байт хвоста, а регистр SI - на начало поля tail с ожидаемой формой параметра. Регистр AL подготавливается для выполнения команды сканирования строки. Команда scasb сравнивает в цикле байты хвоста с содержимым AL (кодом пробела) Сравнение ведется до тех пор, пока не будет найден первый символ, отличный от пробела. Эта операция необходима из-за того, что оператор при вводе команды выгрузки может отделить параметр команды от самой команды любым числом пробелов, которые попадут в хвост команды в PSP и помешают анализировать введенный параметр.

Выход из цикла выполнения команды scasb осуществляется, когда команда проанализировала первый после пробела символ. После этого регистр DI указывает на второй символ параметра. Команда dec DI корректирует указатель DI, направляя его на первый значащий символ введенного параметра. Далее командой сравнения строк cmpsb осуществляется сравнение трех оставшихся символов хвоста. Если символы совпадают с параметром 'off', записанным в программе, устанавливается флаг запроса на выгрузку. Если результат сравнения оказался отрицательным, флаг запроса не устанавливается (и, следовательно, неправильный параметр просто не воспринимается). В любом случае осуществляется переход на продолжение программы, начинающей проверять, не установлена ли уже эта программа в памяти. Если программа еще не установлена, введенный параметр не имеет смысла. Инициализация осуществляется обычным образом: сохраняются и устанавливаются векторы и программа завершается с оставлением в памяти.

При наличии в памяти резидентной копии этой программы осуществляется переход на метку installed, где прежде всего проверяется, установлен ли флаг запроса на выгрузку. Если флаг сброшен, выводится

сообщение о невозможности повторной загрузки и программа завершается с кодом возврата 1. Если флаг запроса установлен, выполняется выгрузка программы, которая заключается в вызове мультиплексного прерывания 2Fh с функцией C8h и подфункцией 01h. Резидентный обработчик этого прерывания, включенный в состав нашей резидентной программы, отработает эту подфункцию, как и в предыдущем примере: восстановит векторы и освободит занятые программой блоки памяти. После возврата управления из обработчика в текущую программу будет выведено сообщение об успешной выгрузке и программа будет завершена функцией 4Ch с нулевым кодом возврата.

Составленная нами программа не избавлена от недостатков. Так, в ней анализируются всегда только 3 значащих символа хвоста. Таким образом, программа будет выгружена и при вводе, например, команды dump offset. Другой недостаток заключается в том, что результат сравнения записанного в программе хвоста с введенным с клавиатуры параметром будет положительным только если с клавиатуры введены строчные буквы. Команда DUMP OFF не приведет к выгрузке программы. По-настоящему следовало включить в программу перед анализом хвоста преобразование символов параметра в прописные буквы.

Статья 59

Деассемблирование и машинные коды команд

Как видно из многочисленных примеров, приведенных в этой книге, программы на языке ассемблера пишутся с использованием достаточно наглядных мнемонических обозначений команд и способов адресации. В процессе составления программы программисту, как правило, не приходится сталкиваться с машинными кодами команд. Однако многие аспекты деятельности программиста требуют хотя бы поверхностного знакомства с правилами образования машинных кодов команд. К таким аспектам можно отнести: отладку и исследование работы сложных программ; оптимизацию программ по памяти или времени выполнения; расшифровку и изучение загрузочных модулей системного и прикладного программного обеспечения. Последнее является не только весьма полезной для самообразования, но часто и необходимой операцией.

В настоящее время все программное обеспечение персональных компьютеров поступает на рынок в виде загрузочных модулей, готовых к выполнению. Исходные тексты программ практически всегда отсутствуют. При возникновении необходимости внесения в программы хотя бы незначительных изменений, приходится тем или иным способом "деассемблировать" загрузочные модули, т.е. преобразовывать их в текст на языке ассемблера, отыскивать интересующие пользователя участки и вносить затем изменения непосредственно в машинные коды команд или данных.

Для деассемблирования можно воспользоваться специальными инструментальными программами - деассемблерами, например, неплохой программой Sourcer фирмы V Communications, или, в простых случаях, встроенным деассемблером последних версий оболочки DOS Norton Commander. При этом, однако, следует иметь в виду, что деассемблер всегда выполняет свою работу не безошибочно. Ведь команды в принципе неотличимы от данных, в результате чего деассемблер обычно часть кодов команд расшифровывает, как данные (числовые или текстовые) и, наоборот, часть данных представляет в виде машинных команд. Для того, чтобы разобраться в получившейся путанице, необходимо иметь представление о правилах кодирования команд процессора. С другой стороны, во многих случаях изменения, вносимые в загрузочный модуль, столь незначительны, что не имеет смысла расшифровывать всю программу. Достаточно найти в ней несколько требуемых байтов. Эту работу вполне можно выполнить "вручную", с помощью какой-либо программы, выводящей на экран шестнадцатеричное содержимое файлов. К таким программам относятся PC Tools, Norton Commander, Нортоновские утилиты. Широко используемая процедура "подправления" программных продуктов - изменение программных строк анализа версии DOS с целью эксплуатации старой программы в новой версии DOS (естественно, этот метод применим лишь в тех случаях, когда анализ версии DOS носит формальный характер, для самой же программы по существу версия DOS безразлична).

Система кодирования команд в процессорах Intel весьма сложна, не отличается последовательностью и иногда неоднозначна. С другой стороны, она обеспечивает достаточно гибкие механизмы адресации и эффективное использование памяти, отводимой программе. Рассмотрим общие закономерности образования машинных кодов команд процессора 8086. Приводимый материал в целом справедлив и для процессоров 80386 и 80486, однако включение в архитектуру этих процессоров дополнительных способов адресации, а также расширение регистров до 32 разрядов еще более усложняет систему кодирования команд.

В общем виде структуру машинной команды можно представить следующим образом:

Префикс - Код операции - Адресация - Смещение - Операнд

Каждый элемент команды занимает один или несколько байтов. Необходимым является только код операции (коп), который характеризует выполняемую операцию (пересылка, сложение) и, естественно, должен присутствовать в любой команде.

Префикс команды занимает 1 байт и задает либо замену сегментного регистра, используемого по умолчанию (ES:, CS:), либо повторение команды CX раз (rep, repe).

Формат префикса замены сегмента приведен на рис. 59.1; в табл. 59.1 указаны значения поля SR (код сегментного регистра) для этого байта.

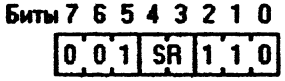


Рис. 59.1. Структура байта префикса замены сегмента.

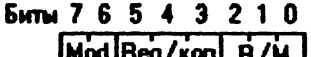
Префиксы повторения rep и repe имеют коды F3h, а префикс repne - код F2h (префиксы rep и repe используются в разных командах, поэтому не возникает неоднозначность при расшифровке кода).

Код операции занимает обычно 1 байт и характеризует команду, при этом многим командам присущи несколько кодов операций, которые частично определяют использованный способ адресации.

Байт адресации в оригинальной литературе носит название MOD R/M и состоит из трех полей: Mod, Reg (или Reg/коп) и R/M (рис. 59.2).

Таблица 59.1. Расшифровка поля SR

Поле SR	Регистр
00	ES
01	CS
10	SS
11	DS



0 0 - В коде команды отсутствует смещение (при адресации к памяти)
0 1 - Смещение в коде команды есть и занимает 1 байт
1 0 - Смещение в коде команды есть и занимает 1 слово
1 1 - Операнды являются регистрами

Рис. 59.2. Структура байта MOD R/M с расшифровкой поля Mod.

Как видно из рис. 59.2, поле Mod занимает 2 бит; в нем указывается, выполняется ли адресация к памяти или к регистрам. Если оба операнда являются регистрами, Mod=11; при других значениях Mod один из операндов находится в памяти. Как известно (см. статью 30) для обращения к памяти предусмотрено несколько способов адресации: прямая, косвенная регистровая и косвенная регистровая со смещением. Ес-

ли Mod=00, смещение в команде отсутствует и, следовательно, выполняется команда типа inc word ptr [BX] или mov AX,[BX][SI]. Если Mod=01, в команде указано смещение, причем оно лежит в диапазоне знаковых байтовых чисел: mov [BX][SI+127],CX или inc word ptr [BX-2]. Если Mod=10, в команде присутствует смещение длиной 2 байт: dec mem[SI] или neg byte ptr [BX][DI+128] (адрес ячейки памяти mem с очевидностью является словом, а число 128 выходит за рамки байтовых знаковых чисел).

Поле Reg/коп для некоторых команд используется, как расширение байта кода операции; в других случаях в нем указывается условный код одного из адресуемых регистров (приемника или источника), причем по по младшему биту кода операции процессор определяет, надо ли использовать полный 16-разрядный регистр или одну из его половин. В табл. 59.2 показано соответствие кодов поля Reg регистрам процессора.

Таблица 59.2. Коды регистров поле Reg

Поле Reg	Регистр 16 бит	Регистр 8 бит
000	AX	AL
001	CX	CL
010	DX	DL
011	BP	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

Поле R/M (Register/Memory, регистр/память) используется для идентификации способа адресации. Расшифровка кода в этом поле зависит от значения поля Mod того же байта. Если Mod=11, в команде используется регистровая адресация, и в поле R/M указывается код регистра, (который может быть как источником, так и приемником); при других значениях Mod выполняется обращение к памяти и в поле R/M закодирована комбинация регистров, используемых для косвенного обращения памяти, а также наличие или отсутствие в команде смещения. В табл. 59.3 приведена расшифровка кодов в поле R/M для различных значений Mod.

Обратите внимание на отсутствие во второй графе таблицы кода R/M для описания адресации через BP. Это не означает, что регистр BP нельзя использовать для косвенного обращения к памяти. Команда типа mov [BP],CX вполне законна, но транслятор преобразует ее к виду mov [BP+смещение],CX, где смещение указывается в третьем байте команды и равно 0.

Таблица 59.3. Расшифровка поля R/M

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

Некоторые одноoперандные команды в случае регистрового способа адресации (push BX, inc CX) кодируются одним байтом, в который входит, наряду с кодом операции, и код регистра-операнда. Формат таких команд приведен на рис. 59.3; коды регистров в этом случае соответствуют табл. 59.2 (столбец 16-разрядных регистров).

Рис. 59.3. Структура байта префикса замены сегмента.

Аналогичная ситуация возникает при использовании в некоторых командах в качестве операнда сегментного регистра. Такие команды могут иметь для этих случаев особый код операции, в котором предусмотрено двухбитовое поле SR для обозначения сегментного регистра. Коды регистров соответствуют табл. 59.1.

В первом байте команды - байте кода операции, в ряде случаев целесообразно выделять два младших бита. Формат такого байта изображен на рис. 59.4.

Бит 0 (W) принимает значение 1, если команда оперирует со словом. В случае байтовой операции W=0.

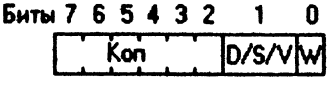


Рис. 59.4. Структура байта кода операции для некоторых команд.

Бит 1 в одних командах обозначает направление операции (D), в других задает характеристики непосредственного операнда (S). В командах сдвигов этот бит (тогда он обозначается V) определяет наличие счетчика сдвигов.

Если D=1, операция выполняется в регистр Reg; если D=0 - из регистра Reg.

Если SW=01, то выполняется операция со словом (W=1), причем операнд образует 16 бит непосредственных данных (add mem,1234h). Если SW=0, то выполняется операция со словом, но в качестве операнда в команде указано байтовое знаковое число (add mem,5), и этот байт непосредственных данных расширяется со знаком для образования "законного" 16-битового операнда.

На рис. 59.5...59.7 приведены примеры кодирования команд.

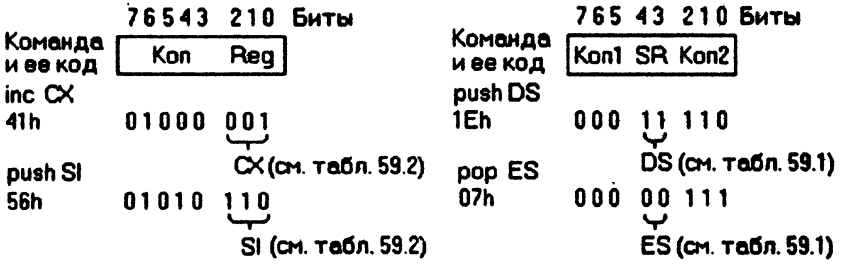


Рис. 59.5. Примеры однобайтовых команд с обращением к регистрам общего назначения и сегментным.

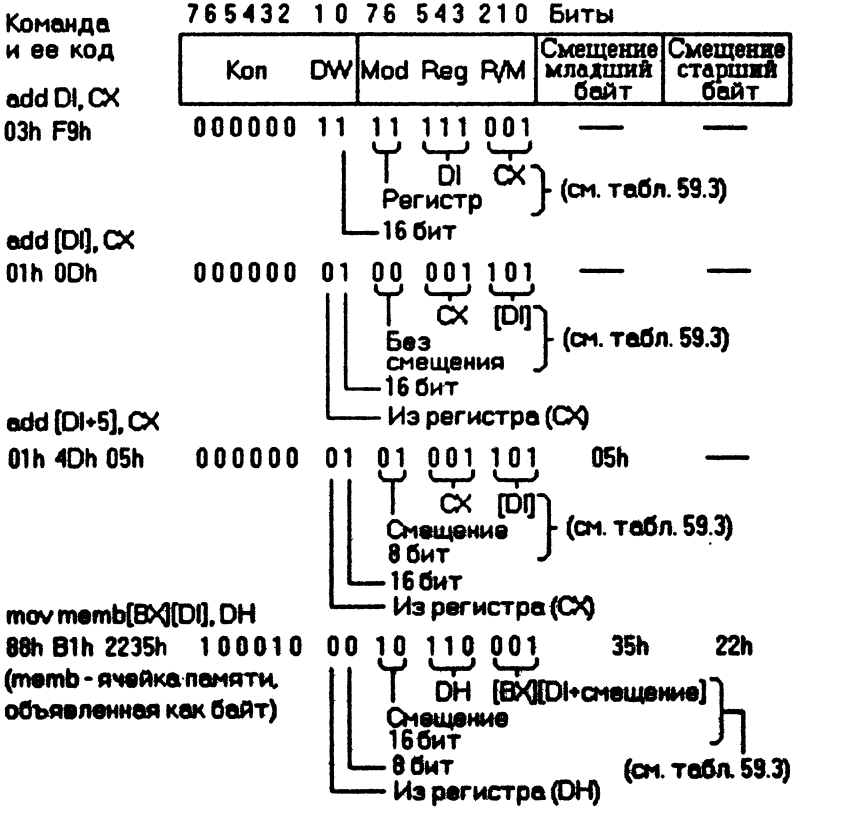


Рис. 59.6. Примеры команд, использующих смещение.

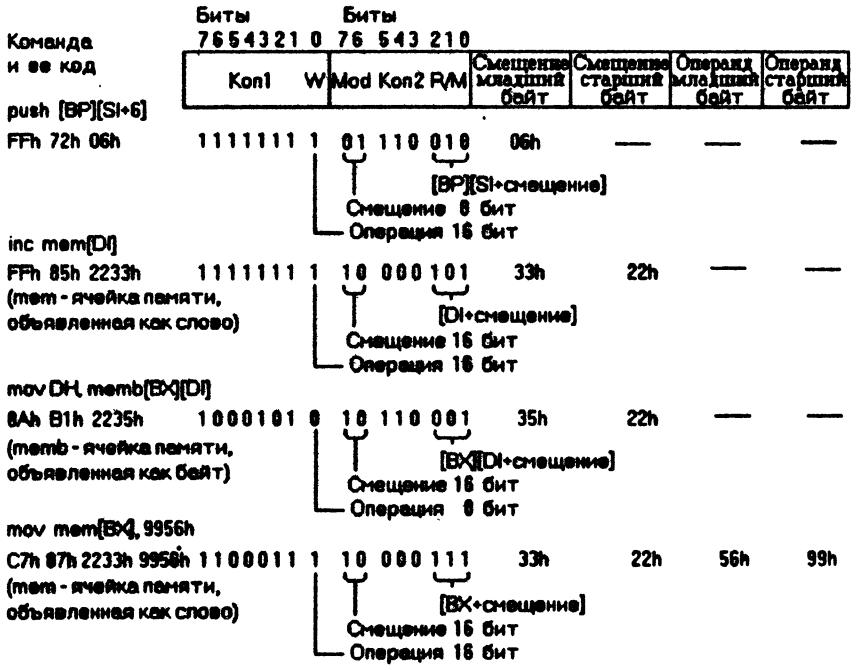


Рис. 59.7. Примеры команд с расширенным кодом операции.

Часть 2

Программирование защищенного режима

Статья 60

Особенности процессоров 80386 - 486

С появлением 32-разрядных процессоров 80386 фирмы Intel, программисты значительно расширили спектр своих возможностей. Эти процессоры могут работать в трех режимах: реальном, защищенном и виртуального процессора 8086.

В реальном режиме достигается полная совместимость с программным обеспечением, разработанным для процессора 8086. Однако в этом случае выполняемая программа может использовать только адресное пространство памяти размером в 1Мбайт. В защищенном режиме используются полные возможности 32-разрядного процессора - обеспечивается непосредственный доступ к 4 Гбайт физического адресного пространства и многозадачный режим. Реализовать многозадачность для программ, разработанных для процессора 8086, можно с помощью режима виртуального процессора 8086. В этом случае операционная система защищенного режима позволяет создать несколько задач, каждая из которых будет выполняться как бы отдельным процессором 8086.

Основные отличия следующей разработки фирмы Intel, процессора 486DX, состоят в том, что математический сопроцессор реализован на одном кристалле вместе с центральным процессором и имеется внутренняя встроенная кэш-память. Поскольку с точки зрения программиста эти процессоры практически не различаются, мы, говоря о процессоре МП 486, будем подразумевать обе модификации - 80386 и i486, а также многочисленные разработки других фирм, совместимые с исходными процессорами фирмы Intel.

Процессор 486 содержит около 40 программно-адресуемых регистров (не считая регистров сопроцессора), из которых десять являются 16-разрядными, а остальные - 32-разрядными. Регистры принято объединять в семь групп: регистры общего назначения (или регистры данных), регистры-указатели, сегментные регистры, управляющие регистры, регистры системных адресов, отладочные регистры и регистры тестирования. Кроме того, в отдельную группу выделяют счетчик команд и регистр флагов (рис. 60.1...60.6).

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

Рис. 60.1. Регистры общего назначения, указатели и сегментные.

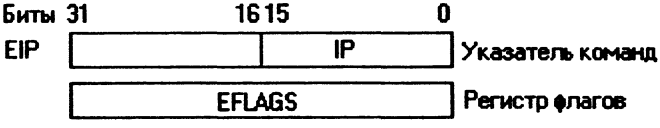


Рис. 60.2. Указатель команд и регистр флагов.

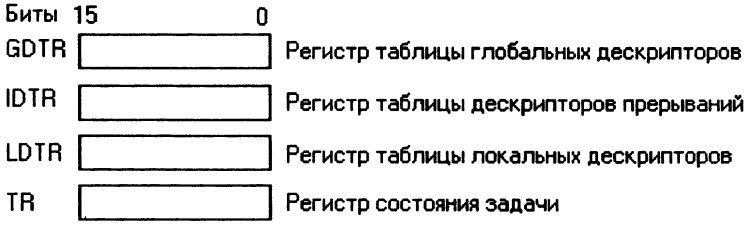


Рис. 60.3. Регистры системных адресов.

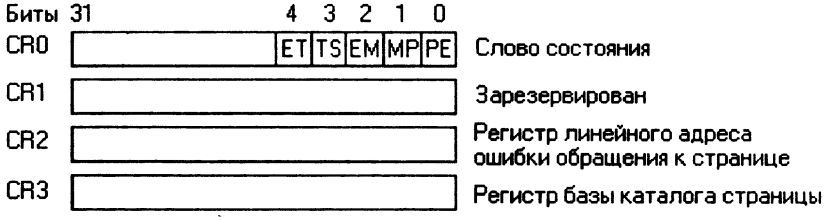


Рис. 60.4. Регистры управляющие.

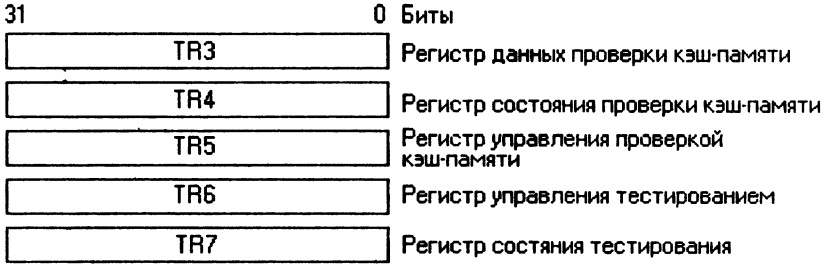


Рис. 60.5. Регистры тестирования.

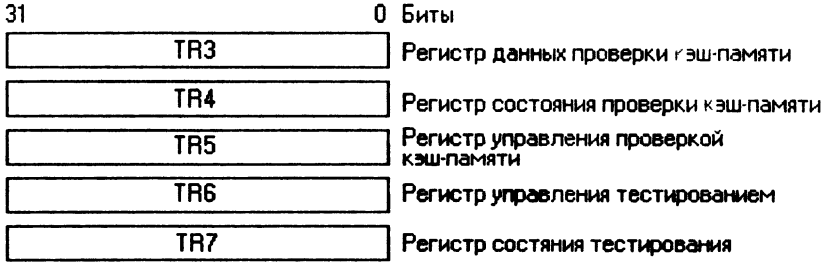


Рис. 60.6. Регистры отладочные.

Для сохранения совместимости с ранними моделями процессоров допускается обращение к младшим половинам всех регистров, которые имеют те же мнемонические обозначения, что и в МП 8086 (AX, BX,.CX, DX, SI, DI, BP и SP). Естественно, сохранена возможность работы с младшими (AL, BL, CL и DL) и старшими (AH, BH, CH и DH) половинками регистров МП 8086. Однако старшие половины 32-разрядных регистров МП 486 не имеют мнемонических обозначений и непосредственно недоступны. Для того, чтобы прочитать, например, содержимое старшей половины регистра EAX (биты 31...16) придется сдвинуть все содержимое EAX на 16 бит вправо (в регистр AX) и прочитать затем содержимое AX. Все регистры общего назначения и указатели программист может использовать по своему усмотрению для временного хранения адресов и данных размером от байта до двойного слова. Так, например, возможно использование следующих команд:

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

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

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

Регистр флагов процессора 486 принято называть EFLAGS. Дополнительно к шести флагам состояния (CF, PF, AF, ZF, SF и OF) и трем флагам управления состоянием процессора (TF, IF и DF), назначение которых было описано в статье 3, он включает три новых флага NT, RF и VM и двухбайтовое поле IOPL (рис. 60.7).

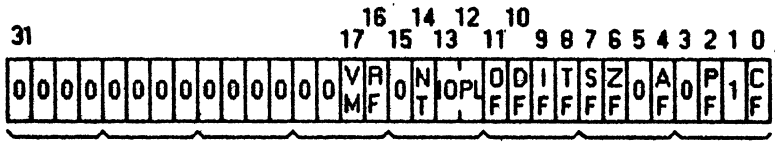


Рис. 60.7. Регистр флагов EFLAGS.

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

Двухразрядное поле привилегий ввода-вывода IOPL (Input/Output Privilege Level) указывает на максимальное значение уровня текущего

приоритета (от 0 до 3), при котором команды ввода-вывода выполняются без генерации исключительной ситуации.

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

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

Управляющий флаг виртуального режима VM (Virtual Mode) используется для перевода МП 486 из защищенного режима в режим виртуального процессора 8086. В этом случае процессор 486 функционирует как быстродействующий микропроцессор 8086, но реализует механизмы защиты памяти, страничной адресации и ряд других возможностей.

При работе с процессором 486 программист имеет доступ к трем управляющим регистрам, в которых содержится информация о состоянии компьютера (см. рис. 60.4). Эти регистры доступны только в защищенном режиме для программ, имеющих уровень привилегий 0.

Регистр CR0 представляет собой слово состояния системы. Для управления режимом работы процессора 486 и указания его состояния используются следующие шесть битов:

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

Бит типа сопроцессора ET (Processor Extension Type) в МП 80286 - 80386 указывает на тип подключенного сопроцессора. Если ET=1, то 80387, если ET=0, то 80287. В МП 486 бит ET всегда установлен.

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

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

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

Бит разрешения защиты PE (Protection Enable). При PE=1 процессор работает в защищенном режиме; при PE=0 - в реальном. PE может

быть установлен при загрузке регистра CR0 командами lmsw или mov CR0, а сброшен только командой mov CR0.

Регистр CR1 зарезервирован фирмой Intel для последующих моделей процессоров.

Регистры CR2 и CR3 служат для поддержки страничного преобразования адреса. Эти два регистра используются вместе. CR2 содержит полный линейный адрес, вызвавший исключительную ситуацию на последней странице, а CR3 - адрес, указывающий базу каталога страницы.

Регистры системных адресов (см. рис. 60.3) используются в защищенном режиме работы процессора 486. Они задают расположение системных таблиц, служащих для организации сегментной адресации в защищенном режиме. В состав процессора 486 входят четыре регистра системных адресов:

GDTR (Global Descriptor Table Register) - регистр таблицы глобальных дескрипторов, в который с помощью специальной структуры данных - псевододескриптора, загружаются характеристики таблицы глобальных дескрипторов (линейный базовый адрес и граница).

LDTR (Local Descriptor Table Register) - регистр таблицы локальных дескрипторов, в который загружается селектор сегмента таблицы локальных дескрипторов.

IDTR (Interrupt Descriptor Table Register) - регистр таблицы дескрипторов прерываний, в который с помощью псевдодескриптора загружаются характеристики сегмента таблицы дескрипторов прерываний.

TR (Task Register) - регистр состояния задачи, в который загружается селектор сегмента состояния задачи.

Отладочные регистры (см. рис. 60.6) предназначены для отладки программ с использованием аппаратных возможностей процессора 486. Все эти регистры являются 32-разрядными и содержат следующую информацию:

- регистры DR0, DR1, DR2, DR3 (Debug Register)- линейные адреса точек останова 0, 1, 2 и 3 соответственно;

- регистры DR4 и DR5 зарезервированы фирмой Intel для следующих версий;

- регистр состояния точки останова DR6 содержит информацию, позволяющую отладчику определить, какое из условий отладки удовлетворено.

- регистр управления отладкой DR7 задает вид доступа к памяти, связанный с каждой контрольной точкой.

Структура регистров DR6 и DR7 показана на рис 60.8.

Регистр состояния точки останова DR6 содержит информацию об условиях, выявленных во время генерирования отладочного исключения. При выявлении разрешенного исключения отладки устанавливается соответствующий разряд Bn.

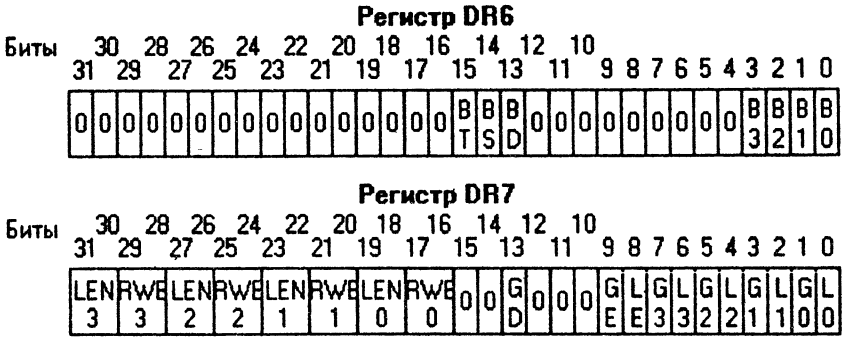


Рис. 60.8. Отладочные регистры DR6 и DR7.

Бит BT связан с отладочным битом ловушки T в TSS. Процессор устанавливает бит BT перед входом в отладочный обработчик исключений в том случае, если произошло переключение на задачу с установленным битом T.

Бит BS связан с флагом TF. Он устанавливается, если отладочное исключение произошло при пошаговом режиме выполнения.

Бит BD установлен, если следующая команда будет читать или записывать один из восьми отладочных регистров, в то время, как они используются встроенным эмулятором.

Как уже говорилось ранее, регистр управления отладкой DR7 задает вид доступа к памяти, связанный с каждой контрольной точкой. Каждому адресу в регистрах DR0 - DR3 соответствует свое поле RWE0...RWE3 в регистре DR7. Содержимое этого поля процессор интерпретирует следующим образом:

00 - останов только при выполнении команды;

01 - останов только на записи данных;

10 - это значение не определено

11 - останов на чтении или на записи данных, но не на вызове команды.

Поля LEN0 - LEN3 определяют размер единичной пересылки данных. Поле длины интерпретируется следующим образом:

00 - длина 1 байт;

01 - длина 2 байт;

10 - это значение не определено;

11 - длина 4 байта;

Биты Gn и Ln показывают разрешения глобальной и локальной точек останова соответственно.

Два из регистров тестирования, TR6 и TR7 (см. рис. 60.5) используются для проверки буфера трансляции адресов процессора, а осталь-

ные 3 (TR3, TR4 и TR5) - для контроля правльности работы внутренней кэш-памяти процессора.

Рассмотрим на простых примерах работу с регистрами МП 486.

Пример 60.1 показывает принципы работы с 32-разрядными регистрами. В регистр EAX заносится 8-битовое шестнадцатеричное число, выполняется операция сложения, а результат записывается в память. Заметим, что для сохранения результата нам требуется 8 байтов, поэтому для описания поля sum следует использовать директиву dd.

Пример 60.1. Программа сложения 32-разрядных операндов.

```
.386
text    segment 'code' use16
        assume CS:text,DS:data
begin:  mov    AX,data
        mov    DS,AX
;Основной фрагмент программы
        mov    EAX,12345678h;Загрузим операнд в 32-разрядный EAX
        add    EAX,87654321h;Выполним сложение
        mov    dword ptr sum,EAX;Запишем результат в поле sum
;Завершение программы
        mov    AX,4C00h
        int    21h
text    ends
data    segment
sum     dd     0
data    ends
end     begin
```

Поскольку в данном примере обрабатываются 32-разрядные числа, в текст программы необходимо включить директиву .386, разрешающую использование команд МП 386 и 486, в частности, для работы с 32-разрядными операндами.

Рассмотрим листинг этой программы, из которого для краткости удалены комментарии, но добавлены номера строк (рис. 60.9). В нем мы увидим как команды МП 8086 для работы с 16-разрядными операндами, так и команды МП 386 для работы с 32-разрядными операндами.

Например, в строках 4 и 6 листинга используется команда засылки операнда в аккумулятор (B8h). Однако в строке 6 наличие перед кодом этой команды префикса замены размера операнда (код 66h) определяет, что длина операнда равна 32 бита, и, следовательно, используется регистр EAX. Префикс замены размера операнда вставляется в объектный модуль транслятором автоматически, если в программе указано мнемоническое обозначение 32-разрядного регистра, например, EAX.

Процесс выполнения этой и последующих программ удобно контролировать с помощью отладчика CodeView. Однако для индикации содержимого 32-разрядных регистров требуется провести дополнительную настройку отладчика. Рассмотрим, как это делается.

```
1:                                .386
2:   0000                        text    segment 'code' use16
3:                                assume  CS:text,DS:data
4:   0000  B8 ---- R              begin:  mov         AX,data
5:   0003  8E D8                          mov         DS,AX
6:   0005  66 | B8 12345678              mov         EAX,12345678h
7:   000B  66 | 05 87654321              add         EAX,87654321h
8:   0011  66 | A3 0000 R                mov         dword ptr sum,EAX
9:   0015  B8 4C00                        mov         AX,4C00h
10:  0018  CD 21                          int         21h
11:  001A                         text    ends
12:  0000                         data    segment
13:  0000  00000000               sum     dd 0
14:  0004                         data    ends
15:                               end     begin
```

Рис. 60.9. Листинг ассемблирования программы примера 60.1.

Войдя в отладчик, выберите в главном меню пункт Options, а в меню этого пункта - опцию 386. Это обеспечит вывод на экран содержимого полных 32-разрядных регистров EAX - ESP взамен 16-разрядных регистров AX - SP. Информационный кадр отладчика, в котором видны результаты отладки программы из примера 60.1, приведен на рис. 60.10.

На рисунке показано содержимое регистров процессора и поля данных sum после выполнения команды

```
mov    dword ptrsum,EAX
```

```
File  View  Search  Run  Watch | Options | Language  Calls  Help | F8=Trace F5=Go
                                 n Flip/Swap
1:      .386                     n Bytes Coded                          EAX=99999999
2:      ;Начало программы          Case Sense                           EBX=00000000
3:      text    segment 'code'   n 386                                  ECX=00000000
4:              assume  CS:text                                         EDX=00000000
5:      begin:                                                          ESP=00000000
6:              mov     AX,data                                         EBP=00000000
7:              mov     DS,AX                                           ESI=00000000
8:      ;Основной фрагмент программы                                    EDI=00000000
9:              mov     EAX,12345678h ;Загрузим операнд в 32-разр        DS=....5916
10:                                   ;регистр EAX,                      ES=....5904
11:             add     EAX,87654321h ;выполним сложение                 FS=....0000
12:             mov     dword ptr sum,EAX ;и запишем результат в         GS=....0000
13:     ;Завершение программы                                           SS=....5914
14:             mov     AX,4C00h                                        CS=....5914
15:             int     21h                                             IP=00000015
16:     text    ends
17:     data    segment                                                    NV UP
18:     sum     dd      0                                                  EI NG
                                                                           NZ NA
>db 0                                                                      PE NC
5916:0000  99 99 99 99 4E 42 30 30-69 00 00 00 00 00 00 00   ....NB
5916:0010  1A 00 00 00 00 00 00 00-05 50 2E 4F 42 4A 01 03   .....
>
```

Рис. 60.10. Информационный кадр отладчика для программы, включающей команды с 32-разрядными операндами

Вторая программа, текст которой приведен в примере 60.2, показывает, как можно использовать 32-разрядный регистр-счетчик ECX. Для того, чтобы организовать достаточно длительную программную задержку, в МП 8086 приходится использовать относительно сложную конструкцию со вложенными циклами. В МП 486 достаточно указать команде loop, чтобы она использовала 32-разрядный счетчик ECX. В этом случае число повторов может достигать 0FFFFFFFFh.

Пример 60.2. Реализация задержки с помощью 32-разрядного регистра.

```
.386
text    segment 'code' use16
assume  CS:text,DS:data
begin:  mov     AX,data
        mov     DS,AX
;Основной фрагмент программы
        mov     CX,3            ;Число повторов внешнего цикла
asdf:   push    CX              ;Начало внешнего цикла
        mov     AH,40h          ;Вывод сообщения
        mov     DX,offset message
        mov     CX,3
        int     21h
        mov     ECX,0F0000000h;Засылка числа шагов внутреннего
                                ;цикла в регистр ECX
qwer:   db      67h             ;Задание размера операнда для команды
loop
        loop    qwer
        pop     CX
        loop    asdf            ;Завершение внешнего цикла
;Завершение программы
        mov     AX,4C00h
        int     21h
text    ends
data    segment
message db      '<>'
data    ends
stk     segment stack 'stack'
        db      256 dup ('^')
stk     ends
end     begin
```

Обратите внимание, что перед обращением к команде loop в текст программы включено определение байта данных - db 67h. Этим несколько необычным способом обеспечивается установка префикса замены размера адреса (код 67h) для следующей команды, которая теперь будет работать с регистром ECX. Если префикс замены размера операнда отсутствует, команда loop работает с регистром CX. Здесь транслятор не может определить, операнд какого размера имеется в виду, и префикс приходится устанавливать в явном виде.

До сих пор мы рассматривали, как используются уже известные нам команды процессора для обработки 32-разрядных операндов. Однако в систему команд процессора 486 включен ряд новых команд, выполне-

ние которых не поддерживается процессорами 8086 - 80286. Приведем список этих команд.

Команды общего назначения

bound - проверка индекса массива относительно границ массива.

bsf/bsr - команды сканирования битов.

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

enter - создание кадра стека для параметров процедур языка высокого уровня.

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

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

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

leave - выход из процедуры языка высокого уровня и восстановление регистров, записанных в стек командой enter.

lss/lfs/lgs - команды загрузки сегментных регистров.

mov DRx,*reg*; *reg*,DRx

mov CRx,*reg*; *reg*,CRx

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

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

push *imm* - запись в стек непосредственного операнда размером байт, слово или двойное слово (например push 0FFFFFFFFh).

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

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

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

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

rcr/rcl/ror/rol *reg/mem,imm* - циклический сдвиг на непосредственно заданное значение.

sar/sal/shr/shl *reg/mem,imm* - арифметический сдвиг на непосредственно заданное значение.

shrd/shld - установка байта по условию.

Команды защищенного режима

arpl - корректировка поля RPL (уровня запрашиваемого приоритета) селектора.

clts - сброс флага переключения задач в регистре CR0.

lar - загрузка байта разрешения доступа.
lgdt - загрузка регистра таблицы глобальных дескрипторов.
lidt - загрузка регистра таблицы дескрипторов прерываний.
lldt - загрузка регистра таблицы локальных дескрипторов.
lmsw - загрузка слова состояния.
lsl - загрузка границы сегмента.
ltr - загрузка регистра задачи.
sgdt - сохранение регистра таблицы глобальных дескрипторов.
sidt - сохранение регистра таблицы дескрипторов прерываний.
sldt - сохранение регистра таблицы локальных дескрипторов.
smsw - сохранение слова состояния.
ssl - сохранение границы сегмента.
str - сохранение регистра задачи.
verr - проверка доступности сегмента для чтения на текущем уровне привилегии.
verw - проверка доступности сегмента для записи на текущем уровне привилегии.

Примеры использования некоторых из перечисленных выше команд покажем в приведенной далее программе.

Пример 60.3. Использование команд процессора 486.

```
.386
text    segment 'code' use16
        assume CS:text,DS:data
begin:  mov    AX,data
        mov    DS,AX
;Основной фрагмент программы
        mov    EAX,0FF000000h
        mov    EBX,64h
        mov    CL,0Ah
        shld   EBX,EAX,CL ;Сдвиг содержимого EBX влево на 0Ah
                          бит и объединение его содержимого с
                          ;содержимым EAX. Результат будет в EBX
        bsf    AX,EBX     ;Проверка разрядов в регистре EBX,
                          ;начиная с младшего, и запись номера
                          ;первого слева разряда, в котором
                          ;находится 1, в регистр AX
        bsr    DX,EBX     ;Проверка разрядов в регистре EBX,
                          ;начиная со старшего, и запись
                          ;номера первого справа разряда, в
                          ;котором находится 1, в регистр DX
        mov    dword ptr sum,EBX;Сохранение суммы
;Завершение программы
        mov    AX,4C00h
        int    21h
text    ends
data    segment
sum     dd     0
data    ends
end     begin
```

В программе выполняется умножение содержимого регистра EBX на два в десятой степени. Для этого используется команда сдвига влево на 10 разрядов (0Ah). С помощью этой же команды обеспечивается прибавление к полученному результату числа 3FCh. Слагаемое предварительно записывается в регистр EAX, содержимое которого при выполнении команды побитно перемещается в регистр EBX, начиная со старшего разряда. Поскольку выполняется сдвиг на 10 разрядов, в регистре EAX находится число 0FF000000h.

Следующие две команды вычисляют номера разрядов, в которых находится старший и младший отличные от нуля биты полученного числа. Старший бит определяется командой сканирования начиная со старшего разряда - bsr и помещается в регистр DX. Для определения младшего используется команда bsl.

Процесс выполнения приведенного фрагмента и работу отдельных команд удобно наблюдать в отладчике.

Статья 61

Первое знакомство с защищенным режимом

Микропроцессор МП 486, так же, как и его предшественники МП 268 и 386, может работать в двух режимах: реального адреса и виртуального защищенного адреса. Обычно эти режимы называют просто реальным и защищенным. В реальном режиме микропроцессоры 80x86 функционируют фактически так же, как МП 8086 с повышенным быстродействием и расширенным набором команд. Многие весьма привлекательные возможности микропроцессоров принципиально не реализуются в реальном режиме, который введен лишь для обеспечения совместимости с предыдущими моделями процессоров. Все программы, приведенные в предыдущих статьях этой книги, относятся к реальному режиму и могут с равным успехом выполняться на любом из этих микропроцессоров без каких-либо изменений. Характерной особенностью реального режима является ограничение объема адресуемой оперативной памяти величиной 1 Мбайт.

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

- увеличение адресуемого пространства до 16 Мбайт для МП 286 и до 4 Гбайт для МП 386 и 486;

- возможность работать в виртуальном адресном пространстве, превышающем максимально возможный объем физической памяти. Для МП 286 виртуальное пространство составляет 1 Гбайт, а для МП 386 и 486 - огромную величину 64 Тбайт. Правда, для реализации виртуального режима необходимы, помимо дисков большой емкости, еще и соответствующая операционная система, которая хранит все сегменты выполняемых программ в большом дисковом пространстве, автоматически загружая в оперативную память те или сегменты по мере необходимости;

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

- страничная организация памяти, повышающая уровень защиты задач друг от друга и эффективность их выполнения.

При включении микропроцессора в нем автоматически устанавливается режим реального адреса. Переход в защищенный режим осуществляется программно путем выполнения соответствующей последовательности команд. Поскольку многие детали функционирования микропроцессора в реальном и защищенном режимах существенно различаются, программы, предназначенные для защищенного режима, должны быть написаны особым образом. Реальный и защищенный режимы не совместимы!

Архитектура современного микропроцессора необычайно сложна. Столь же сложными оказываются и программы, использующие средства защищенного режима. К счастью, однако, отдельные архитектурные особенности защищенного режима оказываются в достаточной степени замкнутыми и не зависящими друг от друга. Так, при работе в однозадачном режиме отпадает необходимость в изучении многообразных и замысловатых методов взаимодействия задач; во многих случаях можно отключить (или, точнее, не включать) механизм страничной организации памяти; часто нет необходимости использовать уровни привилегий. Все эти ограничения существенно упрощают освоение защищенного режима.

Начнем изучение защищенного режима с рассмотрения простейшей (но, к сожалению, все же весьма сложной) программы, которая, будучи запущена обычным образом под управлением MS-DOS, переключает процессор в защищенный режим, выводит на экран для контроля несколько символов, переходит назад в реальный режим и завершается

стандартным для DOS образом. Рассматривая эту программу, мы познакомимся с основополагающей особенностью защищенного режима - сегментной адресацией памяти, которая осуществляется совсем не так, как в реальном режиме.

Следует заметить, что программы этого раздела книги отлаживались на машине с процессором 486, и именно о нем будет в первую очередь идти речь. Более простой МП 286 отличается форматом регистров, сокращенным набором команд и другими архитектурными упрощениями. Для перевода на этот микропроцессор некоторые программы потребуют определенных переделок. МП 386 с точки зрения программирования почти не отличается от МП 486 и для него программы переделывать не придется. Для подготовки программ к выполнению использовались транслятор MASM и компоновщик LINK фирмы Microsoft. Перед запуском программ защищенного режима следует выгрузить драйверы обслуживания расширенной памяти (типа EMM386.EXE) и оболочку Windows.

Пример 61.1. Переход в защищенный режим и обратно.

```
;В защищенном режиме вывод фиксированных символов на экран
.386P                          ;(1)Разрешение трансляции всех, в том
                               ;числе привилегированных команд МП 386
                               ;и 486
;Структура для описания дескрипторов сегментов
descr   struc                  ;(2)
limit   dw      0              ;(3)Граница (биты 0...15)
base_l  dw      0              ;(4)База, биты 0...15
base_m  db      0              ;(5)База, биты 16...23
attr_1  db      0              ;(6)Байт атрибутов 1
attr_2  db      0              ;(7)Граница (биты 16...19) и атрибуты 2
base_h  db      0              ;(8)База, биты 24...31
descr   ends                   ;(9)
data    segment                ;(10)Начало сегмента данных
;Таблица глобальных дескрипторов GDT
gdt_null descr <0,0,0,0,0,0>;  (11)Селектор 0 - обязательный
                               ;нулевой дескриптор
gdt_data descr <data_size-1,0,0,92h,0,0>;(12)Селектор 8, сегмент
                               ;данных
gdt_code descr <code_size-1,0,0,98h,0,0>;(13)Селектор 16, сегмент
                               ;команд
gdt_stack descr <255,0,0,92h,0,0>;(14)Селектор 24, сегмент стека
gdt_screen descr <4095,8000h,0Bh,92h,0,0>;(15)Селектор 32,
                               ;видеобуфер
gdt_size=$-gdt_null            ;(16)Размер GDT
;Поля данных программы
pdescr  dq      0              ;(17)Псевдодескриптор для lgdt
real_sp dw      0              ;(18)Ячейка для хранения SP
sym     db      1              ;(19)Символ для вывода на экран
attr    db      1Eh            ;(20)Его атрибут
mes db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$';(21)
data_size=$-gdt_null           ;(22)Размер сегмента данных
data    ends                   ;(23)Конец сегмента данных
```

```
text    segment 'code' use16; (24)Начало сегмента команд. Будем
                             ;работать в 16-разрядном режиме
        assume CS:text,DS:data; (25)
main    proc                ; (26)
        mov   AX,data       ; (27)Инициализация реального
        mov   DS,AX         ; (28)режима
;Вычислим 32-битовый линейный адрес сегмента данных и загрузим его
;в дескриптор сегмента данных в таблице GDT. В регистре AX уже
;находится сегментный адрес. Умножим его на 16 сдвигом влево на 4
;бита с размещением результата в регистрах DL:AX
        mov   DL,0          ; (29)Очистим DL
        shld  DX,AX,4       ; (30)Сдвинем биты 12...15 AX в DL
        shl   AX,4          ; (31)Сдвинем влево AX на 4 бита
;Теперь в DL:AX 32-битовый линейный адрес сегмента данных
        mov   BX,offset gdt_data; (32)В BX адрес дескриптора
        mov   [BX].base_l,AX ; (33)Загрузим младшую часть базы
        mov   [BX].base_m,DL ; (34)Загрузим среднюю часть базы
;Вычислим 32-битовый линейный адрес сегмента команд и загрузим его
;в дескриптор сегмента команд в таблице глобальных дескрипторов
        mov   AX,CS         ; (35)AX=адрес сегмента команд
        mov   DL,0          ; (36)Та же процедура умножения
        shld  DX,AX,4       ; (37)сегментного адреса на 16
        shl   AX,4          ; (38)сдвигом влево на 4 бита
        mov   BX,offset gdt_code; (39)BX=адрес дескриптора
        mov   [BX].base_l,AX ; (40)Загрузка младшей
        mov   [BX].base_m,DL ; (41)и средней частей базы
;Аналогично для адреса сегмента стека
        mov   AX,SS         ; (42)
        mov   DL,0          ; (43)Та же процедура умножения
        shld  DX,AX,4       ; (44)сегментного адреса на 16
        shl   AX,4          ; (45)сдвигом влево на 4 бита
        mov   BX,offset gdt_stack; (46)BX=адрес дескриптора
        mov   [BX].base_l,AX ; (47)Загрузка младшей
        mov   [BX].base_m,DL ; (48)и средней частей базы
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        mov   BX,offset gdt_data; (49)Адрес GDT
        mov   AX,[BX].base_l ; (50)Получим и занесем в pdescr
        mov   word ptr pdescr+2,AX; (51)базу, биты 0...15
        mov   DL,[BX].base_m ; (52)Получим и занесем в pdescr
        mov   byte ptr pdescr+4,DL; (53)базу, биты 16...23
        mov   word ptr pdescr,gdt_size-1; (54)Граница GDT
        lgdt  pdescr        ; (55)Загрузим регистр GDTR
;Подготовимся к возврату из защищенного режима в реальный
        mov   AX,40h        ; (56)Настроим ES на область
        mov   ES,AX         ; (57)данных BIOS
        mov   word ptr ES:[67h],offset return; (58)Смещение точки
                            ;возврата
        mov   ES:[69h],CS   ; (59)Сегмент точки возврата
;Подготовимся к переходу в защищенный режим
        cli                 ; (60)Запрет аппаратных прерываний
        mov   AL,8Fh        ; (61)Запрет NMI (80h) и выборка
                            ;байта состояния отключения 0Fh
        out   70h,AL        ; (62)Порт КМОП-микросхемы
        jmp   $+2           ; (63)Задержка
        mov   AL,0Ah        ; (64)Установим режим восстановле-
        out   71h,AL        ; (65)ния после сброса процессора
;Переходим в защищенный режим
```

```
        smsw    AX              ;(66)Получим слово состояния машины
        or      AX,1            ;(67)Установим бит PE
        lmsw    AX              ;(68)Запишем назад слово состояния
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
;и заодно очищаем очередь команд
        db      0EAh            ;(69)Код команды far jmp
        dw      offset continue;(70)Смещение
        dw      16              ;(71)Селектор сегмента команд
continue:                       ;(72)
;Делаем адресуемыми данные
        mov     AX,8            ;(73)Селектор сегмента данных
        mov     DS,AX           ;(74)
;Делаем адресуемым стек
        mov     AX,24           ;(75)Селектор сегмента стека
        mov     SS,AX           ;(76)
;Инициализируем ES и выводим символы
        mov     AX,32           ;(77)Селектор сегмента видеобуфера
        mov     ES,AX           ;(78)
        mov     BX,800          ;(79)Начальное смещение на экране
        mov     CX,640          ;(80)Число выводимых символов
        mov     AX,word ptr sym;(81)Начальный символ с атрибутом
screen: mov     ES:[BX],AX      ;(82)Вывод в видеобуфер
        add     BX,2            ;(83)Сместимся в видеобуфере
        inc     AX              ;(84)Следующий символ
        loop    screen          ;(85)Цикл вывода на экран
;Вернемся в реальный режим
        mov     real_sp,SP      ;(86)Сохраним SP
        mov     AL,0FEh         ;(87)Команда сброса процессора
        out     64h,AL          ;(88)в порт 64h
        hlt                     ;(89)Останов процессора до окончания
                                ;сброса
;Теперь процессор снова работает в реальном режиме
;Восстановим операционную среду реального режима
return: mov     AX,data         ;(90)Восстановим адресуемость
        mov     DS,AX           ;(91)данных
        mov     SP,real_sp      ;(92)Восстановим
        mov     AX,stk          ;(93)адресуемость
        mov     SS,AX           ;(94)стека
;Разрешим аппаратные и немаскируемые прерывания
        sti                     ;(95)Разрешение прерываний
        mov     AL,0            ;(96)Сброс бита 7 в порте CMOS-
        out     70h,AL          ;(97)разрешение NMI
;Проверим выполнение функций DOS после возврата в реальный режим
        mov     AH,09h          ;(98)Функция вывода на экран строки
        mov     DX,offset mes;(99)Адрес строки
        int     21h             ;(100)Вызов DOS
        mov     AX,4C00h        ;(101)Завершим программу обычным
        int     21h             ;(102)образом
main    endp                    ;(103)Конец главной процедуру
code_size=$-main                ;(104)Размер сегмента команд
text    ends                    ;(105)Конец сегмента команд
stk     segment stack 'stack';(106)Начало сегмента стека
        db      256 dup ('^');(107)
stk     ends                    ;(108)Конец сегмента стека
        end main                ;(109)Конец программы
```

Микропроцессоры 80386 и 486 отличаются от предыдущих расширенным набором команд, часть которых относится к привилегированным. Для того, чтобы разрешить транслятору обрабатывать эти команды, в текст программы необходимо включить директиву ассемблера .386P.

Программа начинается с описания структуры дескриптора сегмента. В отличие от реального режима, в котором сегменты определяются их базовыми адресами, задаваемыми программистом в явной форме, в защищенном режиме для каждого сегмента программы (т.е. для сегментов команд, данных и стека) в программе должен быть определен дескриптор - 8-байтовое поле, в котором в определенном формате записываются базовый адрес сегмента, его длина и некоторые другие характеристики. (рис. 61.1).

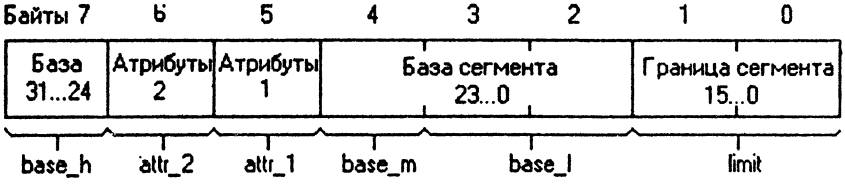


Рис. 61.1. Дескриптор сегмента.

Теперь для обращения к требуемому сегменту программист заносит в сегментный регистр не сегментный адрес, а так называемый селектор (рис. 61.2), в состав которого входит номер (индекс) соответствующего сегменту дескриптора. Процессор по этому номеру находит нужный дескриптор, извлекает из него базовый адрес сегмента и, прибавляя к нему указанное в конкретной команде смещение (относительный адрес), формирует адрес ячейки памяти. Индекс дескриптора (0, 1, 2 и т.д.) записывается в селектор начиная с бита 3, что эквивалентно умножению его на 8. Таким образом, можно считать, что селекторы последовательных дескрипторов представляют собой числа 0, 8, 16, 24 и т.д. (см. комментарии к предложениям 11...15).

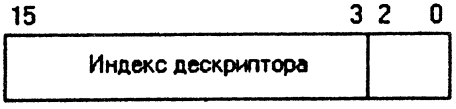


Рис. 61.2. Селектор дескриптора.

Структура descr предоставляет шаблон для дескрипторов сегментов, облегчающий их формирование. Сравнивая описание структуры descr в программе с рис. 61.1, нетрудно проследить их соответствие друг другу.

Рассмотрим (сначала вкратце) содержимое дескриптора. Граница (limit) сегмента представляет собой номер последнего байта сегмента.

Так, для сегмента размером 375 байт граница равна 374. Поле границы состоит из 20 бит и разбито на две части. Как видно из рис. 61.1, младшие 16 бит границы занимают байты 0 и 1 дескриптора, а старшие 4 бита входят в байт атрибутов 2, занимая в нем биты 0...3. Получается, что размер сегмента ограничен величиной 1 Мбайт. На самом деле это не так. Граница может указываться либо в байтах (и тогда, действительно, максимальный размер сегмента равен 1 Мбайт), либо в блоках по 4 Кбайт (и тогда размер сегмента может достигать 4 Гбайт). В каких единицах задается граница определяет самый старший бит байта 7 (атрибуты 2). Этот бит называется битом дробности (гранулярности). Если он равен 0, граница указывается в байтах; если 1 - в блоках по 4 Кбайт.

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

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

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

Поле базы, как и поле границы, разбито на 2 части: биты 0...23 занимают байты 2, 3 и 4 дескриптора, а биты 24...31 - байт 7. Для

удобства программного обращения в структуре descr база описывается тремя полями: младшим словом (base_l) и двумя байтами: средним (base_m) и старшим (base_h).

В байте атрибутов 1 задается ряд характеристик сегмента. Не вдаваясь пока в подробности этих характеристик, укажем, что в примере 61.1 используются сегменты двух типов: сегмент команд, для которого байт attr_1 должен иметь значение 98h, и сегмент данных (или стека) с кодом 92h.

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

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

Помимо единственной таблицы глобальных дескрипторов (она часто называется GDT от Global Descriptor Table) в памяти может находиться множество таблиц локальных дескрипторов (LDT от Local Descriptor Table). Разница между ними в том, что сегменты, описываемые глобальными дескрипторами, доступны всем задачам, выполняемым процессором, а к сегментам, описываемым локальными дескрипторами, может обращаться только та задача, в которой эти дескрипторы описаны. Поскольку пока мы имеем дело с однозадачным режимом, локальная таблица нам не нужна.

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

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

В дескрипторе gdt_data, описывающем сегмент данных программы, заполняется поле границы сегмента (фактическое значение размера сегмента data_size будет вычислено транслятором, см. предложение 22), а также байт атрибутов 1. Код 92h говорит о том, что это сегмент данных

с разрешением записи и чтения. База сегмента, т.е. физический адрес его начала, в явной форме в программе отсутствует, поэтому ее придется программно вычислить и занести в дескриптор уже на этапе выполнения.

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

Дескриптор gdt_stack сегмента стека имеет, как и любой сегмент данных, код атрибута 92h, что разрешает его чтение и запись, и явным образом заданную границу - 255 байт, что соответствует размеру стека. Базовый адрес сегмента стека так же придется вычислить на этапе выполнения программы.

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

Перед переходом в защищенный режим процессору надо будет сообщить физический адрес таблицы глобальных дескрипторов и ее размер (точнее, границу). Размер GDT определяется на этапе трансляции в предложении 16.

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

Сегмент команд text начинается, как и всегда, оператором segment, в котором указывается тип использования use16. Этот описатель объявляет, что в данном сегменте будут по умолчанию использоваться 16-битовые адреса и операнды. Если бы мы готовили нашу программу для работы под управлением операционной системы защищенного режима, реализующей все возможности микропроцессора, тип использования был бы use32. Однако наша программа будет запускаться под управлением DOS, которая работает в реальном режиме с 16-битовыми адресами и операндами. Указание описателя use16 не запрещает использовать в программе 32-битовые регистры, как это будет продемонстрировано в одном из последующих примеров.

Фактически вся программа примера 61.1, кроме ее завершающих строк, а также фрагмента, выполняемого в защищенном режиме, посвящена подготовке перехода в защищенный режим. Прежде всего надо завершить формирование дескрипторов сегментов программы, в которых остались незаполненными базовые адреса сегментов. Базовые (32-

битовые) адреса определяются путем умножения значений сегментных адресов на 16. После обычной инициализации сегментного регистра DS (предложения 27-28), которая позволит нам обращаться к полям данных программы (в реальном режиме!) выполняется очистка регистра DL (предложение 29) и сдвиг старших 4 битов регистра AX в регистр DX (фактически в 4 младших бита DL). Эта операция выполняется командой shld (shift left double, двойной сдвиг влево), входящей в систему команд процессоров, начиная с 80386. Команда сдвигает влево содержимое первого операнда (в нашем случае DX) на указанное константой (или содержимым регистра CL) число бит, причем младшие биты первого операнда заполняются старшими битами второго. Второй операнд, однако, не изменяется. Командой shl (предложение 31) содержимое регистра AX также сдвигается влево на 4 бита. При этом старшие 4 бита теряются, однако они уже находятся в регистре DL. Следующими тремя командами (предложения 32...34) содержимое AX отправляется в поле base_l дескриптора gdt_data, а содержимое DL - в поле base_m.

Аналогично вычисляются 32-битовые адреса сегментов команд и стека, помещаемые в дескрипторы gdt_code и gdt_stack.

Следующий этап подготовки к переходу в защищенный режим - загрузка в регистр процессора GDTR (Global Descriptor Table Register, регистр таблицы глобальных дескрипторов) информации о таблице глобальных дескрипторов. Эта информация включает в себя линейный базовый адрес таблицы и ее границу и размещается в 6 байтах поля данных, называемого иногда псевдодескриптором. Для загрузки GDTR предусмотрена специальная привилегированная команда lgdt (load global descriptor table, загрузка таблицы глобальных дескрипторов), которая требует указания в качестве операнда имени псевдодескриптора. Формат псевдодескриптора приведен на рис. 61.3.

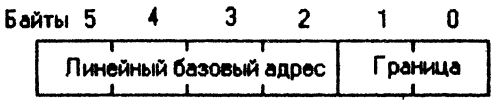


Рис. 61.3. Формат псевдодескриптора.

В нашем примере заполнение псевдодескриптора упрощается вследствие того, что таблица глобальных дескрипторов расположена в начале сегмента данных, и ее базовый адрес совпадает с базовым адресом всего сегмента, который уже был вычислен и помещен в дескриптор gdt_data. В предложениях 49...53 компоненты базового адреса переносятся из дескриптора в требуемые поля pdescr, а в предложении 54 заполняется поле границы. Команда lgdt загружает регистр GDTR и сообщает процессору о местонахождении и размере GDT.

В принципе теперь можно перейти в защищенный режим. Однако мы запускаем нашу программу под управлением DOS (а как еще мы

можем ее запустить?) и естественно завершить ее также обычным образом, чтобы не нарушить работоспособность системы. Но в защищенном режиме запрещены любые обращения к функциям DOS или BIOS. Причина этого совершенно очевидна - и DOS, и BIOS являются программами реального режима, в которых широко используется сегментная адресация реального режима, т.е. загрузка в сегментные регистры сегментных адресов. В защищенном же режиме в сегментные регистры загружаются не сегментные адреса, а селекторы. Кроме того, обращение к функциям DOS и BIOS осуществляется с помощью команд программного прерывания int с определенными номерами, а в защищенном режиме эти команды приведут к совершенно иным результатам. Таким образом, программу, работающую в защищенном режиме, нельзя завершить средствами DOS. Сначала ее надо вернуть в реальный режим.

Возврат в реальный режим можно осуществить сбросом процессора. Действия процессора после сброса определяются одной из ячеек КМОП-микросхемы - байтом состояния отключения, располагаемым по адресу Fh. В частности, если в этом байте записан код Ah, после сброса управление немедленно передается по адресу, который извлекается из двухсловной ячейки 40h:67h, расположенной в области данных BIOS. Таким образом, для подготовки возврата в реальный режим мы должны в ячейку 40h:67h записать адрес возврата, а в байт Fh КМОП-микросхемы занести код Ah. В предложениях 56...59 полный адрес точки возврата return заносится по адресу 40h:67h. Точка возврата может располагаться в любом месте программы

Еще одна важная операция, которую необходимо выполнить перед переходом в защищенный режим, заключается в запрете всех аппаратных прерываний. Дело в том, что в защищенном режиме процессор выполняет процедуру прерывания не так, как в реальном. При поступлении сигнала прерывания процессор не обращается к таблице векторов прерываний в первом килобайте памяти, как в реальном режиме, а извлекает адрес программы обработки прерывания из таблицы дескрипторов прерываний, построенной схоже с таблицей глобальных дескрипторов и располагаемой в программе пользователя (или в операционной системе). В примере 61.1 такой таблицы нет, и на время работы нашей программы прерывания придется запретить. Запрет всех аппаратных прерываний осуществляется командой cli (предложение 60). Однако желательно запретить еще и немаскируемые прерывания, которые поступают в процессор по отдельной линии (вход NMI микропроцессора) и не управляются битом IF регистра флагов. Немаскируемые прерывания обычно используются для обработки таких катастрофических событий, как сбой питания, ошибка памяти или ошибка четности на магистрали; в реальном режиме для них зарезервирован вектор 02h. Для запрета немаскируемых прерываний не предусмотрено никаких команд, однако

это можно сделать окольным способом, установив старший бит в адресном порте 70h КМОП-микросхемы.

В предложениях 61-62 в порт 70h засылается код 8Fh, который выбирает для записи байт Fh КМОП-микросхемы и одновременно запрещает немаскируемые прерывания установкой старшего бита. После небольшой задержки, необходимой для срабатывания микросхемы, в порт данных 71h посылается код Ah, определяющий режим восстановления (переход по адресу, извлекаемому из области данных BIOS).

В предложениях 66...68 осуществляется перевод процессора в защищенный режим. Это можно выполнить различными способами; в примере использованы команды smsw (store machine status word, запись слова состояния машины) и lmsw (load machine status word, загрузка слова состояния машины), которые можно использовать и в МП 286; в следующей статье будет рассмотрен способ, действующий только с МП 386 и 486. Переход в защищенный режим осуществляется установкой в 1 бита 0 слова состояния машины. Поскольку остальные биты этого слова нам могут быть не известны, сначала мы читаем в регистр AX слово состояния машины, затем устанавливаем в нем бит 0 и, наконец, записываем модифицированное слово состояния назад в процессор. Все последующие команды выполняются уже в защищенном режиме.

Хотя защищенный режим установлен, однако действия по настройке системы еще не закончены. Действительно, во всех используемых в программе сегментных регистрах хранятся не селекторы дескрипторов сегментов, а базовые сегментные адреса, не имеющие смысла в защищенном режиме. Между прочим, отсюда можно сделать вывод, что после перехода в защищенный режим программа не должна работать, так как в регистре CS пока еще нет селектора сегмента команд, и процессор не может обращаться к этому сегменту. В действительности это не совсем так.

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

В реальном режиме теневые регистры заполняются не из таблицы дескрипторов, а непосредственно самим процессором. В частности, процессор заполняет поле базы каждого теневого регистра линейным

базовым адресом сегмента, полученным путем умножения на 16 содержимого сегментного регистра, как это и положено в реальном режиме. Поэтому после перехода в защищенный режим в теневых регистрах находятся правильные линейные базовые адреса, и программа будет выполняться правильно, хотя с точки зрения правил адресации защищенного режима содержимое сегментных регистров лишено смысла.

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

Загрузить селекторы в сегментные регистры DS, SS и ES не представляет труда (предложения 73...78). Но как загрузить селектор в программно недоступный регистр CS? Для этого можно воспользоваться искусственно сконструированной командой дальнего перехода, которая, как известно, приводит к смене содержимого и IP, и CS. Предложения 69...71 демонстрируют эту методику. В реальном режиме мы поместили бы во второе слово адреса сегментный адрес сегмента команд, в защищенном же мы записываем в него селектор этого сегмента (число 16).

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

Следующий фрагмент примера 61.1 (предложения 77...85) является чисто иллюстративным. В нем инициализируется (по правилам защищенного режима!) сегментный регистр ES и в видеобуфер экрана выводится некоторое количество цветных символов, чем подтверждается правильное функционирование программы в защищенном режиме.

Как уже отмечалось выше, для того, чтобы не нарушить работоспособность DOS, процессор следует вернуть в реальный режим, после

чего можно будет завершить программу обычным образом. Перейти в реальный режим можно разными способами; в нашем примере сброс процессора выполняется засылкой команды FEh в порт 64h контроллера клавиатуры (предложения 87-88). Эта команда возбуждает сигнал на одном из выводов контроллера клавиатуры, который в конечном счете приводит к появлению сигнала сброса на выводе RESET микропроцессора. Перед выполнением сброса текущее содержимое SP сохраняется в ячейке real_sp, чтобы после перехода в реальный режим можно было восстановить состояние стека.

После сброса процессор начинает работать в реальном режиме, причем управление передается программам BIOS. BIOS анализирует содержимое байта состояния отключения (Fh) КМОП-микросхемы и, поскольку мы записали туда код Ah, осуществляет передачу управления по адресу, хранящемуся в ячейке 40h:67h области данных BIOS. В нашем случае переход осуществляется на метку return.

Команда hlt (halt, останов) позволяет организовать ожидание сброса процессора, который выполняется не мгновенно. Вместо команды hlt можно было использовать бесконечный цикл:

```
stop:   jmp   stop
```

Если команду ожидания опустить, процессор до своего останова успеет выполнить несколько следующих команд, после чего все-таки перейдет на метку return.

Передача управления на метку return осуществляется программами BIOS, которые, естественно, используют регистры процессора. В частности, регистры SS:SP и DS указывают на поля данных BIOS, и их следует инициализировать заново, что и выполняется в предложениях 90...94. Заметим, что сохранение и восстановление указателя стека SP не является обязательным, во всяком случае, в нашем примере, где работа программы до перехода в защищенный режим и после возврата из него протекает независимо. С таким же успехом можно после возврата в реальный режим инициализировать SP заново, выполнив команду

```
        mov   SP,256
```

(в предположении, что стек имеет размер 256 байт).

Для восстановления работоспособности системы следует разрешить маскируемые (предложение 95) и немаскируемые прерывания (предложения 96-97), после чего программа может продолжаться уже в реальном режиме. В рассматриваемом примере для проверки работоспособности системы на экран выводится некоторый текст с помощью функции DOS 09h. Для наглядности в него включены Esc-последовательности смены цвета символов, поэтому программу следует выполнять при установленном драйвере ANSI.SYS.

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

У рассмотренной программы имеется серьезный недостаток - полное отсутствие средств отладки. Для отладки программ защищенного режима используется механизм прерываний и исключений, в нашей же программе этот механизм не активизирован. Поэтому всякие неполадки при работе в защищенном режиме, которые с помощью указанного механизма можно было бы обнаружить и проанализировать, в данном случае будут приводить к сбросу процессора. Однако после сброса программа, скорее всего, будет работать правильно: на экран будет выведена запланированная строка и программа завершится с передачей управления DOS. Таким образом, критерием программных ошибок защищенного режима может служить правильная в целом работа программы при отсутствии на экране цветных символов, которые должны выводиться в защищенном режиме. В дальнейших статьях будут рассмотрены некоторые средства отладки программ защищенного режима.

Статья 62

32-разрядные операнды и другие усовершенствования

Модифицируем пример 61.1 из предыдущей статьи, введя в него использование 32-разрядных операндов и предусмотрев более изящный способ смены режимов микропроцессора. Общий ход программы и результат ее работы не отличается от примера 61.1 - переход в защищенный режим, вывод в видеобуфер цветных символов, возврат в реальный режим и вывод на экран сообщения.

Пример 62.1. Использование 32-битовых операндов и управляющего регистра CR0.

```
.386P                           ;(1)Разрешение команд МП 386 и 486
;Структура для описания дескрипторов сегментов
descr   struc                   ;(2)
limit   dw      0               ;(3)Граница (биты 0...15)
base_l  dw      0               ;(4)База, биты 0...15
base_m  db      0               ;(5)База, биты 16...23
attr_1  db      0               ;(6)Байт атрибутов 1
attr_2  db      0               ;(7)Граница (биты 16...19) и атрибуты 2
base_h  db      0               ;(8)База, биты 24...31
```

```
descr   ends                    ;(9)
data    segment                 ;(10)Начало сегмента данных
;Таблица глобальных дескрипторов GDT
gdt_null descr <0,0,0,0,0,0>;(11);Нулевой дескриптор
gdt_data descr <data_size-1,0,0,92h>;(12)Сел-р 8, сегмент данных
gdt_code descr <code_size-1,,,98h>;(13)Селектор 16, сегмент команд
gdt_stack descr <255,0,0,92h>;(14)Селектор 24, сегмент стека
gdt_screen descr <4095,8000h,0Bh,92h>;(15)Селектор 32, видеобуфер
gdt_size=$-gdt_null             ;(16)Размер GDT
;Поля данных программы
pdescr  dq      0               ;(17)Псевдодескриптор
sym     db      1               ;(18)Символ для вывода на экран
attr    db      1Eh             ;(19)Его атрибут
mes db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$';(20)
data_size=$-gdt_null            ;(21)Размер сегмента данных
data    ends                    ;(22)Конец сегмента данных
text    segment 'code' use16;(23)Укажем 16-разрядный режим
        assume CS:text,DS:data;(24)
main    proc                    ;(25)
        xor     EAX,EAX         ;(26)Очистим EAX
        mov     AX,data         ;(27)Загрузим в DS сегментный
        mov     DS,AX           ;(28)адрес сегмента данных
;Вычислим 32-битовый линейный адрес сегмента данных и загрузим его
;в дескриптор сегмента данных в GDT.
        shl     EAX,4           ;(29)В EAX линейный базовый адрес
        mov     EBP,EAX         ;(30)Сохраним его в EBP
        mov     BX,offset gdt_data;(31)В BX адрес дескриптора
        mov     [BX].base_1,AX;(32)Загрузим младшую часть базы
        rol     EAX,16          ;(33)Обмен старшей и младшей половин EAX
        mov     [BX].base_m,AL;(34)Загрузим среднюю часть базы
;Аналогично для линеймого адреса сегмента команд
        xor     EAX,EAX         ;(35)Очистим EAX
        mov     AX,CS           ;(36)Адрес сегмента команд
        shl     EAX,4           ;(37)Умножим на 16
        mov     BX,offset gdt_code;(38)Адрес дескриптора
        mov     [BX].base_1,AX;(39)Загрузим младшую часть базы
        rol     EAX,16          ;(40)Обмен половин EAX
        mov     [BX].base_m,AL;(41)Загрузим среднюю часть базы
;Аналогично для линейного адреса сегмента стека
        xor     EAX,EAX         ;(42)
        mov     AX,SS           ;(43)
        shl     EAX,4           ;(44)
        mov     BX,offset gdt_stack;(45)
        mov     [BX].base_1,AX;(46)
        rol     EAX,16          ;(47)
        mov     [BX].base_m,AL;(48)
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        mov     dword ptr pdescr+2,EBP;(49)База GDT, биты 0...31
        mov     word ptr pdescr,gdt_size-1;(50)Граница GDT
        lgdt    pdescr          ;(51)Загрузим регистр GDTR
;Подготовимся к переходу в защищенный режим
        cli                     ;(52)Запрет аппаратных прерываний
        mov     AL,80h          ;(53)Запрет NMI
        out     70h,AL          ;(54)Порт КМОП-микросхемы
;Переходим в защищенный режим
        mov     EAX,CR0         ;(55)Получим содержимое CR0
        or      EAX,1           ;(56)Установим бит PE
```

```
        mov     CR0,EAX     ;(57)Запишем назад в CR0
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
;и заодно очищаем очередь команд
        db      0EAh        ;(58)Код команды far jmp
        dw      offset continue;(59)Смещение
        dw      16          ;(60)Селектор сегмента команд
continue:                   ;(61)
;Делаем адресуемыми данные
        mov     AX,8        ;(62)Селектор сегмента данных
        mov     DS,AX       ;(63)
;Делаем адресуемым стек
        mov     AX,24       ;(64)Селектор сегмента стека
        mov     SS,AX       ;(65)
;Инициализируем ES и выводим символы
        mov     AX,32       ;(66)Селектор сегмента видеобуфера
        mov     ES,AX       ;(67)
        mov     BX,800      ;(68)Начальное смещение на экране
        mov     CX,640      ;(69)Число выводимых символов
        mov     AX,word ptr sym;(70)Начальный символ с атрибутом
screen: mov     ES:[BX],AX ;(71)Вывод в видеобуфер
        add     BX,2        ;(72)Сместимся в видеобуфере
        inc     AX          ;(73)Следующий символ
        loop    screen      ;(74)Цикл вывода на экран
;Подготовим переход в реальный режим
;Сформируем и загрузим дескрипторы для реального режима
        mov     gdt_data.limit,0FFFFh;(75)Граница сегмента данных
        mov     gdt_code.limit,0FFFFh;(76)Граница сегмента команд
        mov     gdt_stack.limit,0FFFFh;(77)Граница сегмента стека
        mov     gdt_screen.limit,0FFFFh;(78)Граница дополнительного
                            ;сегмента
        mov     AX,8        ;(79)Загрузим теневой регистр
        mov     DS,AX       ;(80)сегмента данных
        mov     AX,24       ;(81)Загрузим теневой регистр
        mov     SS,AX       ;(82)стека
        mov     AX,32       ;(83)Загрузим теневой регистр
        mov     ES,AX       ;(84)дополнительного сегмента
;Выполним дальний переход для того, чтобы заново загрузить
;селектор в регистр CS и модифицировать его теневой регистр
        db      0EAh        ;(85)Командой дальнего перехода
        dw      offset go   ;(86)загрузим теневой регистр
        dw      16          ;(87)сегмента команд
;Переключим режим процессора
go:     mov     EAX,CR0     ;(88)Получим содержимое CR0
        and     EAX,0FFFFFFFEh;(89)Сбросим бит PE
        mov     CR0,EAX     ;(90)Запишем назад в CR0
        db      0EAh        ;(91)Код команда far jmp
        dw      offset return;(92)Смещение
        dw      text        ;(93)Сегмент
;Теперь процессор снова работает в реальном режиме
;Восстановим операционную среду реального режима
return: mov     AX,data     ;(94)Восстановим
        mov     DS,AX       ;(95)адресуемость данных
        mov     AX,stk      ;(96)Восстановим
        mov     SS,AX       ;(97)адресуемость стека
;Разрешим аппаратные и немаскируемые прерывания
        sti                 ;(98)Разрешение прерываний
```

```
        mov     AL,0            ;(99)Сброс бита 7 в порте CMOS -
        out     70h,AL          ;(100) - разрешение NMI
;Проверим выполнение функций DOS после возврата в реальный режим
        mov     AH,09h          ;(101)
        mov     DX,offset mes;(102)
        int     21h             ;(103)
        mov     AX,4C00h        ;(104)Завершим программу
        int     21h             ;(105);обычным образом
main    endp                    ;(106)
code_size=$-main                ;(107)Размер сегмента команд
text    ends                    ;(108)
stk     segment stack 'stack';(109)
        db      256 dup ('^');(110)
stk     ends                    ;(111)
        end main                ;(112)
```

Сегмент данных программы 62.1 отличается от примера 61.1 лишь тем, что из него удалена ячейка real_sp. В сегменте команд при вычислении линейных 32-битовых адресов естественно воспользоваться 32-битовыми регистрами. Обратите внимание на то, что сегмент команд по-прежнему объявлен с описателем use16, определяющим, что в данном сегменте будут по умолчанию использоваться 16-битовые адреса и операнды. Это естественно, так как мы будем запускать программу под управлением системы реального режима MS-DOS. В то же время описатель use16 не препятствует использованию 32-битовых регистров.

В предложении 26 очищается регистр EAX, после чего с помощью его младшей половины AX обычным образом инициализируется регистр DS для работы в реальном режиме.

32-битовые линейные адреса вычисляются отлично от примера 61.1. Содержимое EAX (а в нем находится сегментный адрес сегмента данных) командой shl сдвигается влево на 4 бита, образуя линейный 32-битовый адрес. Поскольку этот адрес будет использоваться и в последующих фрагментах программы, он запоминается в 32-битовом регистре EBP (или любом другом свободном регистре общего назначения). В BX загружается адрес дескриптора данных, после чего в дескриптор заносится младшая половина линейного адреса из регистра AX. Поскольку к старшей половине регистра EAX (где нас интересуют разряды 17...24) обратиться невозможно, над всем содержимым EAX с помощью команды rol (rotate left, циклический сдвиг влево) выполняется циклический сдвиг на 16 бит, в результате которого младшая и старшая половины EAX меняются местами.

Команда rol осуществляет сдвиг влево всех битов операнда на число разрядов, определяемое вторым операндом, который может находиться в CL или представлять собой непосредственное значение (как в примере 62.1). При каждом сдвиге на 1 разряд старший бит операнда загружается в его младший разряд и, кроме того, поступает в флаг CF. Таким

образом, содержимое CF после завершения команды всегда совпадает с младшим битом операнда.

После сдвига содержимое AL (где теперь находятся биты 17...24 линейного адреса) заносится в поле base_m дескриптора. Аналогично вычисляются линейные адреса сегмента команд (предложения 35...41) и сегмента стека (предложения 42...48).

После подготовки псевдодескриптора pdescr, на что теперь требуется всего две команды, осуществляется загрузка регистра таблицы глобальных дескрипторов GDTR (предложения 49...51).

Смена режимов в настоящем примере осуществляется с помощью управляющего регистра процессора CR0. Всего в МП 386 и 486 имеется 4 программно-адресуемых управляющих регистра с мнемоническими именами CR0, CR1, CR2 и CR3. Регистр CR1 зарезервирован, регистры CR2 и CR3 управляют страничным преобразованием, которое у нас выключено, а регистр CR0 содержит целый ряд управляющих битов, из которых нас сейчас будут интересовать только биты 31 (разрешение страничного преобразования) и 0 (включение защиты). При включении процессора оба эти бита сбрасываются, и в процессоре устанавливается реальный режим с выключенным страничным преобразованием. Установка в 1 младшего бита CR0 переводит процессор в защищенный режим, сброс этого бита возвращает его в режим реальных адресов. Стоит упомянуть, что младшая половина регистра CR0 совпадает со словом состояния машины МП 286, поэтому команды чтения и записи CR0 схожи по своему результату с командами smsw и lmsw, которые ради совместимости сохранены и в МП 386 и 486.

Таким образом, теперь нам не надо для возврата в реальный режим выполнять сброс процессора и, следовательно, отпадает необходимость настраивать байт состояния отключения в КМОП-микросхеме, а также ячейки с адресом возврата в области данных BIOS. Единственное, что надо сделать для перехода в защищенный режим - это запретить все прерывания и установить в 1 бит 0 регистра CR0.

В предложении 52 программы запрещаются аппаратные прерывания, в предложениях 53-54 засылкой в порт 70h кода 80h запрещаются немаскируемые прерывания.

Обращение к регистрам управления осуществляется исключительно с помощью команды mov, причем в качестве операнда должен быть указан регистр общего назначения. Для модификации CR0 его содержимое сначала считывается в EAX, там с помощью команды or устанавливается младший бит, после чего второй командой mov новое значение загружается в CR0 (предложения 55...57). Процессор переходит в защищенный режим.

Дальнейшие (обязательные) действия в защищенном режиме по настройке регистров CS, DS и SS выполняются в точности так же, как и в

примере 61.1. Не отличается и "содержательная" часть программы, в которой (в защищенном режиме) на экран выводятся цветные символы.

Возврат в реальный режим в примере 61.1 осуществлялся путем генерации сигнала сброса процессора RESET. Здесь использован более "мягкий" способ управления через регистр CR0. Как уже отмечалось, режим процессора определяется состоянием бита 0 этого регистра. Если бит равен 1, процессор работает в защищенном режиме, если 0 - в реальном. Однако для корректного возврата в реальный режим надо выполнить некоторые подготовительные операции. Рассмотрение этих операций позволит нам глубже вникнуть в различия реального и защищенного режимов.

При работе в защищенном режиме в дескрипторах сегментов записаны, среди прочего, их линейные адреса и границы. Процессор при выполнении команды с адресацией к тому или иному сегменту сравнивает полученный им относительный адрес с границей сегмента и, если команда пытается адресоваться за пределами сегмента, формирует прерывание (исключение) нарушения общей защиты. Если в программе предусмотрена обработка исключений, такую ситуацию можно обнаружить и как-то исправить. Таким образом, в защищенном режиме программа не может выйти за пределы объявленных ею сегментов, а также не может выполнить действия, запрещенные атрибутами сегмента. Так, если сегмент объявлен исполняемым (код атрибута 1 98h), то данные из этого сегмента нельзя читать или модифицировать; если атрибут сегмента равен 92h, то в таком сегменте не может быть исполняемых команд, на зато данные можно как читать, так и модифицировать. Указав для какого-то сегмента код атрибута 90h, мы получим сегмент с запрещением записи. При попытке записи в этот сегмент процессор сформирует исключение общей защиты.

Как уже отмечалось, дескрипторы сегментов хранятся в процессе выполнения программы в теневых регистрах, которые загружаются автоматически при записи в сегментный регистр селектора.

При работе в реальном режиме некоторые поля теневых регистров должны быть заполнены вполне определенным образом. Так, для сегментов данных и стека (адресуемых через сегментные регистры DS, ES и SS), должны быть установлены следующие характеристики:

граница=FFFFh;

бит дробности=0;

доступ для записи разрешен.

Сегмент команд должен быть объявлен исполняемым с той же границей FFFFh.

Приведенный перечень ограничений не полон; мы рассматриваем здесь только описанные ранее атрибуты сегментов. Следует также заметить, что в МП 386 и 486 не два сегментных регистра данных, а четыре:

DS, ES, GS и FS (соответственно, и четыре теневых регистра для хранения их дескрипторов). Отмеченные выше ограничения относятся к сегментам, адресуемым через любой из этих регистров. Наконец, стоит подчеркнуть, что границы всех сегментов должны быть точно равны FFFFh; любое другое число, например, FFFEh, "не устроит" реальный режим.

Если мы просто перейдем в реальный режим сбросом бита 0 в регистре CR0, то в теневых регистрах останутся дескрипторы защищенного режима и при первом же обращении к любому сегменту программы возникнет исключение общей защиты, так как ни один из наших сегментов не имеет границы, равной FFFFh. Поскольку мы не обрабатываем исключения, произойдет сброс процессора и перезагрузка компьютера, так как в этом варианте программы мы не настроили байт состояния отключения и ячейки области данных BIOS. Таким образом, перед переходом в реальный режим необходимо исправить дескрипторы всех наших сегментов: команд, данных, стека и видеобуфера. К сегментным регистрам FS и GS мы не обращались, и о них можно не заботиться.

В предложениях 75...78 в поля границ всех четырех дескрипторов заносится FFFFh, а в предложениях 79...84 выполняется загрузка селекторов в сегментные регистры, что приводит к перезаписи содержимого теневых регистров. Сегментный регистр CS программно недоступен, поэтому его загрузку придется выполнить с помощью искусственно сформированной команды дальнего перехода (предложения 85...87).

Настроив все использовавшиеся в программе сегментные регистры, можно сбросить бит 0 в CR0 (предложения 88...90). После перехода в реальный режим нам придется еще раз выполнить команду дальнего перехода, чтобы очистить очередь команд в блоке предвыборки и загрузить в регистр CS вместо хранящегося там селектора обычный сегментный адрес регистра команд (предложения 91...93).

Теперь процессор снова работает в реальном режиме, причем, хотя в сегментных регистрах DS, ES и SS остались незаконные для реального режима селекторы, программа будет какое-то время выполняться правильно, так как в теневых регистрах находятся правильные линейные адреса (оставшиеся от защищенного режима) и законные для реального режима границы (загруженные туда нами в предложениях 75...84). Однако если в программе встретятся команды сохранения и восстановления содержимого сегментных регистров, например

```
push   DS
...
...
pop    DS
```

выполнение программы будет нарушено, так как команда pop DS загрузит в DS не сегментный адрес реального режима, а селектор, т.е. число

8 в нашем случае. Это число будет рассматриваться процессором, как сегментный адрес, и дальнейшие обращения к полям данных приведут к адресации начиная с физического адреса 80h, что, конечно, лишено смысла. Даже если в нашей программе нет строк сохранения и восстановления сегментных регистров, они неминуемо встретятся, как только произойдет переход в DOS по команде int 21h, так как диспетчер DOS сохраняет, а затем восстанавливает все регистры задачи, в том числе и сегментные. Поэтому после перехода в реальный режим необходимо загрузить в используемые далее сегментные регистры соответствующие сегментные адреса, что и выполняется для регистров DS и SS в предложениях 94...97.

В рассматриваемом варианте программы нет необходимости сохранять кадр стека, так как содержимое SP, в отличие от предыдущего примера, не разрушается при переходе в реальный режим с помощью регистра CR0, а регистр SS мы инициализируем заново.

Теперь, наконец, возврат в реальный режим завершен, и оставшаяся часть программы не отличается от примера 61.1.

Статья 63

Исключения

Наши первые программы защищенного режима, рассмотренные в предыдущих статьях, работали в условиях запрещенных аппаратных прерываний - как внешних, так и внутренних. К программным прерываниям, реализуемым с помощью команды int, мы также не обращались. Все эти предосторожности были необходимы потому, что механизм обработки прерываний в защищенном режиме сильно отличается от механизма реального режима. Если бы мы, не активизируя механизм обработки прерываний защищенного режима, перешли в защищенный режим при разрешенных прерываниях, первое же аппаратное прерывание (например, от таймера) привело бы к отключению процессора. То же произошло бы при выполнении процессором команды int с любым номером. Между прочим, попытка проверить это утверждение с помощью примера 61.1 не приведет к успеху. Действительно, в этой программе возврат в реальный режим осуществляется как раз с помощью сброса процессора, причем мы заранее предусмотрительно настроили

КМОП-микросхему и ячейки области данных BIOS, чтобы после отключения процессор вернулся в нашу программу. Поэтому в такой программе нельзя обнаружить ошибки защищенного режима, приводящие к отключению процессора.

Иное дело программа статьи 62. Здесь возврат в реальный режим осуществляется программным способом, установкой бита 0 управляющего регистра CR0. Если ошибка в программе имеет своим следствием отключение процессора, то программа аварийно завершится и будет инициирована перезагрузка системы. Происходит это потому, что байт состояния отключения в КМОП-микросхеме, расположенный по адресу Fh, в исходном состоянии содержит код 0. Это значение кода приводит после сброса процессора (и после нажатия клавиш <Ctrl>/<Alt>/<Del>) к перезапуску системы. Включите в программу (в ту ее часть, которая выполняется в защищенном режиме) команду int с любым числовым аргументом (например, int 3 или int 21h). Программа аварийно завершится с перезапуском DOS.

В этой и последующих статьях будет рассмотрена система прерываний защищенного режима и предложены программы, обрабатывающие прерывания разного рода.

Как уже упоминалось в статье 25, в МП 8086 (и, соответственно, в реальном режиме микропроцессоров 80x86) предусмотрены прерывания трех видов: внутренние, возникающие в самом микропроцессоре, внешние, поступающие в процессор от внешних устройств компьютера через контроллеры прерываний, и программные, инициируемые командой int. В защищенном режиме также возможны прерывания всех трех типов, при этом число внешних прерываний, определяемое количеством контроллеров прерываний в компьютере, осталось, естественно, тем же, однако функции внутренних прерываний и их количество существенно расширены. Внутренние прерывания, называемые здесь исключениями, исключительными ситуациями или особыми случаями (все эти термины являются попыткой перевода слова exception), являются важнейшим элементом организации защищенного режима. Мы начнем знакомство с системой прерываний защищенного режима именно с исключений. Внешние аппаратные, а также программные прерывания будут рассмотрены в последующих статьях, хотя принцип обслуживания всех видов прерываний один, и многие рассматриваемые в настоящей статье детали будут относиться ко все трем видам прерываний.

Так же, как и в реальном режиме, все прерывания защищенного режима имеют свои номера, причем их общее количество не должно превышать 256. Под исключения отданы первые 32 номера (0...31), причем реально возникающие исключения в процессоре 486 имеют номера 0...17, а номера с 18 по 31 зарезервированы для будущих моделей процессоров. В МП 286 реальных исключений еще меньше.

В реальном режиме процессор при регистрации прерывания обращается к таблице векторов прерываний, находящейся всегда в самом начале памяти и содержащей двухсловные адреса программ обработки прерываний, обычно называемых обработчиками прерываний. В защищенном режиме аналогом таблицы векторов прерываний является таблица дескрипторов прерываний (IDT от Interrupt Descriptor Table), располагающаяся в операционной системе защищенного режима или в программе пользователя. Таблица IDT содержит дескрипторы обработчиков прерываний, в которые, в частности, входят их адреса. Для того, чтобы процессор мог обратиться к этой таблице, ее адрес следует загрузить в регистр IDTR (Interrupt Descriptor Table Register, регистр таблицы дескрипторов прерываний) в точности так же, как это делается с адресом таблицы глобальных дескрипторов GDT.

Если в таблице глобальных дескрипторов GDT собраны дескрипторы сегментов программы, то таблица дескрипторов прерываний IDT состоит из дескрипторов другого вида, которые называются шлюзами (вентилями). Шлюзы в свою очередь подразделяются на несколько типов, но пока мы не будем на этом останавливаться. Через шлюзы осуществляется доступ к обработчикам прерываний и исключений. Формат шлюза заметно отличается от формата дескриптора сегмента памяти, однако ясно, что в нем в какой-то форме должен присутствовать адрес обработчика соответствующего прерывания. Процессор, зарегистрировав прерывание или исключение, по его номеру извлекает из IDT шлюз, определяет адрес обработчика и передает ему управление. Обработчик должен заканчиваться командой iret, которая возвращает управление в прерванную программу.

Таким образом, в программе, обслуживающей исключения, следует сформировать таблицу дескрипторов прерываний IDT, поместить в нее адреса обработчиков исключений, предусмотренных в программе, загрузить адрес IDT в системный регистр процессора IDTR и перейти в защищенный режим. Пока процессор находится в защищенном режиме, он при возникновении исключений (или вообще прерываний) будет использовать таблицу дескрипторов прерываний; после возврата в реальный режим (но до разрешения прерываний) ее следует заменить таблицей векторов реального режима. Рассмотрим простейшую программу, в которой предусмотрены средства обработки исключений (хотя сама обработка сведена к минимуму). Для облегчения отладки программы и исследования ее работы в нее включены средства отладки в виде макрокоманды debug.

Пример 63.1. Исключения

```
include debug.mac              ;(1)Объявление файла с макросом
.386P                          ;(2)
;Структура для описания дескрипторов сегментов
```

```
descr   struc               ;(3)
        ...                 ;См. предыдущие примеры
descr   ends                ;(4)
;Структура для описания дескрипторов прерываний (шлюзов ловушек)
trap    struc               ;(5)
offs_l  dw      0           ;(6)Смещение обработчика, биты 0...15
sel     dw      16          ;(7)Селектор сегмента команд
rsrv    db      0           ;(8)Зарезервировано
attr    db      8Fh         ;(9)Атрибуты
offs_h  dw      0           ;(10)Смещение обработчика, биты 16...31
trap    ends                ;(11)
data    segment use16       ;(12)
;Таблица глобальных дескрипторов GDT
gdt_null descr <0,0,0,0,0,0>;(13)Нулевой дескриптор
gdt_data descr <data_size-1,0,0,92h>;(14)Сел-р 8, сегмент данных
gdt_code descr <code_size-1,,,98h>;(15)Селектор 16, сегмент команд
gdt_stack descr <255,0,0,92h>;(16)Селектор 24, сегмент стека
gdt_screen descr <4095,8000h,0Bh,92h>;(17)Селектор 32, видеобуфер
gdt_size=$-gdt_null         ;(18)Размер GDT
;Таблица дескрипторов прерываний (исключениий) IDT
idt     label  word         ;(19)Начало таблицы IDT
;Дескрипторы исключений
exc0    trap   <exc_0>      ;(20)Дескриптор исключения 0
exc1    trap   <exc_1>      ;(21)Дескриптор исключения 1
exc2    trap   <exc_2>      ;(22)Дескриптор исключения 2
exc3    trap   <exc_3>      ;(23)Дескриптор исключения 3
exc4    trap   <exc_4>      ;(24)Дескриптор исключения 4
exc5    trap   <exc_5>      ;(25)Дескриптор исключения 5
exc6    trap   <exc_6>      ;(26)Дескриптор исключения 6
exc7    trap   <exc_7>      ;(27)Дескриптор исключения 7
exc8    trap   <exc_8>      ;(28)Дескриптор исключения 8
exc9    trap   <exc_9>      ;(29)Дескриптор исключения 9
exc0ah  trap   <exc_0ah>    ;(30)Дескриптор исключения 10
exc0bh  trap   <exc_0bh>    ;(31)Дескриптор исключения 11
exc0ch  trap   <exc_0ch>    ;(32)Дескриптор исключения 12
exc0dh  trap   <exc_0dh>    ;(33)Дескриптор исключения 13
exc0eh  trap   <exc_0eh>    ;(34)Дескриптор исключения 14
exc0fh  trap   <exc_0fh>    ;(35)Дескриптор исключения 15
exc10h  trap   <exc_10h>    ;(36)Дескриптор исключения 16
exc11h  trap   <exc_11h>    ;(37)Дескриптор исключения 17
idt_size=$-idt              ;(38)Размер таблицы IDT
;Поля данных программы
pdescr  dq     0            ;(39)Псевдодескриптор для команд
                            ;lgdt и lidt
mes db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$';(40)
tblhex  db '0123456789ABCDEF';(41)Таблица преобразования bin-hex
string  db '**** ****-**** ****-**** ****' ;(42)Шаблон для вывода
;           0    5    10   15   20   25      Позиции в шаблоне
len=$-string                ;(43)
home_sel dw    home         ;(44)Адрес возврата из исключения
         dw    10h          ;(45)Сегмент команд
data_size=$-gdt_null        ;(46)Размер сегмента данных
data    ends                ;(47)
text    segment 'code' use16;(48)
        assume CS:text,DS:data;(49)
begin   label  word         ;(50)Начало сегмента команд
exc_0   proc                ;(51)Обработчик исключения 0
```

```
        mov     AX,0            ;(52)Номер исключения для вывода на
                                ;экран
        jmp     dword ptr home_sel;(53)На выход
exc_0   endp                    ;(54)
exc_1   proc                    ;(55)Обработчик исключения 1
        mov     AX,1            ;(56)Номер исключения
        jmp     dword ptr home_sel;(57)На выход
exc_1   endp                    ;(58)
;Остальные обработчики (exc_2...exc_11h) выглядят аналогично
        ...
main    proc                    ;(59)
        xor     EAX,EAX         ;(60)Очистим EAX
        mov     AX,data         ;(61)Загрузим в DS сегментный
        mov     DS,AX           ;(62)адрес сегмента данных
;Вычислим 32-битовый линейный адрес сегмента данных и загрузим его
;в дескриптор сегмента данных в GDT
        ...                     ;См. предыдущий пример
;Аналогично для линейного адреса сегмента команд
        ...                     ;См. предыдущий пример
;Аналогично для линейного адреса сегмента стека
        ...                     ;См. предыдущий пример
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        mov     dword ptr pdescr+2,EBP;(63)База GDT, биты 0...31
        mov     word ptr pdescr,gdt_size-1;(64)Граница GDT
        lgdt    pdescr          ;(65)Загрузим регистр GDTR
;Подготовимся к переходу в защищенный режим
        cli                     ;(66)Запрет аппаратных прерываний
        mov     AL,80h          ;(67)Запрет NMI
        out     70h,AL          ;(68)Порт КМОП-микросхемы
;Таблица прерываний уже заполнена на этапе трансляции
;Загрузим регистр IDTR
        mov     word ptr pdescr,idt_size-1;(69)Граница IDT
        xor     EAX,EAX         ;(70)EAX=0
        mov     AX,offset idt;(71)Смещение IDT в сегменте данных
        add     EAX,EBP         ;(72)Прибавим линейный адрес сегмента
                                ;данных и получим
        mov     dword ptr pdescr+2,EAX;(73)линейный адрес IDT
        lidt    pdescr          ;(74)Загрузка IDTR
;Переходим в защищенный режим
        mov     EAX,CR0         ;(75)Получим содержимое CR0
        or      EAX,1           ;(76)Установим бит PE
        mov     CR0,EAX         ;(77)Запишем назад в CR0
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
        db      0EAh            ;(78)Код команды far jmp
        dw      offset continue;(79)Смещение
        dw      16              ;(80)Селектор сегмента команд
continue:
;Делаем адресуемыми данные
        mov     AX,8            ;(81)Селектор сегмента данных
        mov     DS,AX           ;(82)
;Делаем адресуемым стек
        mov     AX,24           ;(83)Селектор сегмента стека
        mov     SS,AX           ;(84)
;Инициализируем ES адресом видеобуфера
        mov     AX,32           ;(85)Селектор видеобуфера
        mov     ES,AX           ;(86)
```

```
;Организуем периодический вывод на экран символов
        mov    CX,300      ;(87)Число символов
        mov    BX,2720     ;(88)Начальная позиция на экране
        mov    DX,3001h    ;(89)начальный символ
xxxx:   push   CX          ;(90)Сохраним счетчик внешнего цикла
        mov    CX,0        ;(91)Повторение команды
zzzz:   loop   zzzz        ;(92)loop 65536 раз
        mov    ES:[BX],DX  ;(93)Вывод в видеобуфер
        inc    DL          ;(94)Инкремент символа
        add    BX,2        ;(95)Инкремент позиции на экране
        pop    CX          ;(96)Извлечем счетчик внешнего цикла
        loop   xxxx        ;(97)Цикл
        mov    AX,0FFFFh   ;(98)Диагностическое значение
home:   mov    SI,offset string;(99)
        debug              ;(100)
;Выведем на экран диагностическую строку
        mov    SI,offset string;(101)
        mov    CX,len      ;(102)
        mov    AH,74h      ;(103)
        mov    DI,1600     ;(104)
scr:    lodsb              ;(105)
        stosw              ;(106)
        loop   scr         ;(107)
;Подготовим переход в реальный режим
;Сформируем и загрузим дескрипторы для реального режима
        mov    gdt_data.limit,0FFFFh;(108)Граница сегмента данных
        mov    gdt_code.limit,0FFFFh;(109)Граница сегмента команд
        mov    gdt_stack.limit,0FFFFh;(110)Граница сегмента стека
        mov    gdt_screen.limit,0FFFFh;(111)Граница
                           ;дополнительного сегмента
        mov    AX,8        ;(112)Загрузим теневой регистр
        mov    DS,AX       ;(113)сегмента данных
        mov    AX,24       ;(114)Загрузим теневой регистр
        mov    SS,AX       ;(115)стека
        mov    AX,32       ;(116)Загрузим теневой регистр
        mov    ES,AX       ;(117)дополнительного сегмента
;Выполним дальний переход для того, чтобы заново загрузить
;селектор в регистр CS и модифицировать его теневой регистр
        db     0EAh        ;(118)Командой дальнего перехода
        dw     offset go   ;(119)загрузим теневой регистр
        dw     16          ;(120)сегмента команд
;Переключим режим процессора
go:     mov    EAX,CR0     ;(121)Получим содержимое CR0
        and    EAX,0FFFFFFFEh;(122)Сбросим бит защищенного режима
        mov    CR0,EAX     ;(123)Запишем назад в CR0
        db     0EAh        ;(124)Код команда far jmp
        dw     offset return;(125)Смещение
        dw     text        ;(126)Сегмент
;Теперь процессор снова работает в реальном режиме
;Восстановим операционную среду реального режима
return: mov    AX,data     ;(127)Восстановим
        mov    DS,AX       ;(128)адресуемость данных
        mov    AX,stk      ;(129)Восстановим
        mov    SS,AX       ;(130)адресуемость стека
;Восстановим состояние регистра IDTR реального режима
        mov    AX,3FFh     ;(131)Граница таблицы векторов(1Кбайт-1)
        mov    word ptr pdescr,AX;(132)
```

```
        mov     EAX,0       ;(133)Смещение таблицы векторов
        mov     dword ptr pdescr+2,EAX;(134)
        lidt    pdescr      ;(135)Загрузим псевдодескриптор в
                            ;регистр процессора IDTR
;Разрешим аппаратные и немаскируемые прерывания
        sti                 ;(136)Разрешение прерываний
        mov     AL,0        ;(137)Засылка константы с битом
        out     70h,AL      ;(138)7=0 в порт CMOS-разрешение NMI
;Проверим выполнение функций DOS после возврата в реальный режим
        mov     AH,09h      ;(139)
        mov     DX,offset mes;(140)
        int     21h         ;(141)
        mov     AX,4C00h    ;(142)Завершим программу
        int     21h         ;(143)
main    endp                ;(144)
code_size=$-begin           ;(145)Размер сегмента команд
text    ends                ;(146)
stk     segment stack 'stack';(147)
        db      256 dup ('^');(148)
stk     ends                ;(149)
        end main            ;(150)
```

Программа 63.1 имеет в целом ту же структуру, что и предыдущие. Она начинается с описания структуры глобального дескриптора, за которым следует структура дескриптора прерывания (предложения 5...11). В таблицу IDT могут, вообще говоря, входить шлюзы следующих типов:

шлюз задачи;

шлюз прерывания МП 286;

шлюз ловушки МП 286;

шлюз прерывания МП 386 или 486;

шлюз ловушки МП 386 или 486.

Через шлюзы задачи (task) осуществляется переключение на другие задачи в многозадачном режиме; через шлюзы прерываний (interrupt) обслуживаются аппаратные прерывания; шлюзы ловушек (trap) служат для обработки исключений и программных прерываний. В данной программе используются шлюзы ловушек.

Как видно из рис. 63.1, которому соответствует описание структуры trap, в шлюз входит 32-битовое смещение обработчика (байты 0-1 и 6-7), селектор сегмента команд, в котором находится программа данного обработчика, а также байт атрибутов, в котором указываются тип шлюза и некоторые другие его характеристики. В нашем случае (шлюз ловушки МП 486) в поле атрибутов должен быть код 8Fh.

Сегмент данных начинается, как и в предыдущих примерах, с таблицы глобальных дескрипторов. За ней описана таблица дескрипторов прерываний IDT (предложения 20...38), в которую пока включены лишь шлюзы исключений. Поскольку в МП 486 могут возникать исключения с номерами 0...17, в IDT включены 18 дескрипторов с мнемоническими именами exc0, exc1...exc11h. Отличаются эти дескрипторы только отно-

сительными адресами обработчиков исключений (exc_0...exc_11h). Остальные поля для всех дескрипторов одинаковы и описаны в структуре trap. Следует заметить, что в нашей программе дескрипторы прерываний полностью заполняются в процессе трансляции и в дальнейшем тексте программы обращаться к этим полям данных уже не нужно. В этом случае мнемонические имена, данные дескрипторам (exc0, exc1 и т.д.) теряют смысл. Их вполне можно удалить; в программу 63.1 они включены только ради наглядности. В константу idt_size транслятор помещает длину таблицы дескрипторов исключений, которая потребуется при загрузке регистра IDTR.

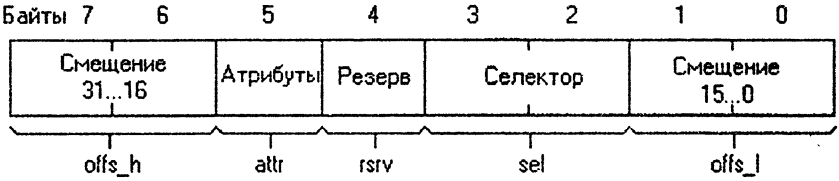


Рис. 63.1. Формат шлюза ловушки.

По сравнению в предыдущими примерами в сегмент данных вошли несколько новых полей. Символьные строки tblhex и string, а также константа len используются макрокомандой debug. В двухсловной ячейке home_sel содержится полный адрес (селектор:смещение) точки программы, куда будет передаваться управление из всех обработчиков исключений. Следует отметить, что это не очень изящный метод завершения обработки исключения. Как уже отмечалось, возврат из обработчика исключения обычно осуществляется командой iret, которая выполняет переход либо на ту команду, при выполнении которой возникло исключение, либо на следующую за ней. На этапе знакомства с системами прерываний и исключений такая процедура заметна усложнила бы отладку программы, поэтому в настоящем примере использован не вполне правильный, но более простой способ завершения обработчиков исключений командой дальнего перехода через поле home_sel.

Включение в сегмент данных дальнего адреса перехода в 16-битовый сегмент команд требует указания, что в данном сегменте будут использоваться по умолчанию 16-битовые адреса. Поэтому в предложении 12 указан тип использования use16.

Сегмент команд начинается с процедур обработчиков исключений. Все они имеют одинаковую структуру: в регистр AX засылается номер данного исключения, после чего осуществляется дальний переход в точку home_sel. В программе предусмотрено, что в случае возникновения того или иного исключения его номер выводится перед завершением

программы на экран. Это дает возможность диагностировать программу, а также исследовать условия возникновения исключений с различными номерами. В следующей статье будет показано, как можно повысить информативность обработчиков исключений.

Затем начинается процедура main. В ней после инициализации регистра DS заполняются дескрипторы сегментов, инициализируется регистр GDTR, запрещаются немаскируемые и маскируемые прерывания (предложения 59...68). Эта часть программы полностью повторяет соответствующий фрагмент предыдущего примера.

После запрещения прерываний можно загрузить регистр IDTR информацией о местонахождении и размере таблицы IDT. Для загрузки IDTR предусмотрена специальная привилегированная команда lidt (load interrupt descriptor table, загрузка таблицы дескрипторов прерываний), которая, как и команда lgdt, требует указания в качестве операнда имени псевдодескриптора. В предложении 69 в псевдодескриптор pdescr заносится граница IDT, в предложениях 70...72 определяется линейный базовый адрес IDT, в предложении 73 он заносится в псевдодескриптор и, наконец, в предложении 74 выполняется загрузка IDTR. Начиная с этого момента процессор будет обрабатывать все прерывания через нашу таблицу IDT. Однако процессор пока работает в реальном режиме, а формат IDT предполагает защищенный режим. Если бы мы не запретили прерывания перед загрузкой регистра IDTR, первое же прерывание (от таймера) привело бы к отключению процессора.

Переход в защищенный режим и инициализация сегментных регистров (предложения 75...86) выполняется, как и в предыдущем примере.

Фрагмент программы от предложения 87 до предложения 97 предназначен для вывода на экран в защищенном режиме некоторой последовательности цветных символов. В отличие от предыдущих примеров в цикл вывода на экран включена небольшая программная задержка (предложения 91-92). В последующих примерах эта задержка позволит нам более наглядно наблюдать взаимодействие основной программы и обработчика аппаратных прерываний от таймера.

Фрагмент программы, начинающийся с предложения 98, служит диагностическим целям. В регистр AX засылается произвольная константа (в примере FFFFh), в регистр SI загружается относительный адрес строки string и выполняется макрокоманда debug. Текст этой макрокоманды приведен ниже.

Отладочная макрокоманда debug.

```
debug macro
;При вызове: AX=преобразуемое число, SI=адрес строки с результатом
;преобразования (одно 4-разрядное 16-ричное число)
        push   AX          ;Сохраним
        push   BX          ;используемые
```

```
push   CX          ;регистры
push   DX
push   AX          ;Сохраним наше число в стеке
and    AX,0F000h   ;Выделим старшую четверку битов
shr    AX,12       ;Сдвинем в начало регистра
mov    BX,offset tblhex;BX=адрес таблицы трансляции
xlat               ;Команда табличной трансляции
mov    [SI],AL     ;Отправим символ в строку
pop    AX          ;Вернем в AX исходное число
push   AX          ;И отправим его обратно в стек
and    AX,0F00h    ;Выделим вторую четверку битов
shr    AX,8        ;Сдвинем в начало регистра
inc    SI          ;Инкремент в строке результата
xlat               ;Команда табличной трансляции
mov    [SI],AL     ;Отправим символ в строку
pop    AX          ;Вернем в AX исходное число
push   AX          ;И отправим его обратно в стек
and    AX,0F0h     ;Выделим третью четверку битов
shr    AX,4        ;Сдвинем в начало регистра
inc    SI          ;Инкремент в строке результата
xlat               ;Команда табличной трансляции
mov    [SI],AL     ;Отправим символ в строку
pop    AX          ;Вернем в AX исходное число
push   AX          ;И отправим его обратно в стек
and    AX,0Fh      ;Выделим младшую четверку битов
inc    SI          ;Инкремент в строке результата
xlat               ;Команда табличной трансляции
mov    [SI],AL     ;Отправим символ в строку
pop    AX          ;Восстановим стек
pop    DX          ;Восстановим
pop    CX          ;используемые
pop    BX          ;регистры
pop    AX          ;
endm
```

Макрокоманда debug преобразует двоичное число в регистре AX в символьную форму и помещает результат преобразования (четыре 16-ричные цифры) в строку, адрес которой находится в регистре SI. Все детали такого преобразования подробно обсуждались в статье 18.

Таким образом, тройка команд

```
mov    AX,0FFFFh
mov    SI,offset string
debug
```

приводит к помещению в строку по адресу string символьного представления содержимого регистра AX (в данном случае числа FFFF). Указанную комбинацию команд можно использовать в программе неоднократно. Пусть, например, в некоторой точке программы нас интересует стояние указателя стека SP и содержимое верхнего двойного слова стека. Если включить в программу предложения

```
mov    AX,SP       ;Содержимое SP в AX
mov    SI,offset string+5
```

```
debug
pop    EAX          ;Двойное слово из стека в EAX
push   EAX          ;И назад, чтобы не смещать стек
mov    SI,offset string+15;Адрес для младшего слова
debug
rol    EAX,16       ;Обменяем половины EAX
mov    SI,offset string+10;Адрес для старшего слова
                    ;(левее младшего на экране)
debug
```

то в строку string начиная с байта 5 будет помещено содержимое SP, а начиная с байта 10 - содержимое верхнего двойного слова стека (сначала старшие биты, затем младшие). При необходимости строку string можно удлинить и выводить на экран больше информации.

В предложениях 101...107 строка string пересылается в видеобуфер, начиная с заданной в регистре DI позиции. Атрибут строки содержится в регистре AH.

Программы преобразования и вывода на экран не требуют системных средств и могут выполнятся в защищенном режиме. Таким образом, с помощью макрокоманды debug можно динамически изучать содержимое регистров, стека и ячеек памяти.

Первая позиция строки string (4 байта) зарезервирована для вывода при нормальном выполнении программы контрольного числа FFFFh, а при наличии исключения - его номера. При возникновении любого исключения процессор находит в таблице дескрипторов прерываний дескриптор, порядковый номер которого совпадает с номером исключения (дескрипторы прерываний можно рассматривать, как векторы прерываний, но только не четырехбайтовые, как в реальном режиме, а восьмибайтовые) и извлекает из него селектор и смещение обработчика исключения. Адрес возврата сохраняется в стеке (подробности будут рассмотрены позже) и выполняется переход на программу обработчика (процедуры exc_0, exc_1 и т.д.). Обработчики исключений нашего примера построены единообразно: в регистр AX засылается номер исключения и выполняется дальний переход в точку home (предложение 99), где вызывается макрокоманда debug, после чего строка string выводится на экран.

После вывода диагностической строки осуществляется возврат в реальный режим точно так же, как и в предыдущем примере. Теневые регистры загружаются значениями, действительными для реального режима (предложения 108...120), с помощью регистра CR0 переключается режим и выполняется дальний переход для загрузки регистра CS сегментным адресом сегмента команд и очистки очереди предвввода (предложения 121...126).

В реальном режиме после восстановления значений DS и SS необходимо выполнить еще одну важную операцию: восстановить содержимое регистра IDTR, действительное для реального режима. Этот регистр

служит для определения характеристик таблицы векторов прерываний. При переходе в защищенный режим мы загрузили в него характеристики таблицы дескрипторов прерываний защищенного режима. Теперь в него следует загрузить характеристики таблицы векторов реального режима. Начальный адрес этой таблицы 0, а размер составляет 1 Кбайт (400h байт). Таким образом, граница равна 3FFh. Указанные значения помещаются в псевдодескриптор (предложения 131...134) и командой lidt осуществляется загрузка регистра IDTR. Переход в реальный режим закончен. После разрешения всех прерываний на экран выводится контрольное сообщение и программа завершается обычным образом.

Статья 64

Исследование исключений

Рассмотрим более подробно классификацию прерываний в процессоре 486.

Если процессор по каким-либо причинам не может выполнить очередную команду, возникает внутреннее событие, называемое исключением. В зависимости от причины возникновения, различают 18 видов исключений, которым присвоены номера векторов от 0 до 17 (строго говоря, исключений только 15 с номерами 0, 1, 3...8, 10...14, 16 и 17; вектор 2 закреплен за немаскируемыми прерываниями, возникающими при поступлении сигнала на ввод NMI микропроцессора, а векторы 9 и 15 в МП 486 не используются). При возникновении исключения процессор умножает его номер на 8 и полученное произведение использует, как индекс в таблице дескрипторов прерываний IDT. Таким образом, первые 18 дескрипторов IDT должны всегда описывать программы обработчиков исключений. Следующие 14 векторов с номерами 18...31 зарезервированы и не должны использоваться в программах.

Различные устройства компьютера (таймер, клавиатура и др.) сигнализируют о необходимости программного вмешательства в их работу с помощью сигнала прерывания, который, пройдя через контроллер прерываний, поступает на вход INTR микропроцессора и инициирует в нем выполнение процедуры прерывания. В состав этой процедуры входит, в частности, чтение номера вектора, устанавливаемого контроллером прерываний на шине данных компьютера. Как известно, контроллер прерываний формирует передаваемый в процессор номер вектора

путем сложения базового номера, хранящегося в контроллере, с номером линии, по которой поступил запрос от устройства. Номер базового вектора (в принципе в диапазоне 0...255) устанавливается в процессе программной инициализации контроллера. Поскольку в защищенном режиме векторы 0...31 зарезервированы за исключениями, базовые векторы контроллеров прерываний должны быть расположены в диапазоне 32...248. Таким образом, для обработки внешних аппаратных прерываний в защищенном режиме необходимо перепрограммировать контроллеры прерываний компьютера.

Сигнал немаскируемого аппаратного прерывания, поступающий на вход NMI микропроцессора, возникает в результате серьезного аппаратного сбоя в работе компьютера. Как уже отмечалось выше, для него зарезервирован вектор прерывания с номером 2.

Программные прерывания возникают при выполнении процессором команды int с числовым аргументом. В принципе этот аргумент может принимать значения от 0 до 255, однако практически для программных прерываний допустимо использовать только векторы, оставшиеся свободными после размещения, во-первых, векторов исключений (32 вектора) и, во-вторых, векторов аппаратных прерываний (16 векторов). При этом два программных прерывания, именно, команда int 3, служащая для отладки, и команда into, фиксирующая арифметическое переполнение (overflow), обрабатываются в составе таблицы исключений. Для первой выделен вектор с номером 3, для второй - с номером 4.

Нарушения в работе программы, приводящие к исключениям, могут иметь разную природу и разные возможности исправления в процессе выполнения программы. В соответствии с этим исключения подразделяются на три класса: нарушения, ловушки и аварии.

Нарушение, или отказ (fault) - это исключение, фиксируемое еще до выполнения команды или в процессе ее выполнения. Типичными примерами нарушений являются адресация за установленной границей сегмента или обращение к отсутствующему дескриптору. При обработке нарушения процессор сохраняет в стеке адрес той команды, выполнение которой привело к исключению. При этом предполагается, что в обработчике нарушения его причина будет ликвидирована, после чего команда iret вернет управление на ту же, еще не выполненную команду. Таким образом, сам механизм обработки нарушений предполагает восстановление этого программного сбоя.

Ловушка (trap) обрабатывается процессором после выполнения команды, вызвавшей это исключение, и в стеке сохраняется адрес не этой, а следующей команды. Таким образом, после возврата из обработчика ловушки выполняется не команда, инициировавшая исключение, а следующая за ней команда программы. К ловушкам относятся все команды программных прерываний int.

Авария, или выход из процесса (abort) является следствием серьезных невосстановимых ошибок, например, обнаружение в системных таблицах неразрешенных или несовместимых значений. Адрес, сохраняемый в стеке, не позволяет локализовать вызвавшую исключение команду, и восстановление программы не предполагается. Обычно аварии требуют перезагрузки системы.

Процессор, зарегистрировав исключение того или иного вида, сохраняет в стеке содержимое расширенного регистра флагов EFLAGS, селектор сегмента команд, смещение точки возврата, а также (в некоторых случаях) 32-битовый код ошибки (рис. 64.1). Даже если программа выполняется в режиме 16-битовых адресов и операндов, в качестве смещения возврата выступает 32-битовое содержимое указателя команд EIP, в котором старшая половина, очевидно, равна 0. Стоит упомянуть, что двухсловные данные располагаются в стеке всегда в таком порядке: в слове стека с меньшим адресом - младшая часть 32-битового данного, в слове стека с большим (на 2) адресом - старшая часть. Для выравнивания стека под содержимое CS также отводится двойное слово, в младшей половине которого располагается CS, а старшая половина ничем не заполняется. Для общности на рисунке обозначен 32-разрядный указатель стека ESP, хотя в режиме 16-разрядных операндов фактически используется только его младшая половина SP.

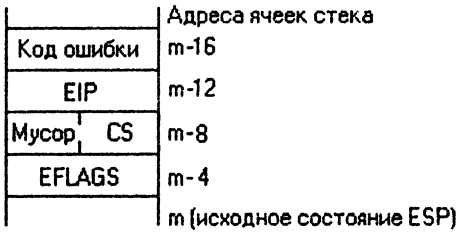


Рис. 64.1. Состояние стека при исключении.

Код ошибки, включаемый для некоторых исключений в стек, позволяет получить дополнительную информацию об исключении, которую может использовать его обработчик. В любом случае перед завершением обработчика (командной iret) код ошибки следует снять со стека.

Таким образом, для правильной обработки исключения необходимо знать, помимо причины его возникновения, к какому виду оно принадлежит и куда вернется управление после завершения обработчика, а также включен ли в стек код ошибки. В таблице 64.1 приведен список всех исключений с краткими характеристиками.

При переходе на обработчик исключения процессор заносит в стек адрес возврата. В обработчике исключения этот адрес можно извлечь и вывести на экран вместе с номером возникшего исключения. Такое усовершенствование обработчиков существенно упрощает процесс отладки программ защищенного режима, так как позволяет определить, в

какой именно точке программы произошло нарушение ее работы. При извлечении из стека адреса возврата следует учитывать возможность наличия в стеке кода ошибки.

Таблица 64.1. Исключения процессора 486

Вектор	Название исключения	Класс исключения	Код ошибки	Команды, вызывающие исключение
0	Ошибка деления	Нарушение	Нет	div, idiv
1	Исключение отладки	Нарушение /ловушка	Нет	Любая команда
2	Немаскируемое прерывание			
3	int 3	Ловушка	Нет	int 3
4	Переполнение	Ловушка	Нет	into
5	Нарушение границы массива	Нарушение	Нет	bound
6	Недопустимый код команды	Нарушение	Нет	Любая команда
7	Сопроцессор недоступен	Нарушение	Нет	esc, wait
8	Двойное нарушение	Авария	Да	Любая команда
9	Выход сопроцессора из сегмента(80386)	Авария	Нет	Команда сопроцессора с обращением к памяти
10	Недопустимый сегмент состояния задачи TSS	Нарушение	Да	jmp, call, iret, прерывание
11	Отсутствие сегмента	Нарушение	Да	Команда загрузки сегментного регистра
12	Ошибка обращения к стеку	Нарушение	Да	Команда обращения к стеку
13	Общая защита	Нарушение	Да	Команда обращения к памяти
14	Страничное нарушение	Нарушение	Да	Команда обращения к памяти
15	Зарезервировано			
16	Ошибка сопроцессора	Нарушение	Нет	esc, wait
17	Ошибка выравнивания	Нарушение	Да	Команда обращения к памяти
18...31 Зарезервированы				
32...255 Предоставлены пользователю для аппаратных прерываний и команд int				

Усовершенствованные программы обработчиков исключений приведены ниже.

```
exc_0   proc                ;Обработчик исключения 0
        pop     EAX         ;AX=IP
        mov     SI,offset string+5;Выведем адрес
        debug               ;возврата на экран
        mov     AX,0        ;Номер исключения
        jmp     dword ptr home_sel;На выход из обработчика
```

```
exc_0   endp
exc_1   proc            ;Обработчик исключения 1
        pop   EAX       ;AX=IP
        mov   SI,offset string+5;Выведем адрес
        debug           ;возврата на экран
        mov   AX,1      ;Номер исключения
        jmp   dword ptr home_sel;На выход из обработчика
exc_1   endp
;Следующие 6 обработчиков (exc_2...exc_7) аналогичны предыдущим
exc_8   proc
        pop   EAX       ;AX=код ошибки (нам не нужен)
        pop   EAX       ;AX=IP
        mov   SI,offset string+5
        debug
        mov   AX,8
        jmp   dword ptr home_sel
exc_8   endp
exc_9   proc
        ...             ;Аналогично exc_0
exc_9   endp
exc_0ah proc
        ...             ;Аналогично esc_8
exc_0ah endp
exc_0bh proc
        ...             ;Аналогично esc_8
exc_0bh endp
exc_0ch proc
        ...             ;Аналогично esc_8
exc_0ch endp
exc_0dh proc
        ...             ;Аналогично esc_8
exc_0dh endp
exc_0eh proc
        ...             ;Аналогично esc_8
exc_0eh endp
exc_0fh proc
        ...             ;Аналогично esc_0
exc_0fh endp
exc_10h proc
        ...             ;Аналогично esc_0
exc_10h endp
exc_11h proc
        ...             ;Аналогично esc_0
exc_11h endp
```

Внеся в программму предыдущей статьи описанные выше усовершенствования, воспользуемся ею для демонстрации причин возникновения исключений. Отлаженная нормально работающая программа выводит на экран последовательность цветных символов, а также диагностическую строку, которая имеет вид

```
FFFF ****_**** ****_**** ****
```

Если в процессе выполнения программы возникает исключение, вместо числа FFFF выводится 16-ричный номер исключения. Включим

в программу различные варианты неправильных программных строк, которые приведут к исключениям. Эти строки можно помещать в программу в любом месте, где процессор работает в защищенном режиме, например, после предложения 176. Естественно, перед включением в программу нового исследовательского фрагмента, старый надо удалить и, прогнав программу, убедиться в том, что ее работоспособность восстановлена.

При отладке программ защищенного режима чаще всего возникает исключение 13 нарушения общей защиты. Попробуем обратиться к сегменту данных по относительному адресу, выходящему за его пределы:

```
mov    AX,DS:[data_size-1];Возникает исключение 13
```

Байт с номером data_size-1 является последним байтом сегмента команд. Если бы мы обратились по этому адресу командой чтения байта

```
mov    AL,DS:[data_size-1];Команда не вызывает нарушений
```

все было бы нормально. Однако при чтении целого слова второй байт этого слова оказывается за пределами сегмента данных и возникает нарушение общей защиты. Таким образом, в защищенном режиме процессор строго следит за тем, чтобы программа не обращалась к не выделенным ей участкам памяти.

Попробуем теперь прочитать байт из сегмента команд. Атрибут 98h, который мы присвоили сегменту команд, говорит о том, что это исполняемый сегмент, который запрещается читать или модифицировать. Команда

```
mov    AL,CS[0];Команда вызывает нарушение общей защиты
```

должна прочитать в регистр AL первый байт сегмента команд (который в нашем случае равен B8h, код команды mov). Однако команда выполнена не будет, а возбудит нарушение общей защиты. Таким образом, в защищенном режиме коды команд в сегменте команд нельзя ни модифицировать, ни даже читать.

Нарушение общей защиты возникает и в том случае, когда программа обращается к памяти за границей таблицы дескрипторов (вспомним, что в псевдодескрипторе, загружаемом в регистр таблицы дескрипторов, имеется не только поле базы таблицы, но и поле ее границы). Включим в программу строку загрузки в сегментный регистр отсутствующего в таблице глобальных дескрипторов селектора:

```
mov    AX,40
mov    FS,AX;Команда вызывает нарушение общей защиты
```

Вторая команда этого фрагмента должна загрузить в теневой регистр, связанный с сегментным регистром FS, дескриптор, отвечающий

селектору 40, т.е. шестой по счету дескриптор. Однако в нашей таблице только пять дескрипторов, поэтому при выполнении команды возникает исключение общей защиты.

Схожая ситуация возникает, если в программе встречается команда прерывания int с номером, отсутствующим в таблице дескрипторов прерываний. Команда

```
int    40;Команда вызывает нарушение общей защиты
```

возбудит исключение общей защиты, поскольку она должна извлечь из таблицы IDT дескриптор с порядковым номером 40, а наша таблица имеет только 18 дескрипторов прерываний. А вот команда

```
int    3              ;Нарушения не будет
```

является совершенно законной, так как для нее в IDT предусмотрен дескриптор, а в сегменте команд соответствующий обработчик (который всего лишь выведет в первую позицию диагностической строки число 0003).

Между прочим, все команды int с аргументами в пределах 0...16 являются формально законными. Они будут программно вызывать обработчики исключений (чем можно воспользоваться для их отладки). Однако в законченных программных продуктах такими командами, естественно, пользоваться нельзя.

Выше уже упоминалось, что обработчики исключений, содержащиеся в нашей программе, построены слишком грубо. Они только сигнализируют об исключении, после чего программа аварийно завершается. В действительности обработчик исключения должен анализировать причину нарушения работы программы и по возможности устранять ее. Рассмотрим принцип восстановления работоспособности программы после регистрации исключения на примере операции деления. Пусть в нашей программе выполняется одна или несколько операций деления 32-битовых чисел на 16-битовые. В этом случае делимое однозначно находится в регистрах DX:AX, частное занимает регистр AX, а остаток, если он есть - регистр DX. Делитель может располагаться в регистре или памяти. Выделим для делителя регистр BX.

Исключение ошибки деления возникает в двух случаях: если делитель равен 0, и если частное не помещается в отведенный для него регистр. Будет считать, что в нашей программе может иметь место только первая ситуация. Для того, чтобы ее исправить, не нарушая работоспособности программы, надо заслать в BX число, отличное от 0, например, 1 и повторить операцию деления. Можно поступить по-другому: просто игнорировать ошибочную операцию, передав управление на следующую за операцией деления команду. Рассмотрим сначала первый вариант обработчика.

Включим в программу (после того же предложения 176) фрагмент, в котором выполняется деление содержимого DX:AX на содержимое BX:

```
mov   DX,0        ;Старшая половина делимого
mov   AX,8        ;Младшая половина делимого
mov   BX,0        ;Делитель=0
div   BX          ;Команда деления
```

Выполним программу с исходным вариантом обработчика исключения 0. На экран будет выведена информация о возникновении исключения 0, и программа аварийно завершится. Теперь изменим текст процедуры exc_0 на следующий:

```
exc_0   proc              ;Обработчик исключения 0
        mov   BX,1        ;Заменим делитель на 1
        db    66h         ;Префикс замены размера операнда
        iret              ;Возврат на команду div
exc_0   endp              ;Конец процедуры обработчика
```

Как уже отмечалось выше, процессор, передавая управление обработчику нарушения, сохраняет в стеке в качестве адреса возврата адрес той команды, попытка выполнения которой привела к возникновению исключения. Таким образом, команда iret, которой заканчивается обработчик, передает управление на команду div, обеспечивая ее повторное выполнение. Поскольку перед этим делитель (в регистре BX) сделан равен 1, деление выполняется успешно и повторное исключение не возникает. В данной ситуации перед командой iret необходимо поставить префикс замены размера операнда 66h. В результате команда iret работает в 32-битовом режиме, снимая со стека не три слова (CS, IP, флаги), как в реальном режиме, а три двойных слова (CS, EIP, EFLAGS). Код ошибки для исключения 0 отсутствует.

Для реализации второго алгоритма (пропуск "дефектной" команды) следует изменить текст процедуры exc_0 следующим образом:

```
exc_0   proc              ;Обработчик исключения 0
        mov   BP,SP       ;Перешлем SP в базовый регистр BP
        add dword ptr word ptr[BP],2;Увеличим адрес возврата на 2
        db    66h         ;Префикс замены размера операнда
        iret              ;Возврат на команду, следующую за div
exc_0   endp              ;Конец процедуры обработчика
```

Здесь перед возвратом в программу относительный адрес в стеке увеличивается на 2 (длина команды div BX), в результате чего возврат осуществляется не на команду div, а на следующую за ней.

Продемонстрируем теперь исключение нарушения стека. После предложения 176 стек должен находиться в исходном состоянии, т.е. SP=100h. Выведем на экран для контроля содержимое SP и попробуем извлечь из стека одно слово. Поскольку стек пуст, это слово будет извлекаться из области памяти, лежащей за его границей.

```
mov    AX,SP       ;Выведем на экран содержимое SP
mov    SI,offset string+5;для контроля
debug
pop    AX          ;Команда вызывает нарушение стека
```

Возникает нарушение с номером Ch - ошибка обращения к стеку.

Рассмотрим еще одно исключение, возникающее при обращении к сегменту, помеченному, как отсутствующий (исключение с вектором 11). Это исключение связано с фундаментальным для защищенного режима понятием присутствия (или отсутствия) сегмента. В дескрипторе любого сегмента имеется бит присутствия сегмента в памяти. Для дескрипторов сегментов памяти (т.е. сегментов команд, данных или стека) этот бит (обычно он обозначается P от Present, присутствующий) находится в разряде 7 байта атрибутов 1.

Бит присутствия предназначен для организации виртуального режима, в котором суммарный размер всех выполняемых одновременно программ может превышать фактический объем оперативной памяти. В этом случае операционная система хранит часть сегментов программ на диске, загружая их в память по мере необходимости на место тех сегментов, к которым временно не происходит обращений. В частности, система может загружать в память то одну, то другую, то третью задачи, создавая иллюзию их одновременной работы.

Таким образом, работа с битом присутствия - функция операционной системы. Выгрузив какой-то сегмент на диск, система сбрасывает бит P в дескрипторе этого сегмента (дескриптор находится в памяти). Если программа делает попытку обращения к отсутствующему сегменту, возникает исключение, которое в конечном счете должно заставить систему загрузить требуемый сегмент в память (возможно, выгрузив перед этим другой сегмент). После загрузки сегмента в память система устанавливает в его дескрипторе бит P, помечая его присутствующим. Последующие обращения к этому сегменту будут выполняться без всяких нарушений.

Если прикладная программа берет на себя часть функций операционной системы, она может сама устанавливать и сбрасывать бит присутствия, регулируя тем самым (с помощью исключения отсутствия сегмента) доступ к сегментам. В простых случаях, когда все сегменты всегда находятся в памяти, биты присутствия в их дескрипторах должны быть установлены.

Продемонстрируем работу с битом присутствия и обработку исключения отсутствия сегмента (вектор 11).

Исключение отсутствия сегмента возникает в тот момент, когда в какой-либо сегментный регистр загружается селектор сегмента, в дескрипторе которого сброшен бит присутствия. Это исключение отличается от уже рассмотренных нами тем, что перед передачей управле-

ния обработчику процессор заносит в стек, кроме регистра флагов и адреса возврата, еще и двухсловный код ошибки.

Модифицируем программу, предусмотрев в ней еще один сегмент данных data1. Для этого надо, во-первых, добавить в программу описание самого сегмента, включив в него произвольные тестовые данные:

```
data1   segment
numb    dd      12345678h
data1_size=$-ddd
data1   ends
```

Во-вторых, в таблицу глобальных дескрипторов надо включить дескриптор нового сегмента. Это дескриптор номер 5 (если вести отсчет от нуля) и ему будет соответствовать селектор 40:

```
gdt_data1 descr <data1_size-1,0,0,92h,0,0> ;Селектор 40
```

В текст программы (на участке, где процессор находится в защищенном режиме, например после все того же предложения 176) включим строки обращения к новому сегменту. Прочитаем данные из него и выведем их на экран с помощью наших отладочных средств:

```
mov   AX,40       ;Настроим на новый сегмент
mov   FS,AX       ;свободный сегментный регистр FS
mov   EAX,FS:[0] ;Прочитаем данные numb из сегмента
mov   SI,offset string+15;Выведем их на экран
debug             ;Сначала младшую половину
rol   EAX,16      ;Затем
mov   SI,offset string+10;старшую
debug             ;половину
```

Убедимся, что программа работает правильно и диагностическая строка имеет вид

```
FFFF ****-1234 5678-**** ****
```

Теперь перед приведенным выше фрагментом (перед предложениями инициализации регистра FS) сбросим бит присутствия в дескрипторе сегмента data1:

```
and   gdt_data1.attr_1,7Fh
```

Повторим прогон программы. Диагностическая строка примет вид

```
000B xxxx-**** ****-**** ****
```

что говорит о возникновении исключения 11 отсутствия сегмента при выполнении команды по адресу xxxx. Теперь попробуем с помощью обработчика этого исключения динамически восстановить работоспособность программы. Учтем при этом, что перед передачей управления обработчику исключения процессор сохраняет в стеке, кроме регистра флагов EFLAGS и четырехсловного адреса возврата, еще и двухсловный

код ошибки (на самом верху стека). Формат кода ошибки зависит от номера исключения. Для исключения 11 (а также для исключений 10, 12 и 13) код ошибки имеет формат, приведенный на рис. 64.2.

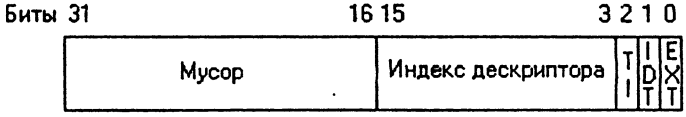


Рис. 64.2. Формат кода ошибки для исключений сегментов.

Бит 0 (EXT от External, внешний) равен 1, если исключение возникло в процессе обработки другого исключения или внешнего прерывания.

Бит 1 (IDT) равен 1, если исключение возникло в процессе чтения элемента таблицы дескрипторов прерываний, что может иметь место только при обработке исключения или прерывания.

Бит 2 (TI от Table Indicator, индикатор таблицы) равен 1, если дескриптор находится в таблице локальных дескрипторов, и 0, если дескриптор глобальный. В нашем случае все дескрипторы - глобальные, так как таблицы локальных дескрипторов у нас нет.

Для нашей простой программы, где отсутствует LDT и не ожидается ошибок при обращении к IDT, в младшем слове кода ошибки фактически содержится селектор того сегмента, при обращении к которому произошло нарушение.

Обработчик исключения 11 примет теперь вид:

```
exc_0bh proc                ;(1)
        push   BP           ;(2)Сохраним BP программы
        mov    BP,SP        ;(3)Запомним в BP текущее SP
        push   EAX          ;(4)Сохраним EAX программы
        push   EBX          ;(5)Сохраним EBX программы
        mov    EAX,[BP]+2   ;(6)В EAX код ошибки
        mov    BX,AX        ;(7)Сохраним селектор в BX
        mov    SI,offset string+25;(8)Выведем для контроля
        debug               ;(9)в диагностическую
        rol    EAX,16       ;(10)строку
        mov    SI,offset string+20;(11)двухсловный
        debug               ;(12)код ошибки
        or     byte ptr gdt_null[BX]+attr_1,80h;(13)Сделаем
                            ;сегмент присутствующим
        pop    EBX          ;(14)Восстановим EBX программы
        pop    EAX          ;(15)Восстановим EAX программы
        pop    BP           ;(16)Восстановим BP программы
        add    SP,4         ;(17)Коррекция стека на поле кода ошибки
        db     66h          ;(18)Назад в программу на
        iret                ;(19)повторение команды mov FS,AX
exc_0bh endp                ;(20)
```

Программа обработчика начинается, как и положено, с сохранения используемых в ней регистров. Состояния стека при входе в обработчик и в процессе его работы показаны на рис. 64.3.

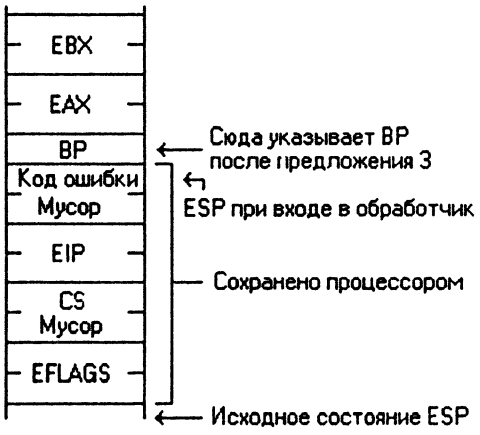


Рис. 64.3. Состояния стека в обработчике исключения exc_0bh.

После сохранения в стеке регистра BP в него заносится текущее значение SP. Далее в стеке сохраняются регистры EAX и EBX. В регистр EAX загружается из стека двойное слово с кодом ошибки. Адресация к стеку осуществляется в этом случае через регистр BP и состояние указателя стека SP не изменяется. Поскольку селектор, находящийся сейчас в регистре AX, будет использоваться, как индекс в таблице глобальных дескрипторов, он переносится в BX (предложение 7). Далее содержимое сначала младшей половины EAX (т.е. AX), а затем старшей половины EAX (командой rol сдвинутой в AX) преобразуется макрокомандой debug в символьную форму и заносится в диагностическую строку string (предложения 8...12). На ходе программы эти действия не отражаются, но позволяют увидеть на экране содержимое двойного слова кода ошибки.

В предложении 13 выполняется основное действие обработчика: в байте атрибутов дескриптора сегмента data2 устанавливается бит присутствия P. При этом адресация дескриптора осуществляется с помощью полученного в коде ошибки селектора (который сейчас находится в BX), чем обеспечивается универсальность обработчика. Тремя последующими командами восстанавливаются сохраненные в стеке регистры, в результате чего указатель стека возвращается в состояние, в каком он был при входе в обработчик. В предложении 17 прибавлением 4 к содержимому SP указатель стека переводится на двойное слово с адресом возврата, после чего командой iret, настроенной на работу с 32-битовыми операндами, осуществляется возврат на команду загрузки сегментного регистра

```
mov FS,AX
```

в процессе выполнения которой возникло исключение. При этом, хотя исключение возникло уже в процессе выполнения этой команды (когда анализ содержимого адресуемого через селектор дескриптора показал, что его бит присутствия сброшен), процессор восстанавливает исходное содержимое участвующих в выполнении команды регистров, так что после возврата из обработчика командой iret эта команда как бы выполняется впервые. Программа продолжается дальше обычным образом: на экран выводится последовательность цветных символов, а затем - диагностическая строка:

```
FFFF ****-1234 5678-5E5E 0028
```

Здесь 12345678 - содержимое двойного слова numb в сегменте data1, 5E5E - старшая половина кода ошибки, заполненная исходным содержимым стека (коды символов ^), а 0028 - собственно код ошибки, который в данном случае представляет собой селектор сегмента data1 (28h=40).

Статья 65

Обработка аппаратных прерываний в защищенном режиме

В предыдущих статьях мы рассмотрели обработку внутренних прерываний процессора, так называемых исключений. Перейдем теперь к обработке внешних, аппаратных прерываний, которые часто называют просто прерываниями. В принципе обработка прерывания выполняется, за исключением своей начальной аппаратной стадии, так же, как и обработка исключения: при поступлении на вход INT микропроцессора сигнала от внешнего устройства процессор считывает с шины данных выставленный контроллером прерываний номер вектора, находит в таблице дескрипторов прерываний дескриптор с соответствующим номером и, сохранив предварительно в стеке адрес возврата, осуществляет переход на обработчик прерывания. Команда iret, которой заканчивается обработчик прерывания, возвращает управление в программу. Таким образом, для обработки аппаратных прерываний мы должны дополнить таблицу IDT дескрипторами обработчиков аппаратных прерываний и включить сами обработчики в текст программы.

В действительности, однако, дело обстоит несколько сложнее.

Поскольку первые 32 вектора зарезервированы для обработки исключений, аппаратным прерываниям придется назначить какие-то другие векторы, например, начиная с номера 32=20h. Однако в машинах типа IBM PC контроллеры прерываний всегда программируются в процессе начальной загрузки так, что базовый вектор ведущего контроллера равен 8, а ведомого - 70h. Таким образом, перед переходом в защищенный режим нам придется перепрограммировать контроллеры прерываний, назначив, например, ведущему контроллеру базовый вектор 20h, а ведомому - 28h (или оставив у ведомого контроллера базовый вектор 70h). Между прочим, различие базовых векторов в реальном и защищенном режимах является еще одной причиной программной несовместимости этих режимов.

Очевидно, что перед возвратом в реальный режим контроллеры надо снова перепрограммировать, иначе не смогут работать обработчики аппаратных прерываний BIOS. Однако это дело относительно трудоемкое. Можно поступить проще: переход в реальный режим реализовать с помощью отключения процессора, предварительно загрузив в область данных BIOS по адресу 40h:67h двухсловный адрес возврата в нашу программу. Если при этом в байт Fh КМОП-микросхемы заслать вместо кода 0Ah, как это мы делали (см. пример 61.1, предложения 64-65), код 05h, то после сброса процессора программы BIOS выполнят перепрограммирование контроллеров прерываний, после чего, как и раньше, передадут управление в указанную нами точку.

Вторая особенность обработки прерываний связана с форматом дескриптора прерывания (шлюза).

Как уже отмечалось в статье 63, в таблицу IDT могут входить шлюзы нескольких типов. В дескрипторе тип шлюза указывается в специально предназначенном для этого байте атрибутов (см. рис. 63.1). Формат этого байта изображен на рис. 65.1.

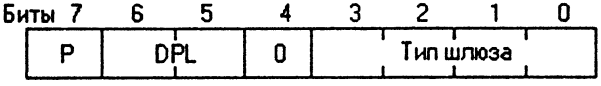


Рис. 65.1. Формат байта атрибутов шлюза.

Вообще поле типа, занимающее 4 бита, может принимать 16 различных значений, однако в таблице IDT допустимо описывать только шлюзы следующих типов:

шлюз задачи (тип 5);
шлюз прерывания МП 286 (тип 6);
шлюз ловушки МП 286 (тип 7);
шлюз прерывания МП 386 или 486 (тип Eh);
шлюз ловушки МП 386 или 486 (тип Fh).

Шлюз задачи используется в тех случаях, когда обработчик прерывания расположен в другой задаче; шлюзы прерываний и ловушек предполагают наличие обработчиков в текущей (прерываемой) задаче. Как видно из приведенного перечня, дескрипторы аппаратных прерываний должны иметь тип Fh.

Поле DPL определяет уровень привилегий шлюза. Как уже отмечалось ранее, уровень привилегий может принимать значения от 0 (максимальные привилегии) до 3 (минимальные привилегии) и используется для защиты программ друг от друга. DPL шлюза проверяется процессором только при выполнении команд программных прерываний-ловушек int и into, чтобы предотвратить обращение программ пользователя к системным программным прерываниям. Для остальных исключений и прерываний поле DPL процессором игнорируется.

Бит P в дескрипторе шлюза должен быть установлен в 1, в противном случае процессор будет рассматривать шлюз, как неправильный, и обращение к такому шлюзу вызовет исключение.

Таким образом, атрибут дескриптора шлюза прерывания МП 486 должен быт равен 8Eh, в отличие от дескрипторов ловушек, где он равен 8Fh.

Так же, как это имеет место и в реальном режиме, если переход на обработчик осуществляется через шлюз прерывания, процессор сбрасывает при входе в обработчик флаг IF в регистре флагов EFLAGS. Команда iret, загружая из стека сохраненное там исходное содержимое EFLAGS, снова разрешает прерывания. При переходе на обработчик через шлюз ловушки состояние флага IF не изменяется.

Модифицируем программу 63.1, введя в нее обработку аппаратных прерываний. Для простоты ограничимся содержательной обработкой прерываний только от одного источника - таймера. Поскольку в компьютере таймер является единственным постоянно активным источником прерываний, а другие прерывания (от клавиатуры, дисков, КМОП-микросхемы и т.д.) сами по себе не возникают, мы можем выполнить инициализацию только этого прерывания: ограничиться перепрограммированием только ведущего контроллера и включить в таблицу IDT лишь один шлюз прерывания. Чтобы полностью обезопасить себя от незапланированных прерываний, их можно замаскировать в контроллерах прерываний.

Для сокращения размера программы упростим процедуру обработки исключений. В программе 63.1 каждому исключению соответствовал свой обработчик. Такое построение программы вместе со средствами отладки (макрокоманда debug) весьма удобно, так как позволяет, в случае каких-либо неправильностей в программе, сразу же получить на экране номер возникшего исключения. Однако программа оказывается довольно громоздкой. В то же время практически при отладке наших

программ будут возникать лишь исключения с номерами 10...14. Поэтому мы оставим только обработчики этих исключений, а для всех остальных (в том числе зарезервированных) исключений предусмотрим общий обработчик, который выводит на экран условный код 1111.

Пример 65.1. Обработка аппаратных прерываний от таймера

```
include debug.mac          ;(1)
.386P                      ;(2)
;Структура для описания дескрипторов сегментов
descr    struc             ;(3)
         ...               ;См. предыдущие примеры
descr    ends              ;(4)
;Структура для описания шлюзов ловушек
trap     struc             ;(5)
offs_l   dw     0          ;(6)Смещение обработчика, биты 0...15
sel      dw     16         ;(7)Селектор сегмента команд
rsrv     db     0          ;(8)Зарезервировано
attr     db     8Fh        ;(9)Присутствие+шлюз ловушки
offs_h   dw     0          ;(10)Смещение обработчика, биты 16...31
trap     ends              ;(11)
;Структура для описания шлюзов прерываний
intr     struc             ;(12)
ioffs_l  dw     0          ;(13)Смещение обработчика, биты 0...15
isel     dw     16         ;(14)Селектор сегмента команд
irsrv    db     0          ;(15)Зарезервировано
iattr    db     8Eh        ;(16)Присутствие+шлюз прерывания
ioffs_h  dw     0          ;(17)Смещение обработчика, биты 16...31
intr     ends              ;(18)
data     segment use16     ;(19)
;Таблица глобальных дескрипторов GDT
gdt_0    label  word       ;(20)Начало GDT
gdt_null descr <0,0,0,0,0,0>;(21)Селектор 0
gdt_data descr <data_size-1,,,92h>;(22)Селектор 8, сегмент данных
gdt_code descr <code_size-1,,,98h>;(23)Селектор 16, сегмент команд
gdt_stack descr <255,,,92h>   ;(24)Селектор 24 - сегмент стека
gdt_screen descr <4095,8000h,0Bh,92h>;(25)Селектор 32, видеобуфер
gdt_size=$-gdt_0           ;(26)Размер GDT
;Таблица дескрипторов прерываний IDT
idt      label  word       ;(27)Начало IDT
;Дескрипторы исключений
trap     10 dup (<dummy_exc>);(28)Общий дескриптор исключений
trap     <exc_0ah>         ;(29)Дескриптор исключения 10
trap     <exc_0bh>         ;(30)Дескриптор исключения 11
trap     <exc_0ch>         ;(31)Дескриптор исключения 12
trap     <exc_0dh>         ;(32)Дескриптор исключения 13
trap     <exc_0eh>         ;(33)Дескриптор исключения 14
trap     17 dup (<dummy_exc>);(34)Дескрипторы зарезервированных
                           ;исключений
idt_08   intr   <new_08h>  ;(35)Вектор 20h, прерывание от таймера
idt_size=$-idt             ;(36)Размер таблицы IDT
;Поля данных программы
pdescr   dq     0          ;(37)Псевдодескриптор
mes db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$';(38)
tblhex db  '0123456789ABCDEF';(39)Таблица преобразования bin-hex
string  db '**** ****_**** ****_**** ****';(40)Шаблон для вывода
```

```
len=$-string                ;(41)
home_sel dw    home         ;(42)Адрес возврата из исключения
         dw    10h          ;(43)Сегмент команд
mark_08h dw    480          ;(44)Позиция на экране для вывода
                            ;из обработчика new_08h
color_08h db   71h          ;(45)Атрибут для new_08h
time_08h db    0            ;(46)Счетчик прерываний
data_size=$-gdt_0           ;(47)Размер сегмента данных
data     ends               ;(48)
text     segment 'code' use16;(49)
         assume CS:text,DS:data;(50)
begin    label  word        ;(51)Начало сегмента команд
;Обработчик исключения недопустимого сегмента состояния задачи TSS
exc_0ah proc                ;(52)
         pop    EAX         ;(53)AX=код ошибки (нам не нужен)
         pop    EAX         ;(54)AX=IP
         mov    SI,offset string+5;(55)
         debug              ;(56)
         mov    AX,0Ah      ;(57)Номер исключения
         jmp    dword ptr home_sel;(58)
exc_0ah endp                ;(59)
;Обработчик исключения отсутствия сегмента
exc_0bh proc                ;(60)
         pop    EAX         ;(61)AX=код ошибки (нам не нужен)
         pop    EAX         ;(62)AX=IP
         mov    SI,offset string+5;(63)
         debug              ;(64)
         mov    AX,0Bh      ;(65)Номер исключения
         jmp    dword ptr home_sel;(66)
exc_0bh endp                ;(67)
;Обработчик исключения ошибки обращения к стеку
exc_0ch proc                ;(68)
         ...                ;Аналогично предыдущим
exc_0ch endp                ;(69)
;Обработчик исключения общей защиты
exc_0dh proc                ;(70)
         ...                ;Аналогично предыдущим
exc_0dh endp                ;(71)
;Обработчик исключения страничного нарушения
exc_0eh proc                ;(72)
         ...                ;Аналогично предыдущим
exc_0eh endp                ;(73)
;Обработчик остальных исключений
dummy_exc proc ;            ;(74)
         pop    EAX         ;(75)AX=Код ошибки
         pop    EAX         ;(76)AX=IP
         mov    SI,offset string+5;(77)
         debug              ;(78)
         mov    AX,1111h    ;(79)Условный код
         jmp    dword ptr home_sel;(80)
dummy_exc endp              ;(81)
new_08h proc                ;(82)Обработчик прерываний от
                            ;таймера (18.2 прерываний в сек)
         push   AX          ;(83)Сохраним используемые
         push   BX          ;(84)регистры
         test   time_08h,03h;(85)Пересчет на 4, чтобы снизить
         jnz    skip        ;(86)частоту вывода символа на экран
```

```
        mov     AL,21h      ;(87)Символ "!"
        mov     AH,color_08h;(88)Цвет
        mov     BX,mark_08h;(89)Позиция на экране
        mov     ES:[BX],AX ;(90)Отправим символ в видеобуфер
        add     mark_08h,2 ;(91)Смещение по экрану
skip:   inc     time_08h    ;(92)Пересчет прерываний
        mov     AL,20h      ;(93)EOI ведущего
        out     20h,AL      ;(94)контроллера
        pop     BX          ;(95)Восстановим используемые
        pop     AX          ;(96)регистры
        db      66h         ;(97)Возврат
        iret                ;(98)в программу
new_08h endp                ;(99)
main    proc                ;(100)Начало главной процедуры
        xor     EAX,EAX     ;(101)Очистим EAX
        mov     AX,data     ;(102)Загрузим в DS сегментный
        mov     DS,AX       ;(103)адрес сегмента данных
;Вычислим 32-битовый линейный адрес сегмента данных и загрузим его
;в дескриптор сегмента данных в GDT
        ...
;Аналогично для линейного адреса сегмента команд
        ...
;Аналогично для линейного адреса сегмента стека
        ...
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        ...
        lgdt    pdescr      ;(104)Загрузка GDTR
;Подготовимся к переходу в защищенный режим
        cli                 ;(105)Запрет аппаратных прерываний
        mov     AL,8Fh      ;(106)Запрет NMI
        out     70h,AL      ;(107)Порт КМОП-микросхемы
        jmp     $+2         ;(108)Задержка
        mov     AL,05h      ;(109)Байт состояния отключения 05:
        out     71h,AL      ;(110)возврат в программу +
                            ;перепрограммирование контроллеров
        mov     AX,40h      ;(111)Загрузим в область
        mov     ES,AX       ;(112)данных BIOS
        mov     word ptr ES:[67h],offset return;(113)адрес
        mov     word ptr ES:[69h],CS;(114)возврата в программу
;Перепрограммируем ведущий контроллер прерываний
;Базовый вектор теперь 32=20h
        mov     DX,20h      ;(115)Порт контроллера
        mov     AL,11h      ;(116)СКИ1: будет СКИ3
        out     DX,AL       ;(117)
        jmp     $+2         ;(118)Задержка
        inc     DX          ;(119)Второй порт контроллера
        mov     AL,20h      ;(120)СКИ2: базовый вектор
        out     DX,AL       ;(121)
        jmp     $+2         ;(122)Задержка
        mov     AL,4        ;(123)СКИ3: ведомый подключен к уровню 2
        out     DX,AL       ;(124)
        jmp     $+2         ;(125)Задержка
        mov     AL,1        ;(126)СКИ4: 80x86, требуется EOI
        out     DX,AL       ;(127)
        mov     AL,0FEh     ;(128)Маска прерываний
        out     DX,AL       ;(129)
;Запретим все прерывания в ведомом контроллере
```

```
        mov     DX,0A1h     ;(130)Порт ведомого контроллера
        mov     AL,0FFh     ;(131)Маска прерываний -
        out     DX,AL       ;(132)замаскируем все прерывания
;Таблица прерываний уже заполнена на этапе трансляции
;Загрузим IDTR
        ...
;Переходим в защищенный режим
        mov     EAX,CR0     ;(133)Получим содержимое CR0
        or      EAX,1       ;(134)Установим бит PE
        mov     CR0,EAX     ;(135)Запишем назад в CR0
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
;и заодно очищаем очередь команд
        db      0EAh        ;(136)Код команды far jmp
        dw      offset continue       ;(137)Смещение
        dw      16          ;(138)Селектор сегмента команд
continue:                   ;(139)
;Делаем адресуемыми данные
        ...
;Делаем адресуемым стек
        ..
;Инициализируем ES адресом видеобуфера
        ...
;Размаскируем NMI и аппаратные прерывания
        mov     AL,0h       ;(140)
        out     70h,AL      ;(141)
        sti                 ;(142)
;Организуем периодический вывод на экран символов
        ...                 ;См. пример 63.1, предл. 87...97
        mov     AX,0FFFFh   ;(143)Диагностическое значение
home:   mov     SI,offset string;(144)
        debug               ;(145)
;Выведем на экран диагностическую строку
        ...                 ;См. пример 63.1
;Переключим режим процессора
        mov     AL,0FEh     ;(146)Переходим в реальный режим
        out     64h,AL      ;(147)засылкой кода FEh в порт 64h
        hlt                 ;(148)Останов в ожидании сброса
;Теперь процессор снова работает в реальном режиме
;Восстановим операционную среду реального режима
return: mov     AX,data     ;(149)
        mov     DS,AX       ;(150)
;Восстановим SS:SP
        mov     AX,stk      ;(151)
        mov     SS,AX       ;(152)
        mov     SP,256      ;(153)
;Размаскируем прерывания в контроллере
        mov     AL,0B8h     ;(154)
        out     21h,AL      ;(155)
        mov     AL,9Dh      ;(156)
        out     0A1h,AL     ;(157)
;Разрешим аппаратные и немаскируемые прерывания
        sti                 ;(158)
        mov     AL,0        ;(159)Разрешим
        out     70h,AL      ;(160)NMI
;Проверим выполнение функций DOS и завершим программу
        mov     AH,09h      ;(161)
```

```
        mov     DX,offset mes;(162)
        int     21h           ;(163)
        mov     AX,4C00h      ;(164)Завершим программу обычным
        int     21h           ;(165)образом
main    endp                  ;(166)
code_size=$-begin             ;(167)Размер сегмента команд
text    ends                  ;(168)
stk     segment stack 'stack';(169)
        db      256 dup ('^');(160)
stk     ends                  ;(161)
        end main              ;(162)
```

Помимо структуры шлюзов ловушек, имевшейся и в программе 63.1, в данную программу включена структура шлюзов прерываний (предложения 12...18), отличающаяся только значением байта типа.

Таблица прерываний должна содержать дескрипторы, расположенные по порядку их векторов. Поэтому начинается таблица с 10 одинаковых дескрипторов исключений, имеющих в нашей программе общий обработчик dummy_exc. Затем идут дескрипторы исключений 10...14, а за ними еще 17 одинаковых дескрипторов исключений (реальных и зарезервированных). Наконец, на 33-м месте (вектор 32) описан дескриптор обработчика аппаратного прерывания от таймера (new_08h).

В сегмент данных включено несколько переменных для обслуживания обработчика от таймера (mark_08h, color_08h и time_08h)

В сегменте команд обработчики прерываний и исключений, так же, как и все остальные процедуры, могут следовать в любом порядке. Наш сегмент команд начинается с обработчиков исключений с номерами 10...14. Предложения 61...64 процедуры exc_0ah позволяют отправить в диагностическую строку string (на второе место в ней) содержимое IP, извлеченное из стека, т.е. адрес той команды, при выполнении которой возникло данное исключение. Произвольное смещение указателя стека (в стеке остались сегментный адрес и регистр EFLAGS) в данном случае не имеет значения, так как в случае исключения выполнение программы в защищенном режиме завершается, и процессор переводится в реальный режим.

Текст общего обработчика исключений dummy_esc практически повторяет процедуры обработки исключений 10...14, за тем исключением, что в строку string выводится не номер исключения, а условный код 1111h.

Далее следует обработчик прерываний от таймера (предложения 82...98). После сохранения используемых в нем регистров, выполняется проверка содержимого ячейки time_08h, причем программа обработчика продолжается, только если в этой ячейке сброшены оба младших бита. Тем самым осуществляется пересчет прерываний в отношении 4:1. На экран выводится цветной символ (конкретно восклицательный знак) и

выполняются инкременты позиции на экране (в ячейке mark_08h) и счетчика прерываний time_08h. Обработчик завершается генерацией сигнала конца прерывания EOI для ведущего контроллера (порт 20h), восстановлением сохраненных ранее регистров и командой iret.

С предложения 100 начинается главная процедура main. В ней обычным образом заполняются глобальные дескрипторы, загружается регистр GDTR, запрещаются немаскируемые и маскируемые прерывания. В предложениях 106..110 в КМОП-микросхеме устанавливается байт состояния отключения (код 5 - после сброса выполняется перепрограммирование контроллера и возврат в программу) и в область данных BIOS заносится адрес возврата в программу return.

С предложения 115 начинаются строки перепрограммирования контроллера прерываний. Некоторые настройки контроллера, например, изменение маски прерываний или генерацию сигнала EOI, можно выполнить засылкой соответствующего кода в порт управления контроллером. Однако сменить базовый вектор таким образом нельзя. 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, представлен на рис. 65.2.

Компьютер имеет два контроллера прерываний, поэтому СКИ1 равно 11h.

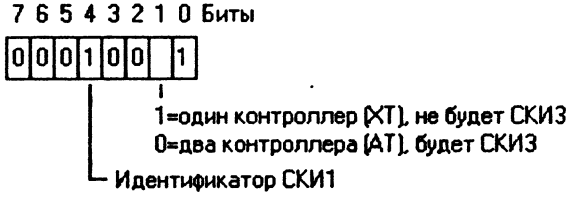


Рис. 65.2. Слово команд инициализации СКИ1.

Второе слово инициализации СКИ2, посылаемое во второй порт контроллера (для ведущего контроллера - 21h) задает базовый вектор. Мы решили расположить векторы прерываний сразу вслед за векторами исключений, поэтому базовый вектор первого контроллера равен 32=20h.

Третье слово инициализации СКИ3 выглядит по-разному для ведущего и ведомого контроллеров. Для ведущего оно определяет номер входа, к которому подсоединен ведомый контроллер. Этот номер записывается в виде двоичной 1, установленной в том бите слова, который соответствует входу для ведомого. Для всех компьютеров типа IBM PC ведомый контроллер подсоединяется ко входу 2 ведущего, поэтому в СКИ3 должен быть установлен бит 2, что соответствует числу 4. СКИ3 посылается (для ведущего контроллера) в порт 21h.

Формат четвертого слова инициализации СКИ4 представлен на рис. 65.3.

Для нашего случая (МП 80x86, требуется сигнал EOI) слово СКИ4 равно 1.На этом последовательность инициализации заканчивается, однако еще надо установить маску прерываний. Таймер подключен к уровню 0, поэтому слово маски, посылаемое в порт 20h, равно FEh.

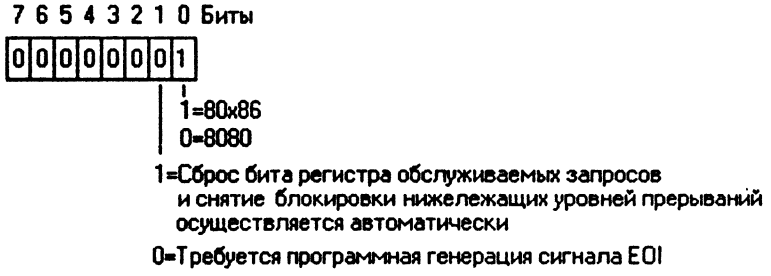


Рис. 65.3. Слово команд инициализации СКИ4.

Инициализация ведомого контроллера, от которого прерывания поступать заведомо не будут, в программе не выполняется, но для надежности все прерывания в нем маскируются (предложения 130...132). Теперь, когда вся система прерываний настроена, можно загрузить регистр IDTR, "подложив" процессору новую таблицу дескрипторов прерываний и исключений. Наконец, установкой бита 0 управляющего регистра CR0 процессор переводится в защищенный режим.

Программа защищенного режима не отличается от примера 63.1, за исключением того, что командой sti разрешаются аппаратные прерывания (предложение 142). Возврат в реальный режим выполняется посылкой команды сброса в порт 64h. Поскольку переход в реальный режим осуществляется через BIOS, после возврата в реальный режим заново инициализируются регистры SS:SP (предложения 151...153).

После сброса BIOS реинициализирует контроллеры прерываний и передает управление в задачу. При этом в обоих контроллерах устанавливаются значения масок прерываний FFh, т.е. все прерывания оказываются замаскированными. Поэтому сразу же после перехода в реальный режим следует установить в контроллерах правильные значения масок. Конкретные значения зависят от конфигурации компьютера, поэтому их надо определить экспериментально. Это можно сделать с помощью отладчика CV, запустив его с какой-либо программой и подав команды ввода из порта

```
i 21
i A1
```

На экран будет выведено значение масок прерываний. В примере 65.1 использованы значения B8h для ведущего контроллера и 9Dh для ведомого. (предложения 154...157). Далее разрешаются маскируемые и немаскируемые прерывания и на экран выводится тестовая строка.

Если программа подготовлена без ошибок, она будет функционировать следующим образом. В 17-ю и последующие строки экрана выводятся с небольшой скоростью черные символы по бирюзовому фону. Их количество (300) задано в программе. Одновременно в 3-ю строку с частотой 18,2/4 раз в секунду поступают изображения восклицательного знака (синие по белому фону). Перед завершением программы в 10-ю строку экрана выводится диагностическая строка

```
FFFF ****_**** ****_**** ****
```

и, наконец, после перехода процессора в реальный режим в позицию курсора красным по зеленому выводится строка

```
Вернулись в реальный режим!
```

Усложним предыдущий пример, сделав рассмотренную программу несколько более универсальной. Для этого введем в нее следующие добавления:

обработчик прерываний от клавиатуры;

инициализацию второго, ведомого контроллера прерываний;

обработчики-заглушки остальных уровней прерываний.

Все добавления носят локальный характер, поэтому мы не будем приводить полный текст новой программы.

Поскольку мы так или иначе будем обрабатывать все 15 аппаратных прерываний, в таблице дескрипторов прерываний должно быть соответствующее количество шлюзов. Добавим в нее после дескриптора int_08h (предложение 35) следующие строки:

```
idt_09  intr   <new_09h> ;Вектор 21h - прерывание от клавиатуры
        intr   6 dup (<master>);Векторы 22h...27h - аппаратные,
                        ;ведущий контроллер прерываний
        intr   8 dup (<slave>);Векторы 28h...2Fh - аппаратные,
                        ;ведомый контроллер прерываний
```

При этом мы однозначно определили, что векторы ведомого контроллера располагаются сразу вслед за векторами ведущего и, следовательно, номер базового вектора ведомого контроллера равен 20h+8=28h

Как видно из описания дескрипторов, прерывания от клавиатуры будут вызывать обработчик new_09h, остальные прерывания, проходящие через ведущий контроллер, - общий для них обработчик master, а все прерывания ведомого контроллера - общий для них обработчик slave.

В поля данных программы (например, после предложения 46) следует добавить две переменные, используемые обработчиком прерываний от клавиатуры:

```
mark_09h dw    800          ;Позиция на экране для вывода
                            ;обработчиком new_09h
color_09h db   1Eh          ;Атрибут символов
```

Вслед за обработчиком прерываний от таймера расположим процедуру new_09h обработчика прерываний от клавиатуры. В настоящем примере функции этого обработчика - прием скен-кодов нажимаемых клавиш и вывод их без всякого преобразования на экран. Естественно, на экране появляются символы, коды ASCII которых совпадают с поступающими скен-кодами. Эта программа поучительна в том отношении, что обрабатывает без фильтрации все прерывания от клавиатуры, отображая, таким образом, и скен-коды нажатия (make-codes), и скен-коды отпускания (break-codes). Более того, с ее помощью можно определить последовательности скен-кодов тех клавиш расширенной клавиатуры, которые при их нажатии генерируют не один, а несколько сигналов прерываний (<Pause>, <PrtScr>, клавиши со стрелками и др.). Текст процедуры new_09h приведен ниже:

```
new_09h proc   far
;Обработчик прерываний от клавиатуры. Обрабатываются оба скен-кода
;- и нажатия, и отпускания
        push   AX           ;Сохраним используемые
        push   BX           ;регистры
        in     AL,60h       ;Вводим скен-код из порта 60h
        mov    BX,mark_09h;Текущая позиция на экране
        mov    AH,color_09h;Атрибут символов
        mov    ES:[BX],AX ;Вывод символа в видеобуфер
        cmp    AL,80h       ;Скен-код нажатия (<80h)?
        jb     make         ;Да, после него сдвинемся на 1 место
        add    mark_09h,2 ;Нет, сдвинемся еще на одно место
make:   add    mark_09h,2
        in     AL,61h       ;Получим содержимое порта
        or     AL,80h       ;Установкой старшего бита
        out    61h,AL       ;и последующим сбросом его
        and    AL,7Fh       ;сообщим контроллеру клавиатуры о
        out    61h,AL       ;приеме скен-кода символа
        mov    AL,20h       ;Сигнал конца прерывания EOI
        out    20h,AL       ;в ведущий контроллер
        pop    BX           ;Восстановим используемые
        pop    AX           ;регистры
        db     66h          ;Возврат
        iret                ;в программу
new_09h endp
```

Помимо процедуры обработчика прерываний от клавиатуры, в сегмент команд следует включить обработчики "лишних" прерываний master (для ведущего контроллера) и slave (для ведомого). Их функции

заключаются в создании сигналов EOI для обоих контроллеров в ответ на прерывания, содержательная обработка которых не предусмотрена в программе. Сигнал EOI, как было описано в статье 49, разблокирует нижележащие уровни прерываний, которые аппаратно блокируются контроллером, когда процессор начинает обслуживать данное прерывание. Строго говоря, в нашем случае необходимости в этих обработчиках нет, во-первых, потому, что эти прерывания реально не возникают, а во-вторых, потому, что обрабатываемые нами прерывания от таймера и клавиатуры имеют наивысшие приоритеты и любое другое прерывание их заблокировать не может. Однако рассматриваемые фрагменты носят общий характер и могут оказаться полезными в других условиях. Тексты процедур master и slave приведены ниже.

```
master  proc
;Процедура для ведущего контроллера
        push    AX          ;Сохраним используемый регистр
        mov     AL,20h      ;Сигнал EOI
        out     20h,AL
        pop     AX          ;Восстановим регистр
        db      66h         ;Возврат
        iret                ;в программу
master  endp
slave   proc
;Процедура для ведомого контроллера
        push    AX          ;Сохраним используемый регистр
        mov     AL,20h      ;Сигнал EOI для
        out     0A0h,AL     ;ведомого контроллера
        mov     AL,20h      ;Сигнал EOI для
        out     20h,AL      ;ведущего контроллера
        pop     AX          ;Восстановим регистр
        db      66h         ;Возврат
        iret                ;в программу
slave   endp
```

Предполагая обрабатывать все аппаратные прерывания, мы должны инициализировать не только ведущий, но и ведомый контроллер:

```
;Инициализация ведомого контроллера. Базовый вектор теперь 28h
        mov     DX,0A0h     ;Порт контроллера
        mov     AL,11h      ;СКИ1: будет СКИ3
        out     DX,AL
        jmp     $+2         ;Задержка
        inc     DX          ;Второй порт контроллера
        mov     AL,28h      ;СКИ2: базовый вектор
        out     DX,AL
        jmp     $+2
        mov     AL,2        ;СКИ3: ведомый подключен к уровню 2
        out     DX,AL
        jmp     $+2
        mov     AL,1        ;СКИ4: 80x86, требуется EOI
        out     DX,AL
        jmp     $+2
```

Реинициализация контроллеров после возврата в реальный режим в программе не предусмотрена, поскольку механизм возврата (сброс процессора и запуск программ BIOS) предполагает их реинициализацию программами BIOS. Необходимо только установить стандартные для компьютера значения масок прерываний.

Статья 66

Работа с расширенной памятью

До сих пор мы имели дело с сегментами небольшого размера (не более 64 Кбайт), размещаемыми в обычной памяти. В настоящей статье будет рассмотрена работа с большими сегментами данных, находящимися в расширенной памяти, т.е. за пределами мегабайтного адресного пространства. Поскольку описание больших сегментов данных имеет некоторую специфику, мы начнем с более подробного, чем прежде, рассмотрения дескриптора памяти. Вообще дескрипторы могут быть трех типов: дескрипторы памяти, системные дескрипторы и шлюзы. Форматы дескрипторов памяти и системных практически совпадают, формат же шлюза несколько отличается. Здесь мы рассмотрим только дескрипторы памяти, служащие для описания сегментов памяти, с которыми работает программа. Формат дескриптора памяти соответствует структуре descr, использовавшейся во всех программах этого раздела, и изображен на рис. 66.1.

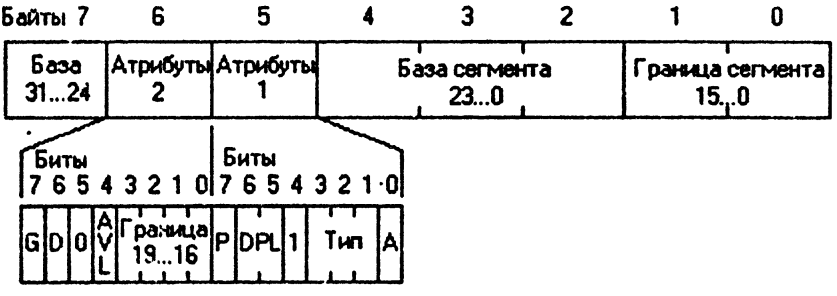


Рис. 66.1. Формат дескриптора сегмента памяти.

Как видно из рисунка, дескриптор занимает 8 байтов. В байтах 2...4 и 7 записывается линейный сегментный адрес базы сегмента. Поскольку базовый адрес имеет 32 бита, он может быть назначен в любой точке 4-гигабайтного адресного пространства (естественно, только там, где имеется реальная память или другие адресуемые объекты, например, видеобуфер). В байтах 0-1 записываются младшие 16 бит границы сегмента, а в младшие четыре бита байта атрибутов 2 - оставшиеся биты 16...19. Таким образом, граница описывается 20-ю битами, и ее численное значение не может превышать 1М. Однако, как уже упоминалось ранее, единицы, в которых задается граница, можно изменять, что осуществляется с помощью бита дробности G (бит 7 байта атрибутов 2). Если G=0, граница указывается в байтах; если 1 - в блоках по 4 Кбайт. Таким образом, размер сегмента можно задавать с точностью до байта, но тогда он не может быть больше 1 Мбайт; если же установить G=1, то сегмент может достигать 4 Гбайт, однако его размер будет кратен 4 Кбайт. При этом база сегмента и в том, и в другом случае задается с точностью до байта.

При выделении программе сегментов большого размера (как это предусмотрено в приведенном ниже примере) следует иметь в виду особенность вычисления процессором значения границы. Если G=1, истинная граница сегмента определяется следующим образом:

Граница сегмента = граница в дескрипторе * 4K + 4095

Таким образом, сегмент всегда простирается до конца последнего 4K-байтного блока. Пусть, например, базовый адрес сегмента равен 0, граница тоже равно 0, и бит дробности установлен. Тогда истинная граница сегмента будет равна

0*4K+4095=4095

т.е. сегмент будет занимать диапазон адресов 0...4095 и иметь размер ровно 4 Кбайт. Если установить значение границы, равное 1, сегмент будет иметь размер ровно 8 Кбайт и т.д.

Рассмотрим теперь атрибуты сегмента, которые занимают два байта дескриптора.

Бит A (Accessed, было обращение) устанавливается процессором в тот момент, когда в какой-либо сегментный регистр загружается селектор данного сегмента. Анализируя биты обращения различных сегментов, программа (в частности, операционная система защищенного режима) может судить о том, было ли обращение к данному сегменту после того, как программа сбросила бит A.

Тип сегмента занимает 3 бита (иногда бит A включают в поле типа, и тогда тип занимает 4 бит) и может иметь 8 значений, перечисленных в табл. 66.1.

Как видно из табл. 66.1, тип определяет правила доступа к сегменту. Указав для некоторого сегмента данных тип 0, мы защитим его от за-

дификации. При попытке записи в такой сегмент возникнет исключение общей защиты. Сегменту команд обычно присваивается тип 4, и тогда его можно только исполнять. Таким образом, в защищенном режиме не предусмотрено составление самомодифицирующихся программ.

Таблица 66.1. Значения поля типа дескриптора сегмента памяти

Тип	Характеристики сегмента
0	Разрешено только чтение (сегмент данных)
1	Разрешены чтение и запись (сегмент данных)
2	Расширение вниз, разрешено только чтение (сегмент стека)
3	Расширение вниз, разрешены чтение и запись (сегмент стека)
4	Разрешено только исполнение (сегмент команд)
5	Разрешены исполнение и чтение (сегмент команд)
6	Разрешено только исполнение (подчиненный сегмент)
7	Разрешены исполнение и чтение (подчиненный сегмент)

Подчиненные, или согласованные (conforming) сегменты обычно используются для хранения подпрограмм общего пользования; для них не действуют общие правила защиты программ друг от друга.

Сегменты с расширением вниз, т.е. в сторону меньших адресов, используются для организации стеков, которые по ходу выполнения программы приходится расширять. Для относительно простых программ размеры сегментов обычно фиксированы и для стека можно использовать обычный сегмент данных (естественно, с разрешением как чтения, так и записи).

Бит 4 байта атрибутов 1 является идентификатором сегмента. Если он равен 1, как это показано на рис. 66.1, дескриптор описывает сегмент памяти. Значение этого бита 0 характеризует дескриптор системного сегмента.

Поле DPL (Descriptor Privilege Level, уровень привилегий дескриптора) служит для защиты программ друг от друга. Уровень привилегий может принимать значения от 0 (максимальные привилегии) до 3 (минимальные). Программам операционной системы обычно назначается уровень 0, прикладным программам - уровень 3, в результате чего исключается возможность некорректным программам разрушить операционную систему. В наших примерах операционная система защищенного режима отсутствует, в некоторой степени программа сама выполняет функции операционной системы, поэтому всем сегментам наших программ назначается наивысший (нулевой) уровень привилегий, что открывает доступ ко всем средствам защищенного режима.

Бит P, как уже отмечалось ранее, говорит о присутствии сегмента в памяти. В основном он используется в тех случаях, когда общий размер

программы (или программ в случае многозадачного режима) превышает объем наличной памяти, и часть сегментов программ хранится на диске. Тогда операционная система защищенного режима с помощью этого бита определяет, находится ли требуемый сегмент в памяти, и при необходимости загружает его с диска. Перед выгрузкой ненужного сегмента на диск бит P сбрасывается. Для наших примеров естественно во всех сегментах устанавливать бит P.

Младшая половина байта атрибутов 2 занята старшими битами границы сегмента. Бит AVL (от Available, доступный) не используется и не анализируется процессором и предназначен для использования прикладными программами.

Бит D (Default, умолчание) определяет действующий по умолчанию размер для операндов и адресов. Он изменяет характеристики сегментов двух типов: исполняемых и стека. Если бит D сегмента команд равен 0, в сегменте по умолчанию используются 16-битовые адреса и операнды, если 1 - 32-битовые.

Атрибут сегмента, действующий по умолчанию, можно изменить на противоположный с помощью префиксов замены размера операнда (66h) и замены размера адреса (67h). Таким образом, для сегмента с D=0 префикс 66h перед некоторой командой заставляет ее рассматривать свои операнды, как 32-битовые, а для сегмента с D=1 тот же префикс 66h, наоборот, сделает операнды 16-битовыми. В некоторых случаях транслятор сам включает в объектный модуль необходимые префиксы, в других случаях их приходится вводить в программу "вручную".

Поскольку мы запускаем наши программы в реальном режиме под управлением MS-DOS, естественно устанавливать D=0. Это, однако, не препятствует использованию в программе 32-битовых регистров (EAX, ECX и т.д.). Если транслятор встречается с командой, в качестве операнда которой используется 32-разрядный регистр, он включает в код команды префикс замены размера операнда 66h. Поэтому команды с явным обращением к 32-битовым операндам транслируются правильно. В то же время при использовании команды iret в защищенном режиме префикс 66h приходится включать в программу в явном виде, чтобы процессор, выполняя эту команду, снял со стека не три слова, как обычно, а 6 слов (три двойных слова). Без префикса команда iret будет выполняться неправильно.

Префикс замены размера адреса 67h необходимо указывать, например, перед командой loop, если в качестве счетчика цикла используется не CX, а ECX. При отсутствии префикса команда loop (в сегменте, где по умолчанию используется 16-битовая адресация) будет выполнять цикл CX, а не ECX раз.

Сегменты стека, адресуемые через регистр SS, с помощью бита D определяют, какой регистр использовать в качестве указателя стека в

командах push и pop. Если D=0, используется регистр SP, если D=1, то ESP.

Сегменты команд, предназначенные целиком для использования в защищенном режиме, транслируются с описателем размера адресов и операндов use32, а описывающие их дескрипторы должны иметь бит D=1.

Теперь мы можем расшифровать значения атрибутов, присвоенных нами сегментам наших программ (рис. 66.2).

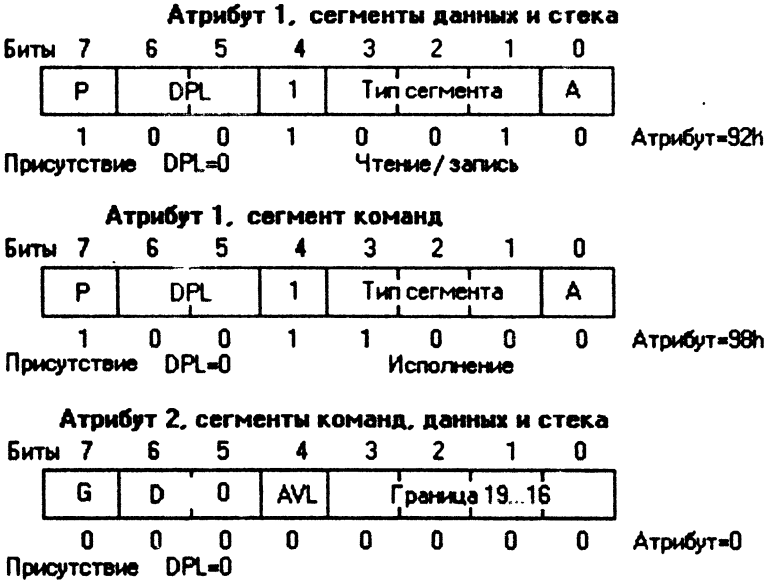


Рис. 66.2. Расшифровка значений атрибутов сегментов в программных примерах.

Рассмотрим теперь формат дескрипторов, из которых строится таблица прерываний IDT, и которые носят название шлюзов. Как уже отмечалось, в таблицу дескрипторов прерываний могут входить шлюзы трех типов: ловушек, прерываний и задач. В рассмотренных ранее примерах таблица дескрипторов прерываний состояла из шлюзов прерываний и ловушек, описывающих обработчики аппаратных прерываний и исключений. Формат шлюза изображен на рис. 66.3.

Если основной частью содержимого сегмента памяти был его линейный адрес, то в шлюзе вместо линейного адреса указывается полный трехсловный адрес обработчика, действующий в защищенном режиме и состоящий из селектора и смещения. Смещение имеет 32 бита и занимает в дескрипторе два поля - байты 0-1 и 6-7. Селектор имеет 16 бит и

занимает байты 2-3. Байт 4 в шлюзах ловушек, прерываний и задач не используется.

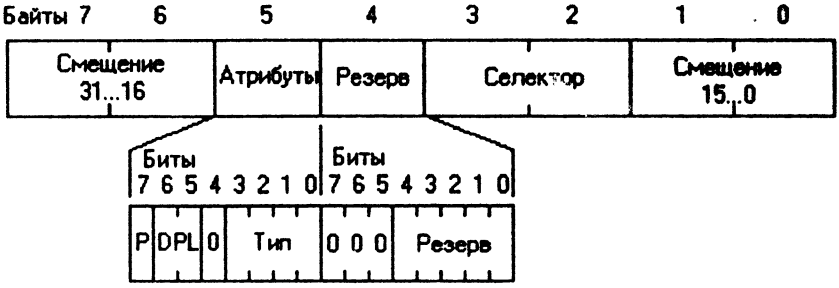


Рис. 66.3. Формат шлюзов, входящих в таблицу дескрипторов прерываний.

Байт атрибутов имеет такую же структуру, как и в дескрипторах памяти и включает тип, идентификатор дескриптора (бит 4), уровень привилегий дескриптора DPL и бит присутствия P. Тип дескриптора может принимать 16 различных значений (табл. 66.2), хотя, как уже отмечалось, в IDT допустимо описывать только шлюзы задач, прерываний и ловушек.

Таблица 66.2. Значения поля типа для системных дескрипторов и шлюзов

Тип	Назначение дескриптора
0	Неопределено
1	Свободный сегмент состояния задачи (TSS) 80286
2	LDT
3	Занятый сегмент состояния задачи (TSS) 80286
4	Шлюз вызова 80286
5	Шлюз задачи
6	Шлюз прерываний 80286
7	Шлюз ловушки 80286
8	Неопределено
9	Свободный сегмент состояния задачи (TSS) 80386 и 486
Ah	Неопределено
Bh	Занятый сегмент состояния задачи (TSS) 80386 и 486
Ch	Шлюз вызова 80386 и 486
Dh	Неопределено
Eh	Шлюз прерываний 80386 и 486
Fh	Шлюз ловушки 80386 и 486

В примерах предыдущих статей использовались дескрипторы типов Eh для описания обработчиков аппаратных прерываний и Fh для описания обработчиков исключений.

В примере 66.1 в расширенной памяти создается сегмент размером 2 Мбайт, который заполняется натуральным рядом чисел (512K четырехбайтовых чисел) с визуальным контролем заполнения. За основу взята программа 65.1, из которой изъяты строки, относящиеся к аппаратным прерываниям (дескрипторы аппаратных прерываний, инициализация контроллера прерываний, сами обработчики аппаратных прерываний и проч).

Пример 66.1. Работа с расширенной памятью

```
include debug.mac           ;(1)
.386P                       ;(2)
;Структура для описания дескрипторов сегментов
descr   struc               ;(3)
        ...
descr   ends                ;(4)
;Структура для описания шлюзов ловушек
trap    struc               ;(5)
        ...
trap    ends                ;(6)
data    segment use16       ;(7)
;Таблица глобальных дескрипторов GDT
gdt_0   label word          ;(8)Начало GDT
;Поля дескрипторов gdt_null, gdt_data, gdt_code, gdt_stack и
;gdt_screen, а также дескрипторов сегментов данных, команд, стека
;и видеобуфера
        ...
gdt_himem descr <511,0,10h,92h,80h>;(9)Селектор 40, сегмент
                            ;расширенной памяти
gdt_size=$-gdt_0            ;(10)Размер GDT
;Таблица дескрипторов прерываний IDT
idt     label word          (11)Начало IDT
;Дескрипторы исключений
trap    10 dup (<dummy_exc>);(12)Общий дескриптор исключений
trap    <exc_0ah>           ;(13)Дескриптор исключения 10
trap    <exc_0bh>           ;(14)Дескриптор исключения 11
trap    <exc_0ch>           ;(15)Дескриптор исключения 12
trap    <exc_0dh>           ;(16)Дескриптор исключения 13
trap    <exc_0eh>           ;(17)Дескриптор исключения 14
trap    17 dup (<dummy_exc>);(18)Дескрипторы зарезервированных
                            ;исключений
idt_size=$-idt              ;(19)Размер IDT
;Поля данных программы
pdescr  dq    0             ;(20)Псевдодескриптор
mes db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$';(21)
tblhex db '0123456789ABCDEF'  ;(22)Таблица bin-hex
string  db '**** ****-**** ****-**** ****';(23)Шаблон для вывода
;           0    5    10   15   20   25  Позиции в шаблоне
len=$-string                ;(24)
number  db '???? ????'      ;(25)Диагностическая строка для
                            ;динамического контроля заполнения
                            ;расширенной памяти
home_sel dw    home         ;(26)Адрес возврата из исключения
         dw    10h          ;(27)Сегмент команд
data_size=$-gdt_null        ;(28)Размер сегмента данных
```

```
data    ends            ;(29)
text    segment 'code' use16;(30)
        assume CS:text,DS:data;(31)
begin   label word      ;(32)Начало сегмента команд
;Обработчики исключений недопустимого сегмента состояния задачи
;exc_0ah, исключения отсутствия сегмента exc_0bh, исключения
;ошибки обращения к стеку exc_0ch, исключения общей защиты
;exc_0dh, исключения страничного нарушения exc_0eh, остальных
;исключений dummy_exc (см. предыдущие примеры)
        ...
main    proc            ;(33)Начало главной процедуры
;Вычислим линейные адреса сегментов данных, команд и стека и
;занесем их в таблицу глобальных дескрипторов
        ...
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        ...
;Подготовимся к переходу в защищенный режим (запрет аппаратных
;прерываний и NMI, засылка байта отключения в порт 71h и адрес
;возврата в ячейки 40h:67h). Все, как в примере 65.1 (предл. 105-
;114), за исключением значения байта отключения - 0Ah вместо 05h
        ...
;Таблица прерываний уже заполнена на этапе трансляции
;Загрузим IDTR
        ...
;Откроем линию A20
        mov     AL,0D1h     ;(34)Команда управления
        out     64h,AL      ;(35)линией A20
        mov     AL,0DFh     ;(36)Код открытия
        out     60h,AL      ;(37)линии A20
;Переходим в защищенный режим
        ...
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
        ...
continue:
;Делаем адресуемыми данные и стек
        ...
;Инициализируем ES селектором видеобуфера
        mov     AX,32       ;(38)Селектор сегмента видеобуфера
        mov     ES,AX       ;(39)
;Заполним мегабайт расширенной памяти
        mov     AX,40       ;(40)Селектор сегмента
        mov     GS,AX       ;(41)в расширенной памяти
        mov     EAX,0       ;(42)Первое число-заполнитель
        mov     EBX,0       ;(43)Индекс в сегменте
        mov     ECX,80000h ;(44)80000h*4=200000h=2Мбайт
fill:   mov     GS:[EBX],EAX;(45)Заполняем память!
;Выведем диагностическую строку о заполнении старшей памяти
        push    EAX         ;(46)Сохраним EAX и CX на время
        push    CX          ;(47)диагностических действий
        mov     SI,offset number+5;(48)Сюда младшую половину EAX
        debug               ;(49)
        shr     EAX,16      ;(50)Обменяем половины EAX
        mov     SI,offset number;(51)Сюда старшую половину EAX
        debug               ;(52)
        mov     SI,offset number;(53)DS:SI=адрес строки
        mov     CX,9        ;(54)CX=длина строки
```

```
        mov     AH,43h      ;(55)Атрибут
        mov     DI,1040     ;(56)ES:DI=позиция на экране
scrh:   lodsb               ;(57)Заберем из строки байт
        stosw               ;(58)И выведем в видеобуфер слово
        loop    scrh        ;(59)Повторить CX раз
        pop     CX          ;(60)Восстановим
        pop     EAX         ;(61)регистры
        add     EBX,4       ;(62)Смещение индекса в памяти
        inc     EAX         ;(63)Инкремент числа-заполнителя
        db      67h         ;(64)Команда loop должна работать с ECX!
        loop    fill_1      ;(65)Вспомогательный переход
        jmp     go          ;(66)Выход из цикла после его окончания
fill_1: jmp     fill        ;(67)Переход на начало цикла
;Проверим заполнение сегмента в расширенной памяти
go:     mov     EBX,0       ;(68)Смещение первого 4-байтового
                            ;слова сегмента в расширенной памяти
        mov     EAX,GS:[EBX];(69)Получили в EAX первое число
        mov     SI,offset string+15;(70)Перекодировка
        debug               ;(71)и помещение
        shr     EAX,16      ;(72)этого слова
        mov     SI,offset string+10;(73)в строку
        debug               ;(74)string
        mov     EBX,2FFFFCh;(75)Смещение последнего слова
                            ;сегмента в расширенной памяти
        mov     EAX,GS:[EBX];(76)В EAX последнее число
        mov     SI,offset string+25;(77)Перекодировка
        debug               ;(78)и помещение
        shr     EAX,16      ;(79)этого слова
        mov     SI,offset string+20;(80)в строку
        debug               ;(81)string
        mov     AX,0FFFFh   ;(82)Диагностическое значение
home:   mov     SI,offset string;(83)
        debug               ;(84)
;Выведем на экран диагностическую строку
        ...                 ;См. пример 63.1, предл. 101...107
;Закроем линию A20
        mov     AL,0D1h     ;(85)Команда управления
        out     64h,AL      ;(86)линией A20
        mov     AL,0DDh     ;(87)Код открытия
        out     60h,AL      ;(88)линии A20
;Переключим режим процессора
        mov     AL,0FEh     ;(89)Переходим в реальный режим
        out     64h,AL      ;(90)засылкой кода FEh в порт 64h
        hlt                 ;(91)Останов в ожидании сброса
;Теперь процессор снова работает в реальном режиме
return:
;Восстановим операционную среду реального режима (DS, SS:SP)
        ...
;Разрешим аппаратные и немаскируемые прерывания
        ...
;Проверим выполнение функций DOS и завершим программу
        ...
```

Желая образовать дополнительный сегмент объемом 2 Мбайт в расширенной памяти, мы должны прежде всего предусмотреть описывающий его дескриптор в таблице глобальных дескрипторов:

```
gdt_himem descr <511,0,10h,92h,80h,>;(9)Селектор 40
```

Будучи расположен в таблице GDT на шестом месте, этот дескриптор получает индекс 5, что соответствует селектору 40.

Линейный адрес первого байта расширенной памяти равен 100000h. Из этого адреса младшие 16 бит (0000h) записываются в поле base_l, следующие 8 бит, содержащие 10h, - в поле base_m, и старшие 8 бит (00h) - в поле base_h.

Поскольку размер сегмента превышает 1 Мбайт, его границу следует указывать в блоках по 4 Кбайт. Поэтому мы устанавливаем в 1 бит дробности G в байте атрибутов 2, который, таким образом, принимает значение 80h. Размер сегмента (2 Мбайт) составляет 512 блоков по 4 Кбайт, поэтому значение границы составляет 511. Байт атрибутов 1, как и для других сегментов памяти, принимает значение 92h (для чтения и записи, присутствует).

В поля данных программы включена символьная строка number, в которую по мере заполнения расширенной памяти числами будет выводиться текущее число. Это даст возможность контролировать ход заполнения памяти и заодно проверить правильность содержимого последней заполненной ячейки.

Перед переходом в защищенный режим (или после перехода в него) следует открыть линию A20, т.е. адресную линию, на которой устанавливается единичный уровень сигнала, если происходит обращение к мегабайтам адресного пространства с номерами 1, 3, 5 и т.д. (первый мегабайт имеет номер 0). В реальном режиме линия A20 заблокирована, и если значение адреса выходит за пределы FFFFFh, выполняется его циклическое оборачивание (линейный адрес 100000h превращается в 00000h, адрес 100001h - в 00001h и т.д.). Открытие (разблокирование) линии A20 выключает механизм циклического оборачивания адреса, что позволяет адресоваться к расширенной памяти. Управление блокированием линии A20 осуществляется через порт 64h, куда сначала следует послать команду D1h управления линией A20, а затем - код открытия (DFh). Эти действия выполняются в предложениях 34...37.

Переход в защищенный режим и действия по инициализации сегментных регистров выполняются обычным образом.

Заполнение расширенной памяти начинается с предложения 40. В свободный сегментный регистр GS заносится селектор сегмента в расширенной памяти и инициализируются регистры EAX, EBX и ECX (число шагов в цикле превышает 64 K, поэтому требуется 32-битовый регистр ECX). В предложении 53 число-заполнитель отправляется в расширенную память. Для динамического контроля хода заполнения памяти в каждом шаге цикла очередное число выводится на экран (предложения 46...61). Далее выполняется смещение индекса в указателе, инкремент числа-заполнителя и возврат в начало цикла.

Для того, чтобы команда loop использовала в своей работе регистр ECX, а не CX (вспомним, что мы установили для сегмента команд D=0 и назначили тем самым по умолчанию 16-битовые адреса и операнды), перед ней в программу включен код префикса замены размера адреса (предложение 64). Другая сложность возникла из-за того, что в цикле оказалось больше 128 байтов, а команда loop может осуществлять только короткие переходы в диапазоне +127...-128 байтов. Поэтому командой loop (предложение 65) выполняется переход не на начало цикла, а на вспомогательную строку fill_1, откуда командой безусловного перехода управление может быть передано уже в любую точку программы. Команда безусловного перехода jmp в предложении 66 позволяет обойти предложение 67 после завершения цикла.

В программе предусмотрено контрольное чтение расширенной памяти и вывод в диагностическую строку string прочитанных чисел. В предложениях 68...74 читается и выводится первое 4-байтовое слово из сегмента расширенной памяти (0000 0000h), в предложениях 75...77 - последнее 4-байтовое слово двухмегабайтного сегмента (0007 FFFFh). Эта проверка может оказаться весьма полезной. Действительно, если программа из-за неправильно написанных строк цикла обратится к памяти за пределами объявленного в таблице глобальных дескрипторов сегмента, возникнет исключение общей защиты, и мы это сразу обнаружим. Однако процессор не реагирует на отсутствие физической памяти по указанному в команде адресу. Поэтому если попытаться заполнить больше памяти, чем есть в компьютере (предварительно установив соответственно границу сегмента в дескрипторе), то программа будет работать так, как если бы эта память имелась, хотя, конечно, реально числа записываться в отсутствующую память не будут. Чтение из памяти последнего записанного в нее числа позволит убедиться в том, что это число действительно записано в память.

Перед переходом в реальный режим следует закрыть линию A20, для чего в порт 64h посылается команда D1h управления линией A20, а затем - код закрытия линии (DDh). Эти действия выполняются в предложениях 92...95. Завершение программы выполняется так же, как и в предыдущей статье, за одним небольшим исключением (см. комментарии в тексте программы).

Читатель может модифицировать программу, увеличив или уменьшив размер сегмента в расширенной памяти и, соответственно, число шагов в цикле заполнения, настроив ее под конкретную конфигурацию своего компьютера.

Статья 67

Переключение задач

Важнейшей особенностью защищенного режима является возможность параллельного выполнения нескольких вычислительных задач с надежной защитой их друг от друга (чем и объясняется название режима). Использование таблиц локальных дескрипторов позволяет разнести физические адресные пространства отдельных задач без права их обращения к "чужой" памяти, а система привилегий, встроенная в архитектуру процессора, предотвращает несанкционированный вызов задачами защищенных процедур (например, программ операционной системы) или обращение к устройствам ввода-вывода. Однако организация мультизадачного режима, вообще говоря, не обязательно предполагает использование механизмов защиты по привилегиям. В простых и достаточно распространенных случаях параллельное выполнение нескольких задач осуществляется на одном (обычно наиболее приоритетом) уровне привилегий, защита же адресных пространств или процедур осуществляется путем аккуратного написания программ комплекса. Очевидно, что методически проще сначала рассмотреть (весьма непростые) методики переключения задач, и лишь затем развить эти методики включением в них механизмов защиты по привилегиям. В настоящей и нескольких последующих статьях будут рассмотрены принципы организации мультизадачного режима.

Под задачей (task) понимают выполняемую на компьютере программу или ее фрагмент, имеющий относительно самостоятельное значение. В частности, задачей можно назвать всю сколь угодно сложную программу, и тогда мультизадачность обозначает параллельное выполнение нескольких, возможно не связанных друг с другом программ. Однако задачей может быть и достаточно самостоятельная часть программы. Например, обработчики аппаратных прерываний уместно назвать задачами: для них как раз характерно выполнение параллельно и независимо от основной части программы. Вообще ничто не мешает включить в состав одной программы несколько самостоятельных участков, переключение между которыми может осуществляться периодически или по каким-то событиям: нажатиям определенных клавиш, прерываниям от диска и т.д. Эти фрагменты могут образовывать отдельные сегменты команд, но могут входить в состав единого программного

сегмента в виде, например, процедур. Именно такие программы будут рассмотрены в качестве примеров этого раздела.

Организация мультизадачного режима, опирается на следующие аппаратные и программные средства, о которых пока не было речи:

сегмент состояния задачи (Task State Segment, TSS);

дескриптор сегмента состояния задачи;

регистр задачи (Task Register, TR);

дескриптор шлюза задачи (Task gate).

Сегмент состояния задачи (TSS) представляет собой поле данных, либо включаемое в состав сегмента данных программы, либо образующее отдельный сегмент небольшого размера. Каждая задача, участвующая в процедуре переключения, должна иметь свой TSS; именно наличие TSS делает данный программный объект задачей. Сегменты состояния задач служат для хранения, при переключениях задач, их текущих контекстов, т.е. содержимого аппаратных регистров процессора и другой важной информации. Поскольку TSS представляется процессору отдельным сегментом, ему должен соответствовать дескриптор. В отличие от обычных сегментов данных, TSS описывается не дескриптором сегмента памяти, а системным дескриптором, который к тому же может находиться только в таблице глобальных дескрипторов. Системные дескрипторы имеют тот же формат, что и дескрипторы памяти (см. рис. 66.1), отличаясь только кодом типа. Для процессоров 386 и 486 дескрипторы TSS имеют код 9. На рис. 67.1 приведен для справки формат дескриптора TSS для процессора 486.

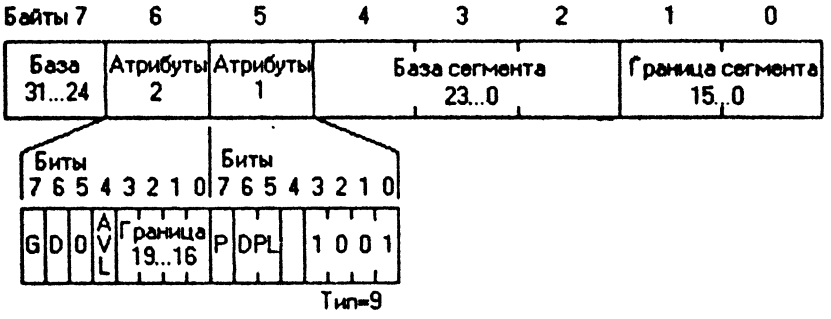


Рис. 67.1. Формат системного дескриптора, описывающего TSS для МП 386 и 486.

В зависимости от геометрического места расположения дескриптора TSS в таблице дескрипторов, ему соответствует тот или иной селектор. Селектор TSS текущей (активной) задачи должен быть загружен в регистр задачи TR. Для исходной, главной задачи эта загрузка осуществляется программно с помощью предназначенной для этого коман-

ды ltr (load task register, загрузка регистра задач); при переключении на новую задачу программа передает процессору селектор нового TSS и перезагрузку регистра TR осуществляет процессор в ходе переключения задач.

Переключение на новую задачу осуществляется командой дальнего вызова call dword ptr или, в некоторых случаях, дальнего перехода jmp dword ptr. В качестве аргумента этих команд указывается двухсловное поле, в первом слове которого записывается селектор требуемого TSS. Существует и другой способ переключения - не через селектор TSS, а через шлюз задачи (дескриптор шлюза с кодом типа, равным 5). В этом случае селектор требуемого TSS указывается в поле для селектора в шлюзе задачи. Переключение через шлюз задачи имеет то преимущество, что его можно выполнить по аппаратному прерыванию, так как, в отличие от дескриптора сегмента состояния задачи TSS, шлюз задачи можно расположить в таблице дескрипторов прерываний (см. текст к табл. 66.2).

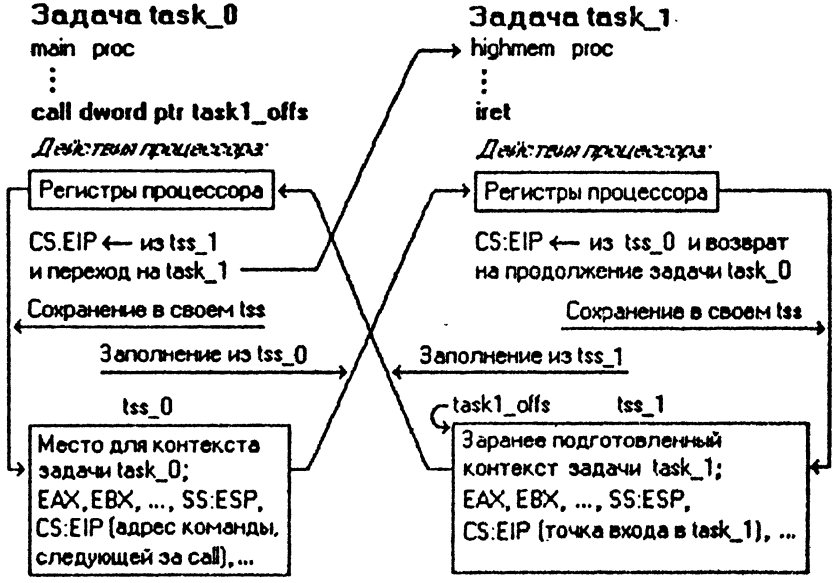


Рис. 67.2. Основные действия при переключении задач.

В процессе переключения на новую задачу (рис. 67.2) процессор сохраняет контекст текущей задачи в ее TSS и загружает новый контекст (включая селектор и относительный адрес точки входа в задачу) из TSS новой задачи.

Задача, на которую осуществляется переключение, должна в этом случае завершаться командой iret, которая обрабатывается процессором совсем не так, как iret обычного обработчика прерывания. В случае переключения задач процессор по команде iret выполняет обратное перемещение контекстов - контекст завершающейся задачи (на момент ее завершения) сохраняется в ее TSS, а в регистры процессора из TSS исходной задачи загружается сохраненный там контекст, соответствующий моменту переключения на вторую задачу. Таким образом, TSS можно рассматривать, как функциональный аналог стека, который тоже используется для сохранения содержимого регистров и другой информации в обработчиках прерываний и подпрограммах. Однако сохранение в стеке и восстановление из него выполняются с помощью последовательностей команд процессора, а сохранение и восстановление контекстов с помощью TSS осуществляется процессором аппаратно в процессе переключения задач.

Поля TSS исходной задачи заполняются процессором при первом переключении на новую задачу (чтобы можно было вернуться в исходную); программист может не заботиться о его содержимом. Другое дело TSS задачи, на которую осуществляется переключение. В TSS содержится такая важная для выполнения задачи информация, как адрес точки входа, а также исходное содержимое сегментных регистров и регистров общего назначения. По крайней мере некоторые из этих полей должны быть заполнены в исходной задаче еще перед переключением на новую. Рассмотрим кратко содержимое TSS (рис. 67.3).

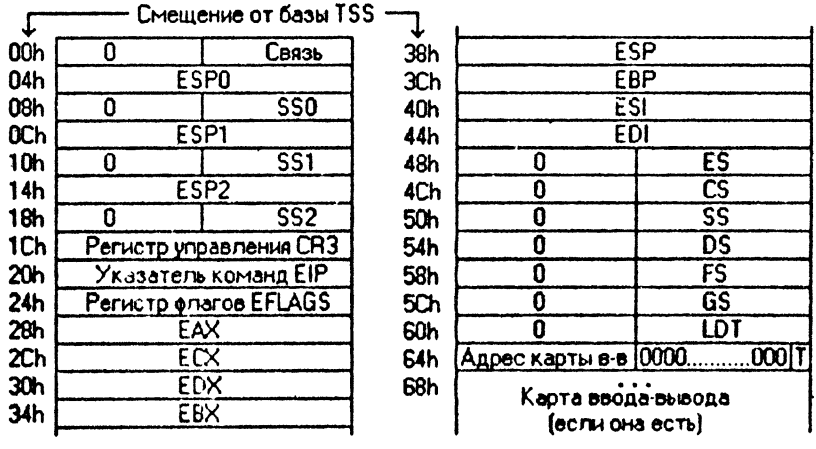


Рис. 67.3. Формат сегмента состояния задачи для МП 386 и 486.

Сегмент состояния задачи имеет размер минимум 104 байт. В самом начале TSS предусмотрено 16-битовое поле связи, используемое при переключении задач. Для исходной, главной задачи (назовем ее task_0) его содержимое не имеет значения. При переключении на новую задачу (task_1) процессор заносит в поле связи TSS новой задачи селектор TSS исходной задачи, чем создается связь между новой и старой задачами. Если новая задача, в свою очередь, переключается на следующую задачу (task_2), в поле связи TSS этой следующей задачи процессор заносит селектор TSS предыдущей задачи и т.д. В результате создается связный список вложенных задач (рис. 67.4).

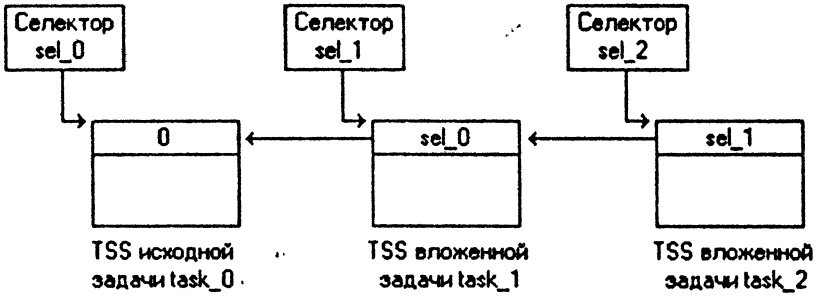


Рис. 67.4. Цепочка связанных TSS.

Команда iret, которой должна завершаться каждая вложенная задача, выполняет обратное переключение задач. В процессе этой операции процессор извлекает из поля связи TSS последней вложенной задачи селектор TSS предыдущей по порядку вложенности задачи и передает ей управление, восстановив перед этим из TSS этой задачи ее контекст.

По адресам 04h, 0Ch и 14h относительно базы TSS располагаются кадры стеков уровней привилегий 0, 1 и 2. Содержимое этих полей загружается в регистры SS и ESP, если при переключении задачи происходит смена уровня привилегий. Задачи наших примеров будут работать на одном (нулевом) уровне привилегий; в этом случае поля кадров стеков не используются, а регистры SS и ESP инициализируются содержимым ячеек TSS с адресами 50h и 38h.

Регистр управления CR3 (поле TSS с адресом 1Ch) содержит базовый физический адрес каталога страниц и используется, если включено страничное преобразование. Наличие в TSS поля для регистра CR3 позволяет иметь для каждой задачи свой каталог страниц и, соответственно, свои таблицы отображения виртуальных страниц программы на физические адреса памяти. В наших примерах страничное преобразование выключено и регистр CR3 не используется.

Двухсловное поле TSS с адресом 20h предназначено для хранения значения указателя команд EIP. В TSS исходной задачи это поле при переключении на новую задачу заполняется процессором адресом команды, следующей за командой переключения, т.е. адресом возврата. В TSS задачи, на которую планируется осуществить переключение, поле для EIP должно быть заполнено программно смещением точки входа в задачу. При этом следует иметь в виду, что при возврате в старую задачу командой iret процессор записывает в поле для EIP завершившейся задачи в качестве "адреса возврата" адрес команды, следующей iret, т.е. адрес уже за пределами задачи. Если планируется повторное переключение на эту задачу, перед каждым переключением необходимо восстанавливать в TSS этой задачи адрес ее точки входа.

Сохранение в TSS исходной задачи текущего содержимого регистра флагов EFLAGS (по адресу 24h) позволяет осуществлять переключение в любой точке задачи без потери ее работоспособности.

Участок TSS с адресами 28h...47h отведен для хранения содержимого регистров общего назначения. При переключении с исходной задачи на новую задачу процессор сохраняет в этих полях TSS исходной задачи текущее содержимое регистров, а при обратном переключении командой iret восстанавливает регистры из этих полей TSS исходной задачи, обеспечивая правильное продолжение ее выполнения. Что же касается TSS новой задачи, то заполнив заранее поля регистров в TSS новой задачи, можно передать ей исходные параметры.

Младшие половины шести двухсловных полей, начиная с адреса 48h, отведены под содержимое сегментных регистров. Эти поля используются точно так же, как и поля регистров общего назначения.

Если новая задача использует таблицу локальных дескрипторов, ее селектор следует занести в TSS по адресу 60h.

Бит 0 слова по адресу 64h используется для отладки переключаемых задач. Если в TSS новой задачи установлен этот бит, то сразу же после переключения генерируется исключение отладки с номером 1. Остальные биты слова по адресу 64h должны быть равны 0.

Слово с адресом 66h содержит смещение битовой карты ввода-вывода, которая, при ее наличии, располагается в TSS по последующим адресам и используется для защиты портов компьютера от несанкционированного доступа. Каждый бит этой карты соответствует одному порту (вся карта, таким образом, может достигать 64 Кбит, или 8 Кбайт). Если бит, закрепленный за некоторым портом, равен 0, задача любого уровня привилегий может обращаться к этому порту. Если бит равен 1, то при обращении к порту задачей с недостаточно высоким уровенем привилегий генерируется исключение общей защиты.

Как уже отмечалось выше, переключение на новую задачу осуществляется обычно с помощью команды дальнего перехода, а возврат в

исходную задачу - командой iret. При этом команда iret должна инициировать довольно сложную процедуру обратного переключения через селектор TSS, хранящийся в поле связи TSS текущей задачи. Однако в обработчиках прерываний и исключений та же команда iret выполняется иначе: она просто снимает со стека три 32-битовых слова (EFLAGS, CS и EIP) и загружает их в соответствующие регистры, обеспечивая возврат из обработчика в прерванную задачу. Каким же образом команда iret определяет, в каком режиме ей надлежит работать?

Режим выполнения команды iret определяется состоянием специального флага NT (Nested Task, вложенная задача), расположенного в бите 14 регистра флагов EFLAGS. Команда iret анализирует состояние флага NT и, если он сброшен, осуществляет обычный возврат из программы обработки прерывания (через стек); если же флаг NT установлен, команда iret инициирует обратное переключение задач через селектор в TSS.

После загрузки компьютера флаг NT находится в установленном состоянии. Однако любое аппаратное прерывание или исключение сбрасывает этот флаг, в результате чего команда iret, завершающая обработчик, выполняется в "облегченном" варианте. То же происходит при выполнении процессором команды программного прерывания int. Поскольку команда iret восстанавливает исходное состояние регистра флагов, после завершения обработчика флаг NT снова оказывается установленным (если, конечно, он не был явным образом сброшен выполняемой программой).

При выполнении процедуры переключения на новую задачу через шлюз задачи или непосредственно через TSS, процессор сохраняет в TSS текущей задачи слово флагов и устанавливает в регистре флагов бит NT (независимо от того, был ли он перед этим сброшен или установлен). Команда iret, завершающая задачу, обнаруживает NT=1 и, вместо осуществления возврата через стек, инициирует механизм обратного переключения задач.

Если вложенная задача, в свою очередь, выполняет переключение на следующую задачу, текущее слово флагов с установленным битом NT сохраняется в TSS текущей задачи. После завершения новой задачи это слово будет возвращено в регистр флагов и, таким образом, задача будет продолжаться с NT=1, что обеспечит ее правильное завершение.

Воспользуемся примером предыдущей статьи для изучения техники переключения задач. Для этого выделим строки заполнения расширенной памяти в отдельную процедуру и оформим ее в качестве задачи. Выполним в главной программе переключение на эту задачу и возврат из нее. Учитывая сложность техники переключения задач, мы приводим программу примера 67.1 почти целиком, хотя значительная часть этой программы текстуально повторяет предыдущие примеры.

;Пример 67.1. Техника переключения задач

```
include debug.mac           ;(1)
.386P                       ;(2)
;Структура для описания дескрипторов сегментов
descr   struc               ;(3)
limit   dw      0           ;(4)Граница (биты 0...15)
base_l  dw      0           ;(5)База, биты 0...15
base_m  db      0           ;(6)База, биты 16...23
attr_1  db      0           ;(7)Байт атрибутов 1
attr_2  db      0           ;(8)Граница (биты 16...19) и атрибуты 2
base_h  db      0           ;(9)База, биты 24...31
descr   ends                ;(10)
;Структура для описания шлюзов ловушек
trap    struc               ;(11)
offs_l  dw      0           ;(12)Смещение обработчика, биты 0...15
sel     dw      16          ;(13)Селектор сегмента команд.
rsrv    db      0           ;(14)Зарезервировано
attr    db      8Fh         ;(15)Присутствие+шлюз ловушки
offs_h  dw      0           ;(16)Смещение обработчика, биты 16...31
trap    ends                ;(17)
data    segment use16       ;(18)
;Таблица глобальных дескрипторов GDT
gdt_0   label word          ;(19)Начало GDT
gdt_null descr <>           ;(20)Селектор 0, нулевой дескриптор
gdt_data descr <data_size-1,,,92h>;(21)Селектор 8, сегмент данных
gdt_code descr <code_size-1,,,98h>;(22)Селектор 16, сегмент команд
gdt_stack descr <255,0,0,92h,0,0>;(23)Селектор 24, сегмент стека
gdt_screen descr <4095,8000h,0Bh,92h>;(24)Селектор 32, сегмент
                            ;видеобуфера
gdt_himem descr <511,0,10h,92h,80h>;(25)Селектор 40,
                            ;сегмент расширенной памяти
gdt_tss_0 descr <103,0,0,89h>;(26)Селектор 48, дескриптор TSS_0
                            ;задачи 0 (главной)
gdt_tss_1 descr <103,0,0,89h>;(27)Селектор 56, дескриптор TSS_1
                            ;задачи 1 (заполнения расширенной
                            ;памяти)
gdt_size=$-gdt_null         ;(28)Размер GDT
idt     label word          ;(29)Начало IDT
trap 10 dup (<dummy_exc>) ;(30)Общий дескриптор исключений
trap    <exc_0ah>           ;(31)Дескриптор исключения 10
trap    <exc_0bh>           ;(32)Дескриптор исключения 11
trap    <exc_0ch>           ;(33)Дескриптор исключения 12
trap    <exc_0dh>           ;(34)Дескриптор исключения 13
trap    <exc_0eh>           ;(35)Дескриптор исключения 14
trap 17 dup (<dummy_exc>) ;(36)Дескрипторы зарезервированных
                            ;исключений
idt_size=$-idt              ;(37)Размер таблицы IDT
pdescr  dq      0           ;(38)Псевдодескриптор
real_sp dw      0           ;(39)Ячейки для хранения кадра
real_ss dw      0           ;(40)стека реального режима
mes db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$';(41)
tblhex  db    . '0123456789ABCDEF' ;(42)Таблица bin-hex
string  db '**** ****_**** ****_**** ****';(43)Шаблон для вывода
;           0    5    10   15   20   25  Позиции в шаблоне
len=$-string                ;(44)
number  db      '???? ????';(45)Диагностическая строка для
```

```
                                ;динамического контроля заполнения
                                ;расширенной памяти
home_sel dw     home            ;(46)Адрес возврата из исключения
         dw     16              ;(47)Сегмент команд
tss_0    db     104 dup (0);(48)Сегмент состояния задачи 0
tss_1    db     104 dup (0);(49)Сегмент состояния задачи 1
task1_offs dw   0               ;(50)Двухсловный адрес для переклю-
task1_sel dw    56              ;(51)чения на задачу 1 через TSS_1
data_size=$-gdt_null            ;(52)Размер сегмента данных
data     ends                   ;(53)
text     segment 'code' use16;(54)
         assume CS:text,DS:data;(55)
begin    label word             ;(56)Начало сегмента команд
;Обработчик исключения недопустимого сегмента состояния задачи TSS
exc_0ah proc                    ;(57)
         pop    EAX             ;(58)AX=код ошибки (нам не нужен)
         pop    EAX             ;(59)AX=IP
         mov    SI,offset string+5;(60)
         debug                  ;(61)
         mov    AX,0Ah          ;(62)Номер исключения
         jmp    dword ptr home_sel;(63)
exc_0ah endp                    ;(64)
;Обработчик исключения отсутствия сегмента
exc_0bh proc                    ;(65)
         pop    EAX             ;(66)AX=код ошибки (нам не нужен)
         pop    EAX             ;(67)AX=IP
         mov    SI,offset string+5 ;(68)
         debug                  ;(69)
         mov    AX,0Bh          ;(70)Номер исключения
         jmp    dword ptr home_sel ;(71)
exc_0bh endp                    ;(72)
;Обработчик исключения ошибки обращения к стеку
exc_0ch proc                    ;(73)
...                             ;Аналогично предыдущим
exc_0ch endp                    ;(74)
;Обработчик исключения общей защиты
exc_0dh proc                    ;(75)
...                             ;Аналогично предыдущим
exc_0dh endp                    ;(76)
;Обработчик исключения страничного нарушения
exc_0eh proc
...                             ; Аналогично предыдущим
exc_0eh endp                    ;(77)
;Обработчик остальных исключений
dummy_exc proc ;                ;(78)
         pop    EAX             ;(79)AX=Код ошибки
         pop    EAX             ;(80)AX=IP
         mov    SI,offset string+5;(81)
         debug                  ;(82)
         mov    AX,1111h        ;(83)Условный код
         jmp    dword ptr home_sel;(84)
dummy_exc endp                  ;(85)
main     proc                   ;(86)Начало главной процедуры
         xor    EAX,EAX         ;(87)Очистим EAX
         mov    AX,data         ;(88)Загрузим в DS сегментный
         mov    DS,AX           ;(89)адрес сегмента данных
;Вычислим 32-битовый, линейный адрес сегмента данных и загрузим его
```

```
;в дескриптор сегмента данных в таблице глобальных дескрипторов
        shl     EAX,4           ;(90)В EAX линейный базовый адрес
        mov     EBP,EAX         ;(91)Сохраним его в EBP для будущего
        mov     BX,offset gdt_data;(92)В BX адрес дескриптора
        mov     [BX].base_l,AX;(93)Загрузим младшую часть базы
        rol     EAX,16          ;(94)Обмен половин EAX
        mov     [BX].base_m,AL;(95)Загрузим среднюю часть базы
;Вычислим 32-битовый линейный адрес сегмента команд и загрузим его
;в дескриптор сегмента команд в таблице глобальных дескрипторов
        xor     EAX,EAX         ;(96)Очистим EAX
        mov     AX,CS           ;(97)Сегментный адрес сегмента команд
        shl     EAX,4           ;(98)Умножим на 16
        mov     BX,offset gdt_code;(99)Адрес дескриптора сегмента
                                ;команд
        mov     [BX].base_l,AX;(100)Загрузим младшую часть базы
        rol     EAX,16          ;(101)Обмен старшей и младшей половин
        mov     [BX].base_m,AL;(102)Загрузим среднюю часть базы
;Вычислим 32-битовый линейный адрес сегмента стека и загрузим его
;в дескриптор сегмента стека в таблице глобальных дескрипторов
        xor     EAX,EAX         ;(103)
        mov     AX,SS           ;(104)
        shl     EAX,4           ;(105)
        mov     BX,offset gdt_stack;(106)
        mov     [BX].base_l,AX;(107)
        rol     EAX,16          ;(108)
        mov     [BX].base_m,AL;(109)
;Вычислим 32-битовый линейный адрес сегмента задачи TSS_0 и
;загрузим его в дескриптор gdt_tss_0 в таблице GDT
        mov     EAX,EBP         ;(110)Линейный адрес сегмента данных
        add     AX,offset tss_0;(111)Прибавим смещение TSS1
        mov     BX,offset gdt_tss_0;(112)Адрес дескриптора
                                ;сегмента команд
        mov     [BX].base_l,AX;(113)Загрузим младшую часть базы
        rol     EAX,16          ;(114)Обмен старшей и младшей половин
        mov     [BX].base_m,AL;(115)Загрузим среднюю часть базы
;Вычислим 32-битовый линейный адрес сегмента задачи TSS_1 и
;загрузим его в дескриптор gdt_tss_1 в GDT
        mov     EAX,EBP         ;(116)
        add     AX,offset tss_1;(117)
        mov     BX,offset gdt_tss_1;(118)
        mov     [BX].base_l,AX;(119)
        rol     EAX,16          ;(120)
        mov     [BX].base_m,AL;(121)
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        mov     dword ptr pdescr+2,EBP;(122)База GDT, биты 0...31
        mov     word ptr pdescr,gdt_size-1;(123)Граница GDT
        lgdt    pdescr          ;(124)Загрузим регистр GDTR
;TSS_0 главной задачи инициализировать не надо. Инициализируем
;сегмент состояния задачи 1, которая будет заполнять расширенную
;память
        mov     word ptr tss_1+4Ch,16;(125)CS
        mov     word ptr tss_1+20h,offset highmem;(126)IP
        mov     word ptr tss_1+50h,24;(127)SS
        mov     word ptr tss_1+38h,128;(128)SP на середине стека
        mov     word ptr tss_1+54h,8;(129)DS
        mov     word ptr tss_1+48h,32;(130)ES
;Подготовимся к переходу в защищенный режим
```

```
        cli                 ;(131)Запрет аппаратных прерываний
        mov     AL,8Fh      ;(132)Запрет NMI
        out     70h,AL      ;(133)Порт КМОП-микросхемы
        jmp     $+2         ;(134)Задержка
        mov     AL,0Ah      ;(135)Байт состояния отключения 0Ah:
        out     71h,AL      ;(136)возврат в программу без
                            ;перепрограммирования контроллеров
        mov     AX,40h      ;(137)Загрузим в область
        mov     ES,AX       ;(138)данных BIOS
        mov     word ptr ES:[67h],offset return;(139)адрес
        mov     word ptr ES:[69h],CS;(140)возврата в программу
;Загрузим IDTR
        mov     word ptr pdescr,idt_size-1;(141)Граница
        xor     EAX,EAX     ;(142)
        mov     AX,offset idt;(143)
        add     EAX,EBP     ;(144)
        mov     dword ptr pdescr+2,EAX;(145)
        lidt    pdescr      ;(146)Загрузка IDTR
;Откроем линию A20
        mov     AL,0D1h     ;(147)Команда управления
        out     64h,AL      ;(148)линией A20
        mov     AL,0DFh     ;(149)Код открытия
        out     60h,AL      ;(150)линии A20
;Переходим в защищенный режим
        mov     EAX,CR0     ;(151)Получим содержимое регистра CR0
        or      EAX,1       ;(152)Установим бит защищенного режима
        mov     CR0,EAX     ;(153)Запишем назад в CR0
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
        db      0EAh        ;(154)Код команды far jmp
        dw      offset continue;(155)Смещение
        dw      16          ;(156)Селектор сегмента команд
continue:                   ;(157)
;Делаем адресуемыми данные
        mov     AX,8        ;(158)Селектор сегмента данных
        mov     DS,AX       ;(159)
;Делаем адресуемым стек
        mov     AX,24       ;(160)Селектор сегмента стека
        mov     SS,AX       ;(161)
;Инициализируем ES селектором видеобуфера
        mov     AX,32       ;(162)Селектор сегмента видеобуфера
        mov     ES,AX       ;(163)
;Инициализируем GS адресом расширенной памяти
        mov     AX,40       ;(164)
        mov     GS,AX       ;(165)
;Загрузим регистр задачи TR селектором TSS основной задачи
        mov     AX,48       ;(166)Селектор TSS_0
        ltr     AX          ;(167)
;Выполним переключение на задачу 1
        call    dword ptr task1_offs;(168)
;Восстановим адрес точки входа в TSS задачи 1
        mov     word ptr tss_1+20h,offset highmem;(169)
;Выполним повторное переключение на задачу 1
        call    dword ptr task1_offs;(170)
;Завершим программу, как и раньше
        mov     AX,0FFFFh   ;(171)Диагностическое значение
home:   mov     SI,offset string;(172)
```

```
        .debug                  ;(173)
;Выведем на экран диагностическую строку
        mov     SI,offset string;(174)
        mov     CX,len          ;(175)
        mov     AH,74h          ;(176)
        mov     DI,1600         ;(177)
scr:    lodsb                   ;(178)
        stosw                   ;(179)
        loop    scr             ;(180)
;Закроем линию A20
        mov     AL,0D1h         ;(181)Команда управления
        out     64h,AL          ;(182)линией A20
        mov     AL,0DDh         ;(183)Код открытия
        out     60h,AL          ;(184)линии A20
;Переключим режим процессора
        mov     AL,0FEh         ;(185)Переходим в реальный режим
        out     64h,AL          ;(186)засылкой кода FEh в порт 64h
        hlt                     ;(187)Останов в ожидании сброса
;Теперь процессор снова работает в реальном режиме
return: mov     AX,data         ;(188)
        mov     DS,AX           ;(189)
;Восстановим SS:SP
        mov     AX,stk          ;(190)
        mov     SS,AX           ;(191)
        mov     SP,256          ;(192)
        mov     SS,real_ss      ;(193)
;Разрешим аппаратные и немаскируемые прерывания
        sti                     ;(194)Разрешим аппаратные прерывания
        mov     AL,0            ;(195)Засылка константы с битом 7=0
        out     70h,AL          ;(196)в порт CMOS - разрешение NMI
;Проверим выполнение функций DOS и завершим программу
        mov     AH,09h          ;(197)
        mov     DX,offset mes;(198)
        int     21h             ;(199)
        mov     AX,4C00h        ;(200)Завершим программу обычным
                                ;образом
        int     21h             ;(201)
main    endp                    ;(202)
;Процедура заполнения расширенной памяти, образующая задачу 1
highmem proc                    ;(203)
;Заполним мегабайт расширенной памяти
        mov     AX,40           ;(204)Селектор сегмента в
                                ;расширенной памяти;
        mov     GS,AX           ;(205)в расширенной памяти
        mov     EAX,0           ;(206)Первое число-заполнитель
        mov     EBX,0           ;(207)Индекс в сегменте
        mov     ECX,80000h ;(208)80000h*4=200000h=2Мбайт
fill:   mov     GS:[EBX],EAX;(209)Заполняем память!
        push    EAX             ;(210)Сохраним EAX и CX на время
        push    CX              ;(211)диагностических действий
        mov     SI,offset number+5;(212)Сюда младшую половину EAX
        debug                   ;(213)
        shr     EAX,16          ;(214)Обменяем половины EAX
        mov     SI,offset number;(215)Сюда старшую половину EAX;
        debug                   ;(216)
;Выведем диагностическую строку про заполнение старшей памяти
        mov     SI,offset number;(217)DS:SI-адрес строки
```

```
        mov     CX,9            ;(218)CX=длина строки
        mov     AH,43h          ;(219)Атрибут
        mov     DI,1040         ;(220)ES:DI=позиция на экране
scrh:   lodsb                   ;(221)Заберем из строки байт
        stosw                   ;(222)И выведем в видеобуфер слово
        loop    scrh            ;(223)Повторить CX раз
        pop     CX              ;(224)Восстановим
        pop     EAX             ;(225)регистры
        add     EBX,4           ;(226)Смещение индекса в памяти
        inc     EAX             ;(227)Инкремент числа-заполнителя
        db      67h             ;(228)Команда loop должна работать с ECX
        loop    fill_1          ;(229)Вспомогательный переход
        jmp     go              ;(230)Выход из цикла после его окончания
fill_1: jmp     fill            ;(231)Длинный переход на начало цикла
go:     iret                    ;(232)Завершение задачи 1 и обратное
                                ;переключение на задачу 0
highmem endp                    ;(233)Конец процедуры
code_size=$-begin               ;(234)Размер сегмента команд
text    ends                    ;(235)
stk     segment stack 'stack';(236)
        db      256 dup ('^');(237)
stk     ends                    ;(238)
        end main                ;(239)
```

Таблица глобальных дескрипторов дополнена двумя новыми дескрипторами gdt_tss_0 и gdt_tss_1 (предложения 26-27). Это дескрипторы сегментов состояния задач TSS. В нашем примере используются TSS минимального размера - по 104 байта, поэтому в поле границы дескрипторов TSS указано число 103. Атрибут 1 дескрипторов имеет значение 89h: присутствующий, уровень привилегий DPL=0, свободный TSS МП 486 (рис. 67.5).

Рис. 67.5. Расшифровка значения атрибута для дескриптора TSS.

Вспомним, что дескриптор TSS может быть четырех типов: свободный TSS МП 286, занятый TSS МП 286, свободный TSS МП 386/486 и занятый TSS МП 386/486 (см. табл. 66.2). Описывая сегменты состояния задач в таблице дескрипторов, мы указываем с помощью кода типа, что они свободны. Как только селектор дескриптора TSS будет загружен в регистр задачи TR, процессор изменяет код атрибута дескриптора этого TSS, объявляя его занятым. При этом TSS исходной задачи остается занятым до ее завершения, даже если выполняются переключения на другие задачи. TSS вложенной задачи помечается процессором, как занятый, как только произойдет переключение на эту задачу, а после ее

завершения и возврата в исходную задачу TSS вложенной задачи снова освобождается. Попытка переключения на задачу, TSS который занят, приводит к исключению нарушения общей защиты. Тем самым предотвращается повторный запуск активной задачи.

В сегменте данных главной задачи зарезервировано место для двух сегментов состояния задач (предложения 48-49). Там же предусмотрено двухсловное поле адреса для команды call dword ptr переключения на вложенную задачу (предложения 50-51). Поскольку переключение осуществляется через TSS задачи, в качестве сегментного адреса задачи указывается селектор ее TSS (в данном случае 56). Слово со смещением при переключении задач игнорируется, хотя должно присутствовать в программе согласно формату команды call.

В предложениях 110...115 вычисляется и загружается в дескриптор TSS_0 линейный адрес сегмента состояния задачи 0; затем та же операция выполняется для TSS_1.

Перед переключением на задачу 1 следует инициализировать некоторые поля TSS_1 (TSS_0 будет заполнен процессором при первом возврате из вложенной задачи в исходную). В данном примере инициализация TSS_1 выполняется еще в реальном режиме, хотя эту операцию можно было бы перенести в защищенный режим. Поля для CS и IP заполняются селектором сегмента команд задачи 1 (в данном случае это общий для обеих задач сегмент с селектором 16) и смещением highmem точки входа в задачу 1. Для стека задачи 1 произвольно выделена область, начиная с середины нашего сегмента стека (селектор сегмента тот же, а смещение 128; предложения 127-128). Поскольку задача 1 будет использовать общий сегмент данных и обращаться к видеобуферу, ей передаются селекторы этих сегментов (предложения 129-130).

Далее следуют уже рассмотренные ранее операции запрета прерываний, подготовки адреса возврата в реальный режим, загрузки регистра IDTR, открытия линии A20 и перехода в защищенный режим.

После перехода в защищенный режим и выполнения обычных процедур инициализации сегментных регистров в регистр задачи TR загружается селектор TSS исходной задачи (предложения 166-167). Переключение на вложенную задачу можно будет выполнить и без этого, однако загрузка TR обеспечит возврат из вложенной задачи в исходную.

Наконец, командой косвенного дальнего вызова (предложение 168) осуществляется переключение на задачу 1, в качестве которой выступает процедура highmem, размещенная нами в самом конце сегмента команд (начиная с предложения 203). Текст процедуры целиком взят из примера 66.1; процедура заканчивается командой iret переключения на исходную задачу.

Задача 1 заполняет расширенную память последовательными числами, выводя их одновременно на экран, что позволяет наглядно

контролировать ее работу. Завершающая команда iret этой задачи осуществляет обратное переключение, передавая управление на очередную команду задачи 0, следующую за командой дальнего вызова. В предложениях 169-170 продемонстрирована техника повторного переключения на задачу 1: в поле TSS для указателя команд восстанавливается начальный адрес процедуры highmem и снова выполняется команда дальнего вызова. Продолжение исходной задачи не отличается от предыдущего примера.

Отладочная макрокоманда debug, используемая в программе, дает возможность детально изучить процесс переключения задач и участие в этом процессе различных системных структур. Можно порекомендовать читателю выполнить анализ следующих объектов.

1. Содержимое области связи в TSS_1 до переключения на задачу 1 и после этого переключения. До переключения эта область пуста, а после переключения в ней должен быть записан селектор TSS_0 (в нашем случае 48=30h)

2. Значение кода атрибута в дескрипторах сегментов состояния задач. До активизации задачи код должен быть равен 89h, после активизации - 8Bh, поскольку сегмент становится занятым.

3. Состояние регистра флагов и, в частности, флага NT, до переключения на вложенную задачу, "внутри нее" и после возврата в исходную задачу. Вывести на экран содержимое регистра флагов (нас будет интересовать только его младшее слово) можно следующим образом:

```
pushf                ;Отправим флаги в стек
pop   AX             ;Извлечем флаги из стека в AX
mov   SI,offset string+10;Преобразуем в символьную
debug                ;форму и выведем на экран
```

Этот эксперимент может оказаться не очень наглядным, поскольку, как уже отмечалось, в исходном состоянии компьютера флаг NT установлен. Естественно, он останется установленным и после переключения. Для того, чтобы детально изучить роль флага NT, следует сбросить его перед переключением на вложенную задачу. Это можно осуществить командами

```
mov   AX,0           ;Обнулим AX
push  AX             ;Отправим 0 в стек
popf                 ;Перенесем его в регистр флагов
```

(то, что сбрасываются и остальные флаги, роли не играет). Теперь анализ регистра флагов окажется более поучительным: флаг NT будет сброшен при выполнении задачи 0, но установлен при выполнении задачи 1. После возврата в задачу 0 флаг снова сбросится, так как именно такое состояние регистра флагов будет храниться в TSS_0.

Чтобы проанализировать процесс сохранения и восстановления содержимого регистра флагов, можно после очистки регистра флагов принудительно установить в нем какой-либо флаг, например CF (что можно выполнить командой stc) и вывести на экран, кроме слова флагов, еще и содержимое ячеек TSS_0 и TSS_1 со смещением 24h.

4. Процесс передачи параметров в вызываемую задачу через ее TSS. Для этого можно заполнить какими-либо числами поля TSS_1, предназначенные для хранения содержимого регистров общего назначения, и проанализировать содержимое этих регистров при входе в задачу 1.

Статья 68

Дескрипторы-псевдонимы

Как было показано в предыдущих статьях, процессор, работая в защищенном режиме, обеспечивает проверку правильности обращения к объявленным в программе сегментам. Так, описав сегмент данных дескриптором с типом 0, можно защитить его от записи; для обычного сегмента команд (тип 4) разрешается только исполнение, но не чтение или модификация. Между тем, во многих случаях требуется расширить права программы при обращении к сегментам. В тех случаях, когда требуемые характеристики не предусмотрены среди стандартных типов дескрипторов, расширение прав доступа к сегменту осуществляется путем описания для того же сегмента второго дескриптора с другими правами доступа. Если, например, нам требуется не только исполнять строки сегмента команд, но и читать их (с целью, например, пересылки программы в другую область памяти или на диск), достаточно присвоить дескриптору этого сегмента команд тип 5 (разрешены исполнение и чтение). Если, однако, мы хотим еще и модифицировать командные строки, то сегмент команд следует описать с помощью двух дескрипторов, одного с типом 4 или 5, а второго с типом 1 (разрешены чтение и запись). Такой дополнительный дескриптор, расширяющий права доступа к сегменту, называется дескриптором-псевдонимом.

Рассмотрим методику использования дескриптора-псевдонима, модифицировав для этого пример предыдущей статьи. В примере 67.1 процедура highmem работала с довольно большим (двухмегабайтным) участком расширенной памяти, заполняя его данными. Перешлем саму

эту процедуру в расширенную память и заставим ее выполняться там в качестве самостоятельной задачи. Место же в обычной памяти, первоначально занимаемое этой процедурой, можно будет при необходимости использовать под другие данные.

Предлагаемая программа почти полностью повторяет пример 67.1 и приведена ниже фрагментарно.

Пример 68.1. Пересылка задачи в расширенную память.

```
...
;Структура для описания дескрипторов сегментов
descr   struc
        ...
descr   ends
;Структура для описания шлюзов ловушек
trap    struc
        ...
trap    ends
data    segment use16
;Таблица глобальных дескрипторов GDT
gdt_0   label word          ;Начало GDT
;Дескрипторы gdt_null, gdt_data, gdt_code, gdt_stack, gdt_screen,
;gdt_himem, gdt_tss_0, gdt_tss_1
        ...
gdt_hm_load descr <0FFFFh,,30h,92h>;Селектор 64, расширенная
                            ;память для загрузки программы
gdt_hm_task descr <0FFFFh,,30h,98h>;Селектор 72, расширенная
                            ;память для выполнения программы
gdt_size=$-gdt_0            ;Размер GDT
;Таблица дескрипторов прерываний IDT
        ...
;Дескрипторы исключений
        ...
;Поля данных программы
        ...                 ;Повторяют поля данных примера 67.1
data    ends
text    segment 'code' use16
        ...
;Обработчики исключений недопустимого сегмента состояния задачи
;exc_0ah, исключения отсутствия сегмента exc_0bh, исключения
;ошибки обращения к стеку exc_0ch, исключения общей защиты
;exc_0dh, ;исключения страничного нарушения exc_0eh, остальных
;исключений dummy_exc
        ...
main    proc                ;Начало главной процедуры
        ...
;Вычислим линейные адреса сегментов данных, команд и стека и
;занесем их в таблицу глобальных дескрипторов
        ...
;Вычислим линейные адреса сегментов задачи TSS_0 и TSS_1 и занесем
;их в таблицу глобальных дескрипторов
        ...
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        ...
;TSS_0 главной задачи инициализировать не надо
```

```
;Инициализируем TSS_1 процедуры заполнения расширенной памяти
        mov    word ptr tss_1+4Ch,72;CS
        mov    word ptr tss_1+20h,0;IP
        mov    word ptr tss_1+50h,64;SS
        mov    word ptr tss_1+38h,0FFFEh;SP
        mov    word ptr tss_1+54h,8;DS
        mov    word ptr tss_1+48h,32;ES
;Подготовимся к переходу в защищенный режим
        ...
;Загрузим IDTR
        ...
;Откроем линию A20 (допустимо только после запрета прерываний)
        ...
;Переходим в защищенный режим
        ...
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
        ...
continue:
;Делаем адресуемыми данные и стек
        ...
;Загрузим TR
        mov    AX,48        ;Селектор TSS_0
        ltr    AX
;Перешлем процедуру highmem в четвертый мегабайт расширенной
;памяти
        push   DS           ;Сохраним DS
        push   CS           ;Перешлем CS в DS
        pop    DS           ;DS->сегмент команд
        mov    SI,offset highmem;DS:SI->пересылаемая программа
        mov    AX,64        ;Настроим ES на сегмент
        mov    ES,AX        ;в расширенной памяти
        mov    DI,0         ;ES:DI->область пересылки
        mov    CX,hm_size   ;Размер пересылаемой программы
rep     movsb               ;Пересылка
        pop    DS           ;Восстановим DS
;Настроим ES на сегмент видеобуфера
        mov    AX,32
        mov    ES,AX
;Выполним переключение на задачу в расширенной памяти
        call   dword ptr task1_offs
        mov    AX,0FFFFh
home:   mov    SI,offset string
        debug
;Выведем на экран диагностическую строку
        ...
;Закроем линию A20
        ...
;Переключим режим процессора
...
;Теперь процессор снова работает в реальном режиме
return:
;Восстановим операционную среду реального режима (DS, SS:SP)
        ...
;Разрешим аппаратные и немаскируемые прерывания
        ...
;Проверим выполнение функций DOS и завершим программу
```

```
        ...
main    endp
;-------------------------------------------------
;Процедура, предназначенная для пересылки в расширенную память
highmem proc
;Заполним область расширенной памяти
        ...
;Выведем на экран диагностическую строку про заполнение старшей
;памяти
        ...
        iret
highmem endp
hm_size=$-highmem
;-------------------------------------------------
code_size=$-begin
text    ends
stk     segment stack 'stack'
        db    256 dup ('^')
stk     end
        end main
```

Для того, чтобы выполнить пересылку (а, следовательно, и чтение) части сегмента команд, его следует описать с разрешением исполнения и чтения. Поэтому в дескрипторе gdt_code сегмента команд указано значение атрибута 9Ah (рис. 68.1).

Рис. 68.1. Расшифровка значения атрибута для дескриптора сегмента команд.

Помимо дескриптора сегмента данных расширенной памяти, который планируется заполнить числами, в таблицу глобальных дескрипторов включены еще два дескриптора, описывающих одну и ту же область расширенной памяти. Эта область располагается с самого начала четвертого мегабайта (линейный адрес 300000h) и имеет размер 64 Кбайт. Дескриптор gdt_hm_load предназначен для обращения к этой области с целью записи в нее пересылаемой программы и имеет атрибут 92h (присутствует, чтение/запись); дескриптор gdt_hm_task описывает ту же область, как сегмент данных и имеет обычный для сегментов команд атрибут 98h. Эти два дескриптора представляют собой псевдонимы. Кстати, необходимость в дескрипторе-псевдониме возникла из-за того, что среди стандартных типов дескрипторов отсутствует тип исполняемого сегмента с разрешением записи.

Несколько иначе, чем в предыдущем примере, выполняется настройка TSS вложенной задачи. В поле для селектора исполняемого

сегмента заносится селектор 72, описывающий участок расширенной памяти, как исполняемый сегмент. Поле для IP содержит 0, так как исполняемая процедура начинается с байта 0 этого участка.

В сегментный регистр SS загружается селектор 64, описывающий тот же участок расширенной памяти, но как сегмент данных с разрешением чтения/записи. Вполне произвольным образом место для стека назначено в конце этого сегмента (для чего он и сделан внушительных размеров - 64 Кбайт). Указатель стека инициализируется числом FFFEh.

Поля сегментных регистров DS и ES заполняются теми же селекторами, с которыми работает главная задача. Таким образом, вложенная задача в нашем случае имеет доступ к данным главной задачи, а также к видеобуферу. На рис. 68.2 изображены (не в масштабе) используемые в нашем примере области памяти.

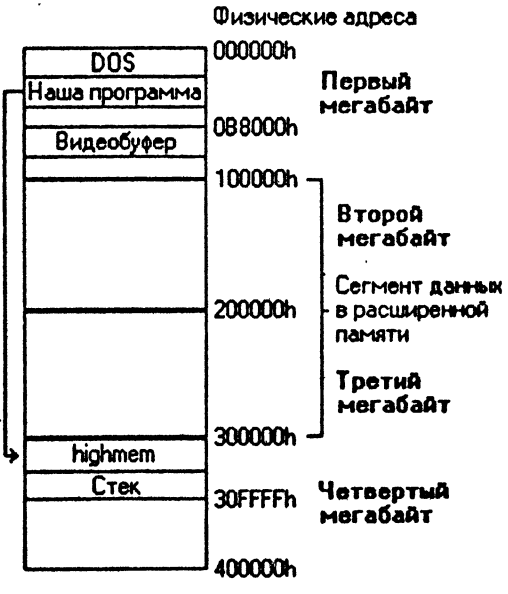


Рис. 68.2. Области памяти, используемые в примере 68.1.

После перехода в защищенный режим программа инициализирует сегментные регистры и объявляет себя задачей, загрузив в регистр задач TR селектор своего TSS (в нашем примере 48).

Теперь можно переслать в расширенную память процедуру highmem, для чего естественно воспользоваться командой movsb. В регистры DS:SI заносится двухсловный адрес процедуры highmem в обычной памяти (8:highmem), в регистры ES:DI - адрес ее будущего расположения в расширенной памяти (64:0). При этом, естественно, используется селектор дескриптора gdt_hm_load, который описывает начало четвертого мегабайта памяти, как сегмент данных. В регистр CX заносится размер процедуры highmem в байтах, и командой movsb с префиксом rep процедура пересылается в расширенную память.

Сегмент состояния задачи TSS_1 уже настроен должным образом, при этом в поле для CS (относительный адрес 4Ch) указан селектор де-

скриптора gdt_hm_task, который описывает ту же область данных (начало четвертого мегабайта), как сегмент команд с правом исполнения.

Уже знакомой нам командой дальнего вызова управление передается процедуре highmem в расширенной памяти, которая теперь выступает в роли вложенной задачи. После ее завершения команда iret осуществляет переключение на исходную задачу, которая завершается так же, как и в примере 67.1.

Таким образом, в нашем примере дескрипторы-псевдонимы позволили сначала переслать в некоторую область памяти коды программы (как данные), а затем обратиться к этой же области памяти, как к исполняемому сегменту.

Между прочим, в примере 67.1 в неявной форме использовался еще один дескриптор-псевдоним. Вспомним, что для вторичного переключения на задачу 1 нам понадобилось "подправить" содержимое поля для указателя команд в TSS_1 (предложение 169 примера 67.1). Однако процессор запрещает обращение к сегментам состояния задач с целью их чтения или модификации. Селекторы дескрипторов TSS можно использовать только для переключения задач; загрузить такой селектор в сегментный регистр не удастся. Для того, чтобы в защищенном режиме выполнить настройку или модификацию сегмента состояния задачи, для него необходимо создать дескриптор-псевдоним с атрибутом разрешения чтения-записи. Загрузив в какой-либо сегментный регистр селектор этого дескриптора, можно обращаться к этому сегменту, как к обычному сегменту данных.

В нашем примере такой дескриптор-псевдоним получился сам собой. Это произошло потому, что память под оба сегмента состояния задач (tss_0 и tss_1) была зарезервирована в пределах обычного сегмента данных, дескриптор которого (gdt_data), естественно, давал право чтения и записи. В пределах этого сегмента данных можно обращаться и к полям TSS. Если бы мы написали текст программы более скрупулезно, выделив каждый TSS в отдельный сегмент, для их модификации пришлось бы дополнительно ввести дескрипторы-псевдонимы.

Рассмотрим другой способ переключения задач - передачу управления вложенной задаче с помощью команды дальнего косвенного перехода jmp dword ptr. В этом случае вложенная задача должна заканчиваться не командой iret, а ответной командой jmp dword ptr, для которой в качестве селектора перехода задан селектор TSS исходной задачи.

Для реализации переключений-переходов в полях данных главной задачи, кроме двухсловного поля в адресом вложенной задачи, следует предусмотреть такое же поле с адресом исходной задачи:

```
task1_offs dw  0          ;Игнорируемое смещение
```

```
task1_sel dw   56          ;Селектор TSS_1
task0_offs dw  0           ;Игнорируемое смещение
task0_sel dw   48          ;Селектор TSS_0
```

Команда переключения на задачу 1 будет выглядеть следующим образом:

```
        jmp    dword ptr task1_offs
```

Процедура highmem должна завершаться такой же командой:

```
go_out: jmp    dword ptr task0_offs
```

Заметим, что поле task0_offs расположено в сегменте данных задачи 0, и для доступа к нему из задачи 1 в ней один из сегментных регистров данных должен содержать селектор этого сегмента. Однако мы заранее побеспокоились об этом, передав задаче 1 содержимое DS (и лишив ее тем самым возможности обращаться через этот регистр к собственным полям данных, которых, впрочем, у нее нет)

Переключение задач в приведенных примерах осуществлялось передачей управления на дескрипторы TSS вложенной или исходной задач, как если бы эти дескрипторы представляли собой программы. Разумеется, дескриптор выполнить нельзя, однако процессор по типу дескриптора (свободный дескриптор TSS) определяет, что от него требуется не переход на подпрограмму, а переключение на задачу, адрес которой содержится в TSS.

Существует другой способ переключения задач, в котором используется специальный дескриптор - шлюз задачи, содержащий, в частности, селектор требуемого TSS. Селектор шлюза задачи указывается в качестве "адреса перехода" для команды дальнего вызова, и процессор, определив по типу дескриптора (теперь уже не свободный дескриптор TSS, а шлюз задачи) характер требуемых от него действий, осуществляет переключение задач через тот же TSS.

Для демонстрации этого метода включим в таблицу глобальных дескрипторов шлюз задачи для задачи 1:

```
gdt_task1 trap <0,56,0,85h,0>
```

Если этот дескриптор стоит в GDT на последнем, десятом месте, ему будет соответствовать селектор 80. Заметьте, что шлюз задачи описан не как дескриптор памяти (структура descr), а как дескриптор ловушки (структура trap). Формат шлюза задачи приведен на рис. 68.3.

В отличие от других дескрипторов, в шлюзе задачи указывается только один сегментный адрес, в качестве которого выступает селектор TSS требуемой задачи, в нашем случае число 56. Код атрибута составляет 85h (присутствует, тип 5).

Для вызова вложенной задачи через шлюз задачи достаточно в поле с адресом задачи указать селектор этого шлюза:

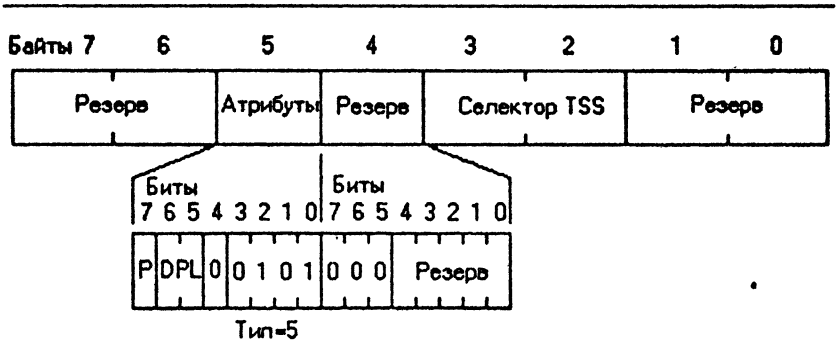


Рис. 68.3. Формат шлюза задачи.

```
task1_offs dw  0          ;Игнорируемое смещение
task1_sel dw   80         ;Селектор шлюза задачи 1
```

По-прежнему переключение можно осуществить как дальним вызовом, так и дальним переходом. Возврат в исходную задачу в осуществляется, как и раньше, командой iret (если задача была вызвана командой call) или комадной jmp dword ptr (если задача была вызвана такой же командой). В последнем случае возврат можно выполнить как через дескриптор TSS, так и через шлюз задачи 0.

Статья 69

Переключение задач по аппаратным прерываниям

В предыдущих статьях были рассмотрены вопросы синхронного переключения задач, когда переключение исходной задачи на другую (вложенную) задачу осуществлялось по командам дальнего вызова, находящимся в исходной задаче. Такой программный комплекс отличается большой жесткостью: порядок переключения задач определен раз и навсегда на этапе написания программы. На практике часто желательно иметь возможность асинхронного переключения задач по аппаратным прерываниям или по исключениям. Наиболее типично периодическое переключение задач от таймера (это основа мультизадачного режима с разделением времени), а также от клавиатуры, когда пользователь может по желанию вызывать к жизни ту или иную задачу. Рас-

смотрим элементы асинхронного переключения задач по командам, подаваемым с клавиатуры.

Для управления задачами по прерываниям или исключениям в архитектуре процессора предусмотрена возможность включения шлюзов задач непосредственно в таблицу дескрипторов прерываний. В этом случае прерывание немедленно приводит к переключению на вложенную задачу, которая выполняется до завершающей команды iret, после чего управление возвращается в главную задачу. Вложенная задача, помимо своих основных функций, должна взять на себя всю обработку прерывания, в частности, управление контроллером аппаратуры (если оно требуется), а также контроллером прерываний (что требуется всегда).

Преобразуем пример 68.1 так, чтобы переключение на процедуру highmem, располагаемую в расширенной памяти, происходило по нажатию любой клавиши. Вообще говоря, расширенная память не имеет отношения к рассматриваемому вопросу, но мы предпочли описать изменения пусть излишне сложного, но уже знакомого примера, чем приводить полный текст нового. Хотя полный текст предлагаемой программы не приводится, для удобства последующих ссылок будет считать, что мы рассматриваем пример 69.1.

Пример 69.1. Переключение задач по аппаратным прерываниям через шлюз задачи в таблице дескрипторов прерываний

Ниже описываются изменения, которые следует внести в текст примера 68.1.

. Сегмент таблицы глобальных дескрипторов не изменился. Единственное усовершенствование - изменена граница исполняемого сегмента в расширенной памяти. В качестве границы указан не последний байт 64-кбайтного участка, а последний байт процедуры:

```
gdt_hm_task descr <hm_size-1,0,30h,98h>;Селектор 72, расширенная
                                       ;память для выполнения программы
```

Указанное изменение не отражается на выполнении, однако демонстрирует более грамотное описание объектов программы.

Для обеспечения обработки аппаратных прерываний от клавиатуры в таблицу IDT вслед за дескрипторами исключений необходимо включить дескрипторы обработчиков таймера и клавиатуры. Поскольку в настоящей программе не предполагается обработка прерываний от таймера, его дескриптор может быть фиктивным. Конец таблицы дескрипторов прерываний будет выглядеть следующим образом:

```
trap    17+1 dup (<dummy_exc>);17 дескрипторов зарезервированных
                               ;исключений плюс 1 дескриптор
                               ;обработчика прерываний от таймера
trap    <0,56,0,85h,0>         ;Шлюз задачи 1
idt_size=$-idt
```

В качестве дескриптора прерываний от клавиатуры использован непосредственно шлюз вложенной задачи. Формат шлюза задачи описывался в предыдущей статье; поле селектора имеет значение 56, что соответствует дескриптору gdt_tss_1.

Следующее изменение касается строк подготовки к переходу в защищенный режим. Поскольку мы будет перепрограммировать контроллер прерываний, байт состояния отключения должен иметь значение 05h (возврат в программу с перепрограммированием контроллеров):

```
mov    AL,05h
out    71h,AL
```

Далее надо перепрограммировать ведущий контроллер прерываний и запретить прерывания ведомого так, как это делалось в примере 65.1 (предложения 115...132). При этом необходимо изменить маску прерываний в ведущем контроллере (предложение 128 примера 65.1) с FEh на FDh, чтобы размаскировать прерывания от клавиатуры (и замаскировать от таймера).

После перехода в защищенный режим, выполнения настроечных операций и пересылки в расширенную память процедуры higmem надо разрешить прерывания:

```
mov    AL,0Dh      ;Засылка константы с битом 7=0
out    70h,AL      ;в порт CMOS - разрешение NMI
sti                ;Разрешим аппаратные прерывания
```

Естественно, в данном варианте задача 1 не должна вызываться из главной задачи, поэтому команды дальнего вызова отсутствуют. Однако для того, чтобы дать пользователю время для ввода команды с клавиатуры, главная задача программа должна что-делать. Организуем периодический вывод символов на экран, как это было сделано в примере 65.1:

```
       mov    CX,500      ;Число символов
       mov    BX,2720     ;Начальная позиция на экране
       mov    DX,0E01h    ;начальный символ и атрибут
xxxx:  push   CX          ;Сохраним счетчик внешнего цикла
       mov    CX,0        ;Цикл задержки для замедления
zzzz:  loop   zzzz        ;вывода на экран
       mov    ES:[BX],DX  ;Вывод в видеобуфер
       inc    DX          ;Инкремент символа
       add    BX,2        ;Инкремент позиции на экране
       pop    CX          ;Извлечем счетчик внешнего цикла
       loop   xxxx        ;Цикл
```

Далее идут строки вывода диагностического сообщения, блокирования линии A20, перехода в реальный режим и завершения программы:

```
       mov    AX,0FFFFh
home:  mov    SI,offset string
       debug
```

Процедура highmem теперь будет вызываться через шлюз задачи непосредственно аппаратным прерыванием от клавиатуры, поэтому в нее необходимо включить строки обслуживания клавиатуры и контроллера прерываний:

```
highmem proc
        in      AL,60h      ;Ведем скен-код нажатой клавиши
        cmp     AL,80h      ;Скен код отпускания?
        jae     go          ;Да, на выход
;Заполним мегабайт расширенной памяти
        ...                 ;См. пример 67.1, предл. 204...230
fill_1: jmp     fill
go:     in      AL,61h      ;Получим содержимое порта B
        or      AL,80h      ;Установкой старшего бита
        out     61h,AL      ;и последующим сбросом его
        and     AL,7Fh      ;сообщим контроллеру клавиатуры о
        out     61h,AL      ;приеме скен-кода символа
        mov     AL,20h      ;Сигнал EOI
        out     20h,AL      ;в ведущий контроллер прерываний
        iret                ;Переключение на исходную задачу
        jmp     highmem     ;Точка входа при последующих
                            ;переключениях - переход на начало
highmem endp
hm_size=$-highmem               ;Размер процедуры
```

Прежде всего из порта клавиатуры 60h вводится скен-код нажатой клавиши и разрешается нажатие следующей. Как известно, при нажатии клавиши в порт 60h поступает код нажатия, а при отпускании генерируется новое прерывание и в порт 60h посылается код отпускания, который отличается от кода нажатия наличием 1 в старшем бите. Если не отсеять скен-коды отпускания, процедура highmem будет вызываться и при нажатии, и при отпускании клавиш, т.е. дважды на каждое нажатие. Поэтому в процедуре выполняется проверка скен-кода, и если он превышает 80h, т.е. является кодом отпускания, осуществляется переход на завершение процедуры.

Следующие изменения касаются завершения программы. Перед выходом из задачи (метка go) необходимо послать в контроллер прерывания сигнал конца прерывания EOI, иначе уровень прерывания от клавиатуры (и все нижележащие) останутся заблокированными до перепрограммирования контроллера.

Наконец, следует как-то решить проблему повторного переключения на задачу. Затруднение заключается в том, что при переключении на исходную задачу по команде iret в сегмент состояния задачи TSS_1 будет занесен контекст задачи на момент ее завершения, в котором содержимое указателя команд IP соответствует адресу следующей за iret команды, т.е. находится уже за пределами сегмента команд (размер которого мы сделали точно равным размеру процедуры). Повторный вызов процедуры, естественно, приведет к исключению общей защиты.

Возможный способ решения отмеченного затруднения заключается во включении в текст процедуры после (!) команды iret команды безусловного перехода на начало процедуры. После первого прогона вложенной задачи в поле для IP в ее сегменте состояния будет занесен адрес этой команды (следующей за iret). При втором вызове работа процедуры higmem начнется с этого адреса, что приведет к переходу на начало процедуры и дальнейшему нормальному выполнению. Этот процесс будет повторяться и при дальнейших вызовах.

Недостатком нашей программы является необходимость включения в вызываемую задачу строк управления аппаратурой, что лишает ее универсальности. Лучше, конечно, передать функции управления клавиатурой и контроллером прерываний специальному обработчику прерываний, оставив за задачей 1 только ее собственные содержательные функции. В этом случае задача 1 может иметь любое содержание и ее текст не будет жестко привязан к процедуре ее вызова.

Пример 69.2. Переключение задач по аппаратным прерываниям с помощью команды дальнего вызова

Ниже описываются изменения, которые следует внести в текст примера 69.1.

Аппаратное прерывание должно теперь активизировать обработчик прерываний, который, в свою очередь, будет вызывать переключение задачи. Поэтому в таблицу дескрипторов прерываний на соответствующее вектору клавиатуры место мы поместим не шлюз задачи, а обычный шлюз прерывания:

```
<new_09h,,,8Eh>                    ;Шлюз прерывания
```

В текст главной программы наряду с прочими обработчиками следует включить и обработчик прерывания от клавиатуры new_09h. Он должен выполнять весь необходимый анализ кодов от клавиатуры и в случае положительных результатов этого анализа переключаться на задачу 1. Полный текст обработчика приведен ниже.

```
new_09h proc
        push    AX          ;Сохраним
        push    BX          ;используемые регистры
        in      AL,60h      ;Введем скен-код
        mov     BL,AL       ;Перенесем его в BL
        in      AL,61h      ;Получим содержимое порта B
        or      AL,80h      ;Установкой старшего бита
        out     61h,AL      ;и последующим сбросом его
        and     AL,7Fh      ;сообщим контроллеру клавиатуры о
        out     61h,AL      ;приеме скен-кода символа
        cmp     BL,80h      ;Код отпускания?
        jae     out_09h     ;Да, на выход
        cmp     BL,2        ;Скен-код клавиши <1>?
        jne     out_09h     ;Нет, на выход
```

```
        call    dword ptr task2_offs;Переключение на вложенную
                                    ;задачу через дескриптор TSS
out_09h:mov     AL,20h              ;Сигнал
        out     20h,AL              ;EOI
        pop     BX                  ;Восстановим
        pop     AX                  ;используемые регистры
        db      66h                 ;Префикс 32-битовых операндов
        iret                        ;Выход из обработчика
new_09h endp
```

После сохранения единственного используемого в обработчике регистра выполняется ввод скен-кода клавиши, "отбраковка" кодов отпускания и анализ кода нажатия. Если нажата любая незапланированная клавиша (скен-код не равен 2), осуществляется переход на метку out_09h, генерация сигнала EOI и выход из обработчика в задачу. Если нажата клавиша <1>, выполняется переключение на задачу 1 командой дальнего косвенного вызова. В программе должна быть предусмотрена двухсловная ячейка с адресом перехода:

```
task1_offs dw  0            ;Смещение игнорируется
task1_sel dw   56           ;Селектор дескриптора gdt_tss_1
```

Из процедуры highmem убираются все строки управления контроллерами. Она завершается командой iret, за которой по-прежнему следует команда jmp на начало процедуры. После отработки задачи 1 управление вернется в обработчик new_09h, который закончит обслуживание аппаратного прерывания и командой iret передаст управление в главную задачу.

Статья 70

Мультизадачный режим с управлением от клавиатуры

Программы, рассмотренные в предыдущей статье, носят "полусинхронный" характер - вторую задачу можно запустить в произвольный момент нажатием клавиши, однако дальше активизированная задача становится неуправляемой, выполняясь до завершающей команды iret. В мультизадачной системе должна быть предусмотрена возможность переключения с задачи на задачу по каким-либо событиям, например, истечению заданного кванта времени или командам с клавиатуры. Перед тем, как описывать пример такой системы, нам придется более детально

рассмотреть процесс переключения задач в условиях действия аппаратных прерываний. На рис. 70.1 схематически изображено взаимодействие программно-аппаратных элементов при выполнении программы 69.2.

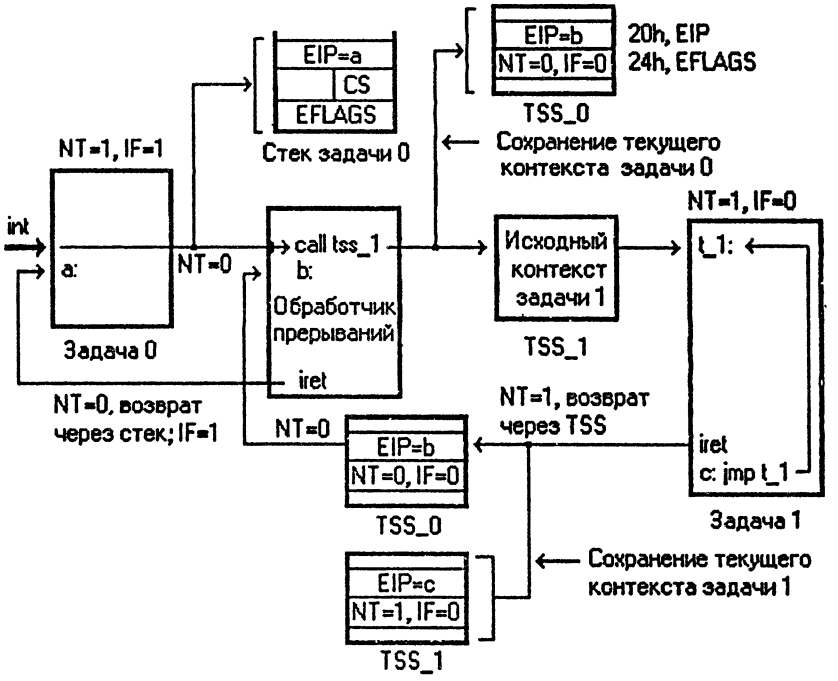


Рис. 70.1. Взаимодействие вычислительных объектов в процессе переключения задачи.

Исходная задача 0 выполняется в условиях установленных флагов NT и IF. Начальное состояние флага NT особого значения не имеет, а прерывания, естественно, должны быть разрешены. Приход сигнала аппаратного прерывания (в нашем случае от клавиатуры) приводит к сбросу флагов NT и IF, занесению вектора прерванного процесса в стек задачи 0 и передаче управления через шлюз прерывания на обработчик прерываний от клавиатуры. В нашем примере он находится в том же сегменте, что и задача 0. В качестве адреса возврата в стеке сохраняется адрес очередной команды (на рисунке он обозначен меткой a:).

В обработчике после анализа кода нажатой клавиши выполняется команда дальнего косвенного вызова

```
call  dword ptr task1_offs
```

для которой в качестве сегментного адреса перехода указан селектор сегмента состояния задачи 1 TSS_1. На рисунке эта команда условно обозначена call tss_1, чтобы подчеркнуть, что вызов осуществляется через сегмент состояния задачи. Команда call инициирует действия по переключению задач: сегмент состояния первой задачи TSS_0 заполняется текущим контекстом, а содержимое TSS_1 используется в качестве исходного контекста для запуска задачи 1.

В этой процедуре есть два тонких момента. Во-первых, в TSS_0 загружается фактически не контекст задачи 0, а контекст обработчика прерываний к моменту выполнения команды call. В частности, в качестве точки возврата указывается адрес возврата в обработчик (метка b:), а в слове флагов сброшены флаги NT и IF (оба флага сбрасываются процессором при получении сигнала аппаратного прерывания). Если обработчик изменил содержимое каких-то регистров, то в TSS попадут именно эти измененные значения.

Другая особенность первого переключения на задачу 1 заключается в том, что контекст задачи целиком берется из сегмента состояния задачи TSS_1, который должен быть предварительно программно инициализирован. В нашем примере поле флагов (по относительному адресу 24h) содержит 0. Следовательно, при выполнении задачи 1 будут запрещены прерывания. Что же касается флага NT, то, хотя в слове флагов TSS он был сброшен, задача 1 будет выполняться при NT=1, так как этот флаг устанавливается процессором при переключении задачи.

Задача 1 выполняется до завершающей команды iret, которая при NT=1 инициирует обратное переключение задач, получив из области связи текущего TSS_1 адрес "предыдущего" TSS_0. TSS_1 заполняется текущим контекстом задачи 1, при этом в поле для EIP записывается адрес команды, следующей за командой iret (метка c: на рисунке). Из TSS_0 берется контекст возврата (фактически - контекст обработчика прерываний) и управление передается на метку b: обработчика. Программа обработчика продолжается и, поскольку флаг NT сброшен, завершающая команда iret передает управление не через сегмент состояния задачи, а через стек, обеспечивая правильный возврат в исходную задачу на метку a:.

Последующие переключения на задачу 1 (по командам с клавиатуры) будут передавать управление на метку c:. Поскольку по этому адресу у нас находится команда безусловного перехода на начало программы (метка t_1 на рисунке), задача будет выполняться правильно.

Легко видеть, что в рассматриваемом примере разрушение контекста задачи 0 обработчиком и сохранение в TSS_0 "неправильного" контекста не приводит к нарушению работы всего комплекса, если только в начале программы обработчика сохраняются (например, в стеке) все используемые в нем регистры, а перед завершающей командой iret вы-

полняется их восстановление. С другой стороны, задача 1 выполняется при запрещенных прерываниях, и воздействовать на ход ее выполнения нельзя - она всегда будет выполняться до команды iret.

Для того, чтобы получить возможность переключаться не только на задачу 1, но и из нее, надо выполнять эту задачу при разрешенных прерываниях. Прерывания можно разрешить как в самой задаче командой sti, так и в TSS задачи, записав в него исходное слово флагов с установленным флагом IF. Теперь нажатие клавиши в процессе выполнения задачи 1 снова активизирует обработчик прерывания, в котором, проанализировав код нажатой клавиши, можно выполнить команду переключения на другую задачу. Однако здесь нас подстерегают неприятности. Вспомним (см. табл. 66.2), что дескриптор сегмента состояния задачи может описывать TSS двух типов - свободные (код 9) и занятые (код Bh). Создавая в программе для каждой задачи сегменты состояния, мы объявляем их свободными. Команда call, осуществляя переключение на задачу, изменяет атрибут в дескрипторе соответствующего TSS, объявляя его занятым. TSS останется занятым до тех пор, пока в этой задаче не будет выполнена команда iret возврата назад по цепочке связанных задач. При этом команда call может активизировать только задачу со свободным TSS; при попытке вызывать командой call задачу с занятым TSS возбуждается исключение общей защиты. Из вложенной задачи с помощью команды call можно активизировать следующую задачу, но нельзя вернуться назад - для этого нужна команда iret.

Для того, чтобы обойти указанное ограничение, можно воспользоваться командой дальнего перехода через сегмент состояния задачи. Команда jmp, выполняя переключение задачи, не включает ее в связный список вложенных задач и, соответственно, не изменяет тип дескриптора TSS. Поэтому с помощью команды jmp можно осуществлять переключение задач как "вперед" - на новую задачу, так и "назад" - на ту задачу, из которой была активизирована данная.

Рассмотрим пример мультизадачной системы, в которой переключение задач осуществляется по командам с клавиатуры с помощью команд дальнего косвенного перехода jmp. Предусмотрим в программном комплексе главную задачу 0, в которой будет осуществляться инициализация всей системы, а также две демонстрационные задачи 1 и 2. Поначалу ради простоты расположим все задачи в одном сегменте команд и будем считать, что всем задачам доступен общий сегмент данных.

Пример 70.1. Мультизадачный режим с переключением от клавиатуры

```
include debug.mac
include mac.mac
.386P
;Структура для описания дескрипторов сегментов
descr   struc
```

```
        ...
descr   ends
;Структура для описания шлюзов ловушек и прерываний
trap    struc
        ...
trap    ends
data    segment use16
;Таблица глобальных дескрипторов GDT
;Поля дескрипторов gdt_null, gdt_data gdt_code, gdt_stack и
;gdt_screen, а также дескрипторов сегментов данных, команд, стека
;и видеобуфера
        ...
gdt_tss_0 descr <103,0,0,89h>;Селектор 40 - дескриптор TSS_0
gdt_tss_1 descr <103,0,0,89h>;Селектор 48 - дескриптор TSS_1
gdt_tss_2 descr <103,0,0,89h>;Селектор 56 - дескриптор TSS_2
gdt_size=$-gdt_0          ;Размер GDT
;Таблица дескрипторов прерываний IDT
;Дескрипторы исключений dummy_exc (10 штук), exc_0ah, exc_0bh,
;exc_0ch, exc_0dh, ;exc_0eh и снова dummy_exc (18 штук)
        ...
pdescr  dq      0           ;Псевдодескриптор
mes     db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$'
tblhex  db      '0123456789ABCDEF'
string  db '**** ****_**** ****_**** **** ****'
;           0    5    10   15   20   25   30
len=$-string
number  db      '???? ????'
home_sel dw     home        ;Адрес возврата из исключения
        dw      10h         ;Сегмент команд
t1_addr dw      0,40        ;Для переключения через TSS_0
t2_addr dw      0,48        ;Для переключения через TSS_1
t3_addr dw      0,56        ;Для переключения через TSS_2
tss_0   dw      72 dup (0) ;Сегмент состояния задачи TSS_0
tss_1   dw      72 dup (0) ;Сегмент состояния задачи TSS_1
tss_2   dw      72 dup (0) ;Сегмент состояния задачи TSS_2
mes1    db      22,22,22,'Работает задача Task_1',22,22,22
mes1_len=$-mes1
mes2    db      22,22,22,'Работает задача Task_2',22,22,22
mes2_len=$-mes2
data_size=$-gdt_0         ;Размер сегмента данных
data    ends
text    segment 'code' use16
        assume CS:text,DS:data
begin   label word        ;Начало сегмента команд
;Обработчики исключений недопустимого сегмента состояния задачи
;exc_0ah, исключения отсутствия сегмента exc_0bh, исключения
;ошибки обращения к стеку exc_0ch, исключения общей защиты
;exc_0dh, исключения страничного нарушения exc_0eh, остальных
;исключений dummy_exc (см. пример 67.1)
        ...
;Обработчик аппаратных прерываний от клавиатуры
new_09h proc
        push    AX          ;Сохраним используемые
        push    DX          ;регистры
        in      AL,60h      ;Введем скен-код
        mov     DL,AL       ;Сохраним скен-код в DL
        in      AL,61h      ;Получим содержимое порта B
```

```
        or      AL,80h      ;Установкой старшего бита
        out     61h,AL      ;и последующим сбросом его
        and     AL,7Fh      ;сообщим контроллеру клавиатуры о
        out     61h,AL      ;приеме скен-кода символа
        mov     AL,20h      ;Сигнал
        out     20h,AL      ;EOI
        cmp     DL,0Bh      ;Скен-код клавиши <0>?
        je      tsk0        ;Да, на запуск задачи 0
        cmp     DL,2        ;Скен-код клавиши <1>?
        je      tsk1        ;Да, на запуск задачи 1
        cmp     DL,3        ;Скен-код клавиши <2>?
        je      tsk2        ;Да, на запуск задачи 2
        jmp     out_09h     ;Введено незапланированное
tsk0:   jmp     dword ptr t0_addr;Переключение на вложенную
                            ;задачу через дескриптор TSS_0
        jmp     out_09h     ;На выход из обработчика
tsk1:   jmp     dword ptr t1_addr;Переключение на вложенную
                            ;задачу через дескриптор TSS_1
        jmp     out_09h     ;На выход из обработчика
tsk2:   jmp     dword ptr t2_addr;Переключение на вложенную
                            ;задачу через дескриптор TSS_2
out_09h:pop     DX          ;Восстановим
        pop     AX          ;регистры
        db      66h         ;Префикс 32-битовых операндов
        iret                ;Выход из обработчика
new_09h endp
main    proc
;Вычислим линейные адреса сегментов данных, команд и стека и
;занесем их в таблицу глобальных дескрипторов
        ...
;Вычислим 32-битовые линейные адресе сегментов задачи TSS_0, TSS_1
;и TSS_2 и занесем их в таблицу глобальных дескрипторов
        ...
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
...
;Инициализируем TSS_1
        mov     tss_1+4Ch,16    ;CS
        mov     tss_1+20h,offset task2;IP
        mov     tss_1+50h,24    ;SS
        mov     tss_1+38h,256   ;SP
        mov     tss_1+54h,8     ;DS
        mov     tss_1+48h,32    ;ES
        sti                     ;Установим флаг IF
        pushfd                  ;EFLAGS в стек
        pop     EAX             ;EFLAGS в EAX
        mov     dword ptr tss_1+24h,EAX;EFLAGS+IF
;Инициализируем TSS_2
        mov     tss_2+4Ch,16    ;CS
        mov     tss_2+20h,offset task3;IP
        mov     tss_2+50h,24    ;SS
        mov     tss_2+38h,512   ;SP
        mov     tss_2+54h,8     ;DS
        mov     tss_2+48h,32    ;ES
        mov     dword ptr tss_2+24h,E AX;EFLAGS+IF
;Подготовимся к переходу в защищенный режим
        ...
;Перепрограммируем ведущий контроллер прерываний
```

```
        ...
;Запретим все прерывания в ведомом контроллере
        ...
;Загрузим IDTR
        ...
;Переходим в защищенный режим
        ...
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
        ...
continue:
;Делаем адресуемыми данные и стек, инициализируем регистр ES
;селектором видеобуфера
        ...
;Загрузим TR
        mov    AX,40       ;Селектор TSS_0
        ltr    AX
;Размаскируем NMI
        ...
        sti                ;Разрешение аппаратных прерываний
;Организуем периодический вывод на экран символов
        ...
        mov    AX,0FFFFh   ;Диагностическое значение
home:   mov    SI,offset string
        debug
;Выведем на экран диагностическую строку
        ...
;Переключим режим процессора
        ...
;Теперь процессор снова работает в реальном режиме
return:
;Восстановим операционную среду реального режима (DS, SS:SP),
;разрешим аппаратные и немаскируемые прерывания, проверим
;выполнение функций DOS и завершим программу. Закроем главную
;процедуру main
        ...
;Задача 1
task1   proc
a1:     mov    AH,71h      ;Атрибут
        mov    CX,2        ;Счетчик внешнего цикла
a2:     push   CX          ;Сохраним его в стеке
        mov    DI,800      ;Позиция на экране
        mov    CX,mes1_len     ;Длина строки
        mov    SI,offset mes1 ;Адрес строки
        cld                ;Пересылка вперед
a3:     lodsb              ;Загрузим в AL очередной символ строки
        stosw              ;Выведем символ с атрибутом
        loop   a3          ;Цикл по длине строки
        delay  30          ;Задержка
        pop    CX          ;Восстановим счетчик внешнего цикла
        mov    AH,17h      ;Инверсный атрибут
        loop   a2          ;Повторим вывод строки
        jmp    a1          ;Зациклим задачу
task1   endp
;Задача 2
task2   proc
b1:     mov    AH,34h      ;Другой атрибут
```

```
        mov     CX,2
b2:     push    CX
        mov     DI,1120     ;Другая позиция на экране
        mov     CX,mes2_len
        mov     SI,offset mes2
        cld
b3:     lodsb
        stosw
        loop    b3
        delay   30
        pop     CX
        mov     AH,43h
        loop    b2
        jmp     b1
task2   endp
code_size=$-begin
text    ends
stk     segment stack 'stack'
db      256*3 dup ('^')  ;Область для трех стеков
stk     ends
        end main
```

В таблице глобальных дескрипторов предусмотрены три дескриптора gdt_tss_0, gdt_tss_1 и gdt_tss_2 для сегментов состояния задач. Размер TSS - 104 байта (граница равна 103), тип дескриптора - свободные TSS МП 486.

В таблице IDT на месте, соответствующем вектору клавиатуры, расположен дескриптор - шлюз прерывания для вызова обработчика new_09h.

Три двухсловных ячейки t0_addr, t1_addr и t2_addr предназначены для команд дальних косвенных переходов, реализующих переключение задач. Первые слова этих ячеек игнорируются, во вторых записаны селекторы сегментов состояния задач TSS_0, TSS_1 и TSS_2.

Сообщения с именами mes1 и mes2 предназначены для вывода задачами 1 и 2. Сообщения украшены с обеих сторон символами полосочек (код ASCII 22).

Обработчик прерываний от клавиатуры анализирует коды нажимаемых клавиш и осуществляет переключение на задачи 0, 1 или 2 при нажатии клавиш <0>, <1> и <2>, соответственно. Поскольку при запуске программы начинает работать задача 0, первое переключение должно быть на задачу 1 или 2. После этого задачи можно переключать в любом порядке, хотя недопустимо переключение на активную задачу.

Весьма странная на первый взгляд последовательность из 6 команд jmp, идущих подряд, будет пояснена позже.

На этапе инициализации следует обратить внимание на инициализацию сегментов состояния переключаемых задач. TSS задачи 0, как и раньше, мы не инициализируем: он будет заполнен процессором при первом переключении на задачи 1 или 2. TSS задач 1 и 2 инициализи-

руются почти одинаково. В TSS заполняются поля CS (селектором общего сегмента команд), IP (относительными адресами задач), SS (селектором общего сегмента стека), DS (селектором общего сегмента данных), ES (селектором видеобуфера). В поля для SP заносятся разные смещения в пределах одного сегмента стека, размер которого увеличен в три раза. Таким образом, фактически задачи будут работать на разных стеках, что необходимо, поскольку переключения по аппаратным прерываниям могут происходить в произвольные моменты времени. Для того, чтобы вход в задачи 1 и 2 происходил при разрешенных прерываниях, в поля для EFLAGS заносится текущее содержимое EFLAGS после того, как командой sti явно разрешены прерывания.

Задачи 1 и 2 практически одинаковы. В каждой из них выводится на экран мигающая строка текста, для чего организован цикл из двух шагов. В одном шаге цикла строка выводится с одним атрибутом, в другом шаге - с другим. Завершающая команда jmp обеспечивает бесконечное выполнение каждой задачи. Таким образом, в программе не предусматривается "естественное" завершение задач, что потребовало бы их заметного усложнения.

Рассмотрим теперь ход выполнения нашего программного комплекса, а также его возможности и недостатки (рис. 70.2).

Исходная задача 0, как и в предыдущем примере, выполняется в условиях установленных флагов NT и IF. Приход сигнала аппаратного прерывания приводит к сбросу флагов NT и IF, занесению вектора прерванного процесса в стек задачи 0 и передаче управления через шлюз прерывания на обработчик прерываний от клавиатуры. В качестве адреса возврата в стеке сохраняется адрес очередной команды задачи 0 (на рисунке он обозначен меткой a:).

В обработчике после анализа кода нажатой клавиши с помощью команды дальнего косвенного перехода

```
jmp    dword ptr task1_offs
```

выполняется переключение на задачу 1. Сегмент состояния главной задачи TSS_0 заполняется текущим контекстом, а содержимое TSS_1 используется в качестве исходного контекста для запуска задачи 1. Поскольку в момент переключения выполняется программа обработчика, в TSS_0 в поле для EIP записывается адрес команды обработчика, следующей за командой jmp и помеченной на рисунке меткой b. Задача 1 представляет собой бесконечный цикл, поэтому она будет выполняться до тех пор, пока пользователь не подаст с клавиатуры команду переключения задач.

Приход прерывания от клавиатуры, как и раньше, приводит к сбросу флагов NT и IF, занесению вектора прерванного процесса в стек задачи 1 и передаче управления на обработчик прерываний от клавиатуры.

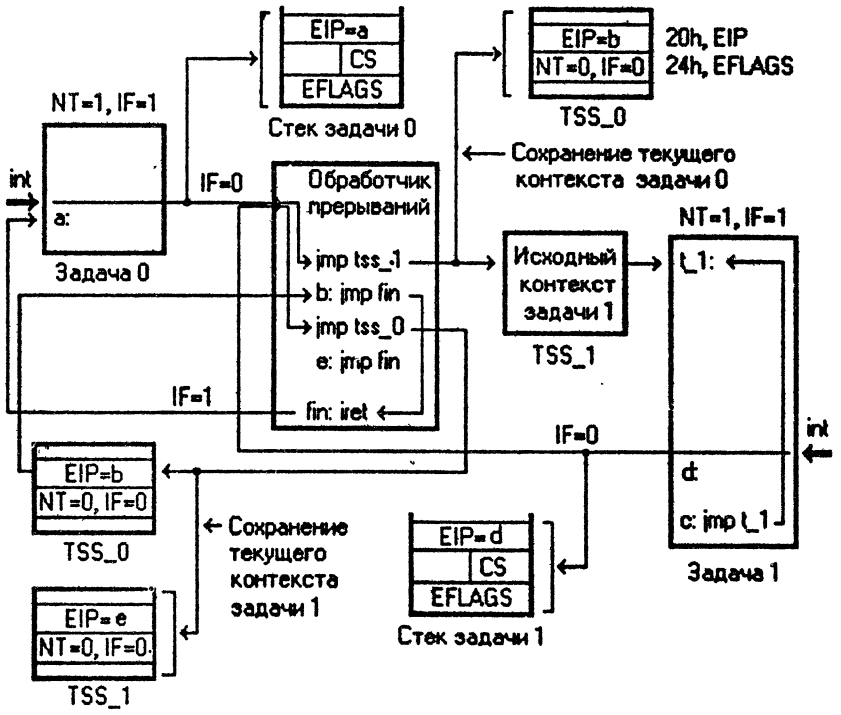


Рис. 70.2. Взаимодействие вычислительных объектов в мультизадачном комплексе.

Пусть пользователь подал команду возврата в задачу 0 (нажав на клавишу <0>). Обработчик, проанализировав код нажатой клавиши, передает управление на команду переключения на задачу 0 (на рисунке она обозначена, как jmp tss_0). Эта команда инициирует в процессоре процедуру переключения, в процессе которой TSS задачи 1 заполняется текущим контекстом, а содержимое TSS_0 используется в качестве исходного контекста для запуска задачи 0. Но в TSS_0 в качестве адреса возврата указано значение метки b в обработчике. Таким образом, происходит переключение не на задачу 0, а на обработчик. После каждой команды переключения jmp в обработчике стоит команда перехода на завершающую команду iret. Поскольку флаг IF сброшен, команда iret выполняет обычный возврат через стек текущей задачи, т.е. задачи 0. В результате происходит возврат в задачу 0 в ту точку, где она была прервана при переключении на задачу 1.

Все три задачи нашего программного комплекса вполне равноправны, поэтому, подавая соответствующие команды с клавиатуры, можно переключать их в произвольном порядке. Правда, в программе не пред-

усмотрена защита от повторного переключения на активную задачу. Если попытаться выполнить такое переключение, будет возбуждено исключение общей защиты. Не предусмотрено также завершение задач 1 или 2 - они прекращают работу при завершении главной задачи.

Отмеченные выше недостатки являются следствием примитивности нашей системы, которая предназначена не для изучения принципов построения мультизадачных систем, а лишь для демонстрации техники переключения задач и знакомства с соответствующими средствами микропроцессора. Однако и с этой точки зрения приведенный пример далек от совершенства. Одним из слабых его мест является использование всеми задачами комплекса общего сегмента данных. Так, обработчик прерываний выполняет переход на задачи через ячейки с селекторами TSS, расположенные в сегменте данных главной задачи; задачи 1 и 2 выводят сообщения, также хранящиеся в общем сегменте данных. Казалось бы, задачи 1 и 2 вполне могут использовать собственные сегменты данных, так как при переключении задач происходит сохранение старого контекста (включая сегментные регистры) и загрузка нового. Однако вызов обработчика аппаратных прерываний происходит без переключения контекста (заменяется только содержимое CS:IP); обработчик фактически работает на контексте прерванной задачи. Поэтому манипуляции с сегментными регистрами в задаче могут оказаться фатальными для ее работоспособности. Впрочем, здесь нет ничего нового: и в реальном режиме при входе в обработчик прерываний сегментные регистры адресуют сегменты прерванного процесса.

Рассмотрим пример, в котором несколько снижена взаимозависимость операционных сред задач и обработчика прерываний.

Пример 70.2. Вариант мультизадачного режима с переключением с клавиатуры

Ниже описаны лишь изменения, внесенные в программу 70.1.

Из общего сегмента данных удалены ячейки t0_addr, t1_addr и t2_addr с адресами TSS задач, а также сообщения задач mes1 и mes2. Адреса TSS задач перенесены в процедуру обработчика прерываний, а сообщения задач - в процедуры задач. Несколько изменилась программа обработчика прерываний:

```
new_09h proc
        push    AX          ;Сохраним используемые
        push    DX          ;регистры
        in      AL,60h      ;Введем скен-код
        mov     DL,AL       ;Сохраним скен-код в DL
        in      AL,61h      ;Получим содержимое порта B
        or      AL,80h      ;Установкой старшего бита
        out     61h,AL      ;и последующим сбросом его
        and     AL,7Fh      ;сообщим контроллеру клавиатуры о
        out     61h,AL      ;приеме скен-кода символа
        mov     AL,20h      ;Сигнал
```

```
        out     20h,AL      ;EOI
        cmp     DL,0Bh      ;Скен-код клавиши <0>?
        je      tsk0        ;Да, на запуск задачи 1
        cmp     DL,2        ;Скен-код клавиши <1>?
        je      tsk1        ;Да, на запуск задачи 2
        cmp     DL,3        ;Скен-код клавиши <2>?
        je      tsk2        ;Да, на запуск задачи 2
        jmp     out1        ;Введено незапланированное
tsk0:   pop     DX          ;Восстановим регистры перед
        pop     AX          ;переключением на задачу 0
        jmp     dword ptr CS:t0;Переключение на задачу 0 через
                            ;дескриптор TSS_0
        jmp     out_09h     ;На выход из обработчика
tsk1:   pop     DX          ;Восстановим регистры перед
        pop     AX          ;переключением на задачу 1
        jmp     dword ptr CS:t1;Переключение на вложенную задачу
                            ;через дескриптор TSS_1
        jmp     out_09h     ;На выход из обработчика
tsk2:   pop     DX          ;Восстановим регистры перед
        pop     AX          ;переключением на задачу 2
        jmp     dword ptr CS:t2;Переключение на вложенную задачу
                            ;через дескриптор TSS_2
out_09h:db      66h         ;Префикс 32-битовых операндов
        iret                ;Выход из обработчика
out1:   pop     DX          ;При вводе незапланированного
        pop     AX          ;восстановим регистры и
        jmp     out_09h     ;на выход из обработчика
t0_addr dw      0,40        ;Ячейки для косвенных
t1_addr dw      0,48        ;переходов с селекторами
t2_addr dw      0,56        ;TSS задач 0, 1 и 2
new_09h endp
```

Между прочим, команды вида

```
        jmp     dword ptr CS:t0
```

с заменой сегмента и адресацией через сегмент команд возможны лишь при условии, что сегмент команд объявлен с разрешением исполнения и чтения (тип дескриптора 5). В наших первый программах, где сегмент команд объявлялся только исполняемым (тип 4), такая адресация привела бы к нарушению общей защиты.

При инициализации сегментов состояния задач изменено содержимое DS: поскольку поля данных задач размещены в их же процедурах, в DS загружается селектор сегмента команд:

```
;Инициализируем TSS_1
        mov     tss_1+4Ch,16    ;CS
        mov     tss_1+20h,offset task2;IP
        mov     tss_1+50h,24    ;SS
        mov     tss_1+38h,256   ;SP
        mov     tss_1+54h,16    ;DS
        mov     tss_1+48h,32    ;ES
        sti
        pushfd
        pop     EAX
        mov     dword ptr tss_1+24h,EAX;EFLAGS+IF
```

```
;Инициализируем TSS_2
        mov     tss_2+4Ch,16    ;CS
        mov     tss_2+20h,offset task3;IP
        mov     tss_2+50h,24    ;SS
        mov     tss_2+38h,512   ;SP
        mov     tss_2+54h,16    ;DS
        mov     tss_2+48h,32    ;ES
        mov     dword ptr tss_2+24h,EAX;EFLAGS+IF
```

В задачах 1 и 2 вывод сообщений на экран выполняется непосредственно из сегмента команд. Поскольку регистры DS и ES настраиваются автоматически в процессе переключения, требуется только настроить SI и DI (для выполнения команд обработки цепочек):

```
task1   proc
a1:     mov     AH,71h      ;Атрибут
        mov     CX,2        ;Счетчик внешнего цикла
a2:     push    CX          ;Сохраним его в стеке
        mov     DI,800      ;Позиция на экране
        mov     CX,mes1_len     ;Длина строки
        mov     SI,offset mes1 ;Адрес строки
        cld                 ;Пересылка вперед
a3:     lodsb               ;В AL очередной символ строки
        stosw               ;Выведем в видеобуфер с атрибутом
        loop    a3          ;Цикл по длине строки
        delay   30          ;Задержка
        pop     CX          ;Восстановим счетчик внешнего цикла
        mov     AH,17h      ;Инверсный атрибут
        loop    a2          ;Повторим вывод строки
        jmp     a1          ;Зациклим задачу
mes1    db      22,22,22,'Работает задача Task 1',22,22,22
mes1_len=$-mes1
task1   endp
```

Аналогично выглядит и задача task2.

Введем теперь в программу защиту от повторного переключения на текущую задачу. Фактически нам надо после каждого нажатия клавиш <0>, <1> или <2> выяснять, какая задача является текущей и не допускать переключения, если с клавиатуры случайно подана команда переключения на ту задачу, которая как раз выполняется. Узнать, какая задача является текущей, просто: достаточно прочитать содержимое регистра задачи TR. В нем всегда находится селектор сегмента состояния текущей задачи. Преобразуем процедуру обработчика прерывания от клавиатуры, введя в нее анализ регистра TR.

Пример 70.3. Анализ регистра состояния задачи.

Ниже приводится только модифицированный обработчик прерываний от клавиатуры. Остальной текст программы не изменился.

```
new_09h proc
        push    AX          ;Сохраним используемые
        push    DX          ;регистры
```

```
        in      AL,60h      ;Введем скен-код
        mov     DL,AL       ;Сохраним скен-код в DL
        in      AL,61h      ;Получим содержимое порта B
        or      AL,80h      ;Установкой старшего бита
        out     61h,AL      ;и последующим сбросом его
        and     AL,7Fh      ;сообщим контроллеру клавиатуры о
        out     61h,AL      ;приеме скен-кода символа
        mov     AL,20h      ;Сигнал
        out     20h,AL      ;EOI
        cmp     DL,0Bh      ;Скен-код клавиши <0>?
        je      tsk0        ;Да, на запуск задачи 1
        cmp     DL,2        ;Скен-код клавиши <1>?
        je      tsk1        ;Да, на запуск задачи 2
        cmp     DL,3        ;Скен-код клавиши <2>?
        je      tsk2        ;Да, на запуск задачи 3
        jmp     out1        ;Введено незапланированное
tsk0:   str     AX          ;Получим содержимое регистра TR
        cmp     AX,40       ;В TR уже селектор задачи 0?
        je      out1        ;Да, переключать не надо
        pop     DX          ;В TR селектор другой задачи.
        pop     AX          ;Восстановим сохраненные регистры
        jmp     dword ptr CS:t0       ;и переключим на задачу 0
        jmp     out_09h     ;На выход из обработчика
tsk1:   str     AX          ;Получим содержимое TR
        cmp     AX,48       ;В TR уже селектор задачи 1?
        je      out1        ;Да, переключать не надо
        pop     DX          ;В TR селектор другой задачи.
        pop     AX          ;Восстановим сохраненные регистры
        jmp     dword ptr CS:t1       ;и переключим на задачу 1
        jmp     out_09h     ;На выход из обработчика
tsk2:   str     AX          ;Получим содержимое TR
        cmp     AX,56       ;В TR уже селектор задачи 2?
        je      out1        ;Да, переключать не надо
        pop     DX          ;В TR селектор другой задачи.
        pop     AX          ;Восстановимсохраненные регистры
        jmp     dword ptr CS:t2       ;и переключим на задачу 2
out_09h:db      66h         ;Префикс 32-битовых операндов
        iret                ;Выход из обработчика
out1:   pop     DX          ;При вводе незапланированного
        pop     AX          ;восстановим регистры и
        jmp     out_09h     ;на выход из обработчика
t0      dw      0,40        ;Ячейки для косвенных
t1      dw      0,48        ;переходов с селекторами
t2      dw      0,56        ;TSS задач 0, 1 и 2
new_09h endp                ;
```

Если обработчик фиксирует нажатие клавиши <0> (из порта 60h получен скен-код 0Bh), выполняется чтение TR, для чего предусмотрена специальная команда str (store task register, сохранение регистра задачи). Полученное значение сравнивается с известным нам селектором сегмента состояния задачи 0 (число 40). В случае равенства осуществляется выход из обработчика без выполнения каких-либо действий. Точно так же обрабатываются команды <1> и <2>.

Статья 71

Раздельные операционные среды и таблицы локальных дескрипторов

В примерах статьи 70 все три задачи программного комплекса использовали общие сегменты команд, данных и стека. Иногда, если задачи тесно связаны друг с другом и должны передавать друг другу данные, это удобно, однако в принципе мультизадачный режим предполагает разделение операционных сред отдельных задач и защиту их друг от друга, чтобы никакая задача не могла (как непреднамеренно, так и умышленно) разрушить поля данных или команд других задач. Процессор предоставляет несколько механизмов защиты задач друг от друга. Одним из них является концепция таблиц локальных дескрипторов.

В предыдущих статьях мы познакомились с двумя типами таблиц дескрипторов: таблицей глобальных дескрипторов, в которой описываются сегменты памяти, используемые программой, и таблицей дескрипторов прерываний, которая содержит шлюзы для вызова обработчиков прерываний и исключений. И та, и другая таблица могут существовать только в одном экземпляре; все задачи (если им это разрешено) могут обращаться к этим таблицам и работать с описанными в них сегментами и программами. Помимо этих таблиц, в мультизадачной системе можно для каждой задачи построить свою таблицу локальных дескрипторов (LDT, от Local Descriptor Table), которая будет определять сегменты памяти, доступные только этой конкретной задаче. Структура таблицы локальных дескрипторов такая же, как и у глобальной таблицы, при этом сами локальные таблицы описываются в глобальной таблице с помощью системных глобальных дескрипторов.

Преобразуем пример 70.2, выделив для задач 1 и 2 отдельные сегменты памяти и организовав две таблицы локальных дескрипторов для описания этих сегментов. Структура получившегося программного комплекса приведена на рис. 71.1.

Однако в действительности наш программный комплекс имеет значительно больше программных элементов. В него входят таблицы дескрипторов (глобальных, локальных и прерываний); сегменты состояния задач; обработчики прерываний и исключений; сегмент видеобуфера. Все эти элементы должны быть описаны в тех или иных таблицах дескрипторов, за исключением "главных" таблиц GDT и IDT, которые

не входят в другие таблицы и не имеют соответствующих им дескрипторов.

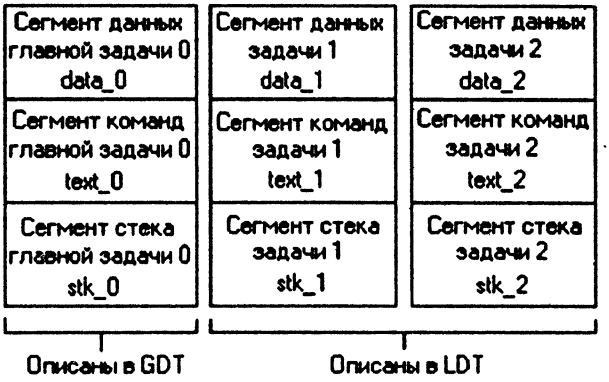


Рис. 71.1. Структура программного комплекса примера 71.1.

На рис. 71.2 изображен состав дескрипторных таблиц примера 71.1, позволяющий судить как о детальной структуре программного комплекса, так и взаимосвязях его отдельных элементов.

Рассмотрим программный комплекс (пример 71.1), который по своему содержанию и функциям в точности соответствует примерам предыдущей статьи, но имеет раздельные операционные среды для каждой задачи.

Пример 71.1. Раздельные операционные среды для каждой задачи.

```
        ...
;Структура для описания дескрипторов сегментов
descr   struc
        ...
descr   ends
;Структура для описания шлюзов ловушек
trap    struc
        ...
trap    ends
;Сегмент данных главной задачи 0
data_0  segment use16
;Таблица глобальных дескрипторов GDT
;Поля дескриптора gdt_null, дескрипторов сегментов главной задачи
;gdt_data_0, gdt_code_0 и gdt_stk_0, экрана gdt_screen, сегментов
;состояния трех задач gdt_tss_0, gdt_tss_1 и gdt_tss_2
        ...
gdt_ldt_1 descr <ldt_1_size-1,,,82h>;Селектор 64, дескриптор
                                    ;таблицы локальных дескрипторов LDT_1
gdt_ldt_2 descr <ldt_2_size-1,,,82h>;Селектор 72, дескриптор
                                    ;таблицы локальных  дескрипторов LDT_2
gdt_size=$-gdt_null                 ;Размер GDT
```

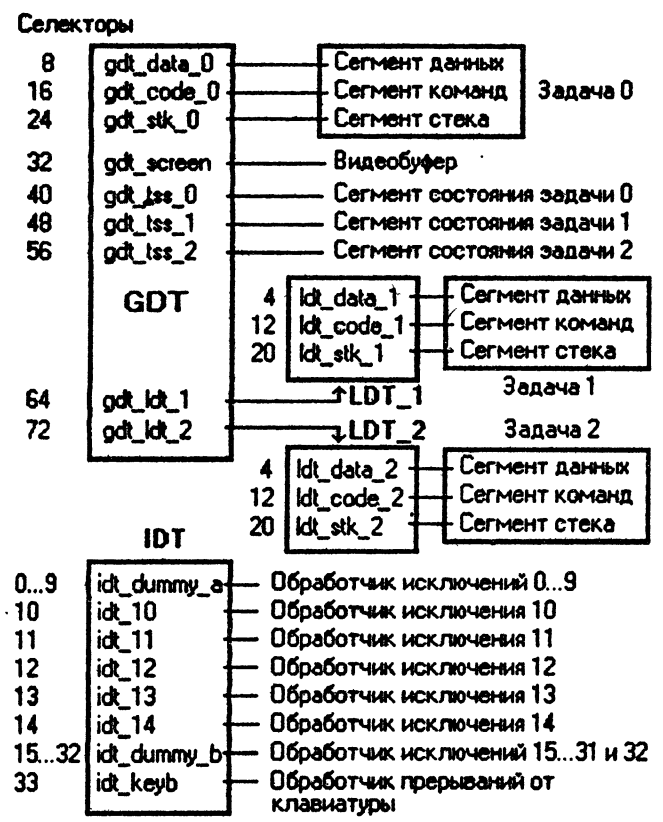


Рис. 71.2. Дескрипторные таблицы и составляющие элементы программного комплекса примера 71.1.

```
;Таблица дескрипторов прерываний IDT
;Дескрипторы исключений dummy_exc (10 штук), exc_0ah, exc_0bh,
;exc_0ch, exc_0dh, exc_0eh и снова dummy_exc (18 штук)
        ...
idt_keyb trap  <new_09h,,,8Eh,>;Шлюз прерывания
idt_size=$-idt              ;Размер IDT
pdescr   dq     0           ;Псевдодескриптор
mes db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$'
tblhex   db     '0123456789ABCDEF'
string   db '**** ****_**** ****_**** **** ****_**** ****_****'
len=$-string
tss_0    dw     72 dup (0) ;Сегмент состояния задачи TSS_0
tss_1    dw     72 dup (0) ;Сегмент состояния задачи TSS_1
tss_2    dw     72 dup (0) ;Сегмент состояния задачи TSS_2
;Первая таблица локальных дескрипторов (для задачи 1)
ldt_1    label  word
ldt_data_1 descr <data_1_size-1,,,92h>;Селектор 4, сегмент данных
```

```
                              ;data_1 задачи 1
ldt_code_1 descr <code_1_size-1,,,98h>;Селектор 12, сегмент команд
                              ;text_1 задачи 1
ldt_stk_1 descr <255,,,92h,,> ;Селектор 20, сегмент стека stk_1
                              ;задачи 1
ldt_1_size=$-ldt_1
;Вторая таблица локальных дескрипторов (для задачи 2)
ldt_2    label word
ldt_data_2 descr <data_2_size-1,,,92h>;Селектор 4, сегмент данных
                              ;data_2 задачи 2
ldt_code_2 descr <code_2_size-1,,,98h>;Селектор 12, сегмент команд
                              ;text_2 задачи 2
ldt_stk_2      descr <255,,,92h,,>;Селектор 20, сегмент стека
                              ;stk_2 задачи 2
ldt_2_size=$-ldt_2
data_0_size=$-gdt_null    ;Размер сегмента данных
data_0   ends
;Сегмент команд главной задачи 0
text_0   segment 'code' use16
         assume CS:text_0,DS:data_0
begin    label word
home_seldw     home
         dw     16          ;Селектор главной задачи
;Обработчик исключения недопустимого сегмента состояния задачи TSS
exc_0ah  proc
         mov    AX,8        ;Создадим адресуемость сегмента
         mov    DS,AX       ;данных главной задачи
         pop    EAX
         pop    EAX
         mov    SI,offset string+5
         debug
         mov    AX,0Ah
         jmp    CS:home_sel
exc_0ah  endp
;Обработчики исключений недопустимого сегмента состояния задачи
;exc_0ah, отсутствия сегмента exc_0bh, ошибки обращения к стеку
;exc_0ch, общей защиты exc_0dh, страничного нарушения exc_0eh и
;остальных исключений dummy_exc выглядят так же (за исключением
;кода номера исключения, засылаемого в AX)
         ...
;Обработчик прерываний от клавиатуры
new_09h proc
         push   AX          ;Сохраним используемые
         push   DX          ;регистры
         in     AL,60h      ;Введем скен-код
         mov    DL,AL       ;Сохраним скен-код в DL
         in     AL,61h      ;Получим содержимое порта B
         or     AL,80h      ;Установкой старшего бита
         out    61h,AL      ;и последующим сбросом его
         and    AL,7Fh      ;сообщим контроллеру клавиатуры о
         out    61h,AL      ;приеме скен-кода символа
         mov    AL,20h      ;Сигнал
         out    20h,AL      ;EOI
         cmp    DL,0Bh      ;Скен-код клавиши <0>?
         je     tsk0        ;Да, на запуск задачи 0
         cmp    DL,2        ;Скен-код клавиши <1>?
         je     tsk1        ;Да, на запуск задачи 1
```

```
        cmp     DL,3            ;Скен-код клавиши <2>?
        je      tsk2            ;Да, на запуск задачи 2
        jmp     out1            ;Введено незапланированное
tsk0:   str     AX              ;Получим содержимое TR
        cmp     AX,40           ;В TR уже дескриптор задачи 0?
        je      out1            ;Да, переключать не надо
        pop     DX              ;Восстановим регистры
        pop     AX              ;и выполним переключение на
        jmp     dword ptr CS:t0;задачу 0 через дескриптор TSS_0
        jmp     out_09h         ;На выход из обработчика
tsk1:   str     AX              ;Получим содержимое TR
        cmp     AX,48           ;В TR уже дескриптор задачи 1?
        je      out1            ;Да, переключать не надо
        pop     DX              ;Восстановим регистры
        pop     AX              ;и выполним переключение на
        jmp     dword ptr CS:t1;задачу 1 через дескриптор TSS_1
        jmp     out_09h         ;На выход из обработчика
tsk2:   str     AX              ;Получим содержимое TR
        cmp     AX,56           ;В TR уже дескриптор задачи 2?
        je      out1            ;Да, переключать не надо
        pop     DX              ;Восстановим регистры
        pop     AX              ;и выполним переключение на
        jmp     dword ptr CS:t2;задачу 2 через дескриптор TSS_2
out_09h:db      66h             ;Префикс 32-битовых операндов
        iret                    ;Выход из обработчика
out1:   pop     DX              ;При вводе незапланированного
        pop     AX              ;восстановим регистры и
        jmp     out_09h         ;на выход из обработчика
t0_addr dw      0,40            ;Ячейки для косвенных
t1_addr dw      0,48            ;переходов с селекторами
t2_addr dw      0,56            ;TSS задач 0, 1 и 2
new_09h endp
;Главная процедура главной задачи 0
main    proc
;Вычислим и занесем в таблицу глобальных дескрипторов линейные
;адреса сегментов данных data_0, команд text_0 и стека stk_0, а
;также сегментов состояния задач tss_0, tss_1 и tss_2
        ...
;Вычислим 32-битовый линейный адрес таблицы локальных дескрипторов
;ldt_1 и занесем его в дескриптор gdt_ldt_1 в таблице GDT
        mov     EAX,EBP     ;Линейный адрес сегмента данных
        add     AX,offset ldt_1;Прибавим смещение LDT_1
        mov     BX,offset gdt_ldt_1;Адрес дескриптора LDT_1
        mov     [BX].base_1,AX;Загрузим младшую часть базы
        rol     EAX,16      ;Обмен старшей и младшей половин
        mov     [BX].base_m,AL;Загрузим среднюю часть базы
;Вычислим 32-битовый линейный адрес таблицы локальных дескрипторов
;ldt_2 и занесем его в дескриптор gdt_ldt_2 в GDT
        ...
;Заполняем таблицу локальных дескрипторов LDT_1 для задачи 1
;Вычислим 32-битовый линейный адрес сегмента данных data_1 и
;занесем его в дескриптор сегмента данных в таблице LDT_1
        xor     EAX,EAX     ;Очистим EAX
        mov     AX,data_1   ;В AX адрес сегмента data_1
        shl     EAX,4       ;В EAX линейный адрес data_1
        mov     BX,offset ldt_data_1;В BX адрес дескриптора
        mov     [BX].base_1,AX ;Загрузим младшую часть базы
```

```
        rol    EAX,16      ;Обмен половин EAX
        mov    [BX].base_m,AL ;Загрузим среднюю часть базы
;Вычислим 32-битовый линейный адрес сегмента команд text_1 и
;занесем его в дескриптор сегмента команд в таблице LDT_1
        ...
;Вычислим 32-битовый линейный адрес сегмента стека stk_1 и занесем
;его в дескриптор сегмента стека в таблице LDT_1
        ...
;Аналогично заполним таблицу локальных дескрипторов LDT_2
;линейными адресами сегментов data_2, text_2 и stk_2
        ...
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        ...
;Инициализируем TSS_1
        mov    tss_1+4Ch,12   ;CS, дескриптор из LDT_1
        mov    tss_1+20h,offset task_1;IP точки входа в задачу 1
        mov    tss_1+50h,20   ;SS, дескриптор из LDT_1
        mov    tss_1+38h,256  ;SP
        mov    tss_1+54h,04   ;DS, дескриптор из LDT_1
        mov    tss_1+48h,32   ;ES, дескриптор из GDT
        sti                   ;Установим флаг прерываний
        pushfd                ;Получим слово флагов
        pop    EAX            ;в EAX
        mov    dword ptr tss_1+24h,EAX;EFLAGS+IF
        mov    tss_1+60h,64   ;Дескриптор LDT_1
;Инициализируем TSS_2
        mov    tss_2+4Ch,12   ;CS, дескриптор из LDT_2
        mov    tss_2+20h,offset task_2;IP точки входа в задачу 2
        mov    tss_2+50h,20   ;SS, дескриптор из LDT_2
        mov    tss_2+38h,256  ;SP
        mov    tss_2+54h,04   ;DS, дескриптор из LDT_2
        mov    tss_2+48h,32   ;ES, дескриптор из GDT
        mov    dword ptr tss_2+24h,EAX;EFLAGS+IF
        mov    tss_2+60h,72   ;Дескриптор LDT_2
;Подготовимся к переходу в защищенный режим
        ...
;Перепрограммируем ведущий контроллер прерываний
        ...
;Запретим все прерывания в ведомом контроллере
        ...
;Загрузим IDTR
        ...
;Переходим в защищенный режим
        ...
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
        ...
continue:
;Делаем адресуемыми данные и стек, инициализируем регистр ES
;селектором видеобуфера
        ...
;Загрузим TR
        mov    AX,40          ;Селектор сегмента TSS_0 из GDT
        ltr    AX
;Размаскируем NMI
        ...
        sti                   ;Разрешение аппаратных прерываний
```

```
;Организуем периодический вывод на экран символов
        ...
        mov    AX,0FFFFh  ;Диагностическое значение
home:   mov    SI,offset string
        debug
;Выведем на экран диагностическую строку
        ...
;Переключим режим процессора
        ...
;Теперь процессор снова работает в реальном режиме
return:
;Восстановим операционную среду реального режима (DS, SS:SP),
;разрешим аппаратные и немаскируемые прерывания, проверим
;выполнение функций DOS и завершим программу. Закроем главную
;процедуру main
        ...
code_0_size=$-begin
text_0  ends
;Сегмент stk_0 стека главной задачи 0
stk_0   segment stack 'stack'
        ...    ;256 байт
;Локальный сегмент данных data_1 задачи 1
data_1  segment
data_1_beg=$
mes1    db     22,22,22,'Работает задача Task 1',22,22,22
mes1_len=$-mes1
data_1_size=$-data_1_beg
data_1  ends
;Локальный сегмент команд задачи 1
text_1  segment 'code' use16
        assume CS:text_1,DS:data_1
task_1  proc
a1:     mov    AH,71h      ;Атрибут
        mov    CX,2        ;Счетчик внешнего цикла
a2:     push   CX          ;Сохраним его в стеке
        mov    DI,800      ;Позиция на экране
        mov    CX,mes1_len
        mov    SI,offset mes1;
        cld                ;Пересылка вперед
a3:     lodsb              ;В AL очередной символ строки
        stosw              ;Выведем в видеобуфер вместе с
                           ;атрибутом
        loop   a3          ;Цикл по длине строки
        delay  30          ;Задержка
        pop    CX          ;Восстановим внешний счетчик
        mov    AH,17h      ;Инверсный атрибут
        loop   a2          ;Повторим вывод строки
        jmp    a1          ;Зациклим задачу
task_1  endp
code_1_size=$-task_1
text_1  ends
;Локальный сегмент стека задачи 1
stk_1   segment 'stack'
        ...    ;256 байт
;Локальный сегмент данных data_2 задачи 2 аналогичен сегменту
;data_1, но в выводимом сообщении mes2 указано "Task 2" и другие
;символы обрамления (например, коды 16)
```

```
        ...
;Локальный сегмент команд задачи 2. Аналогично задаче 1. Текст
;программы тот же, но, разумеется, используются другие метки
;(например, b1, b2 и b3 вместо a1, a2 и a3), и все ссылки
;относятся к сегменту данных задачи 2 (mes2 вместо mes1 и т.д.).
;В выводимом сообщении фраза Task 1 заменена на Task 2, изменен
;атрибут выводимых символов (например, 34h и 43h вместо 71h и
;17h), а также начальная позиция строки на экране (1120 вместо
;800)
        ...
;Локальный сегмент стека задачи 2
stk_2   segment 'stack'
        ...                 ;256 байт
stk_2   ends
        end main            ;Директива окончания трансляции
```

В таблицу глобальных дескрипторов по-прежнему входят дескрипторы, описывающие сегменты данных, команд и стека главной задачи 0, а также сегмент видеобуфера и три сегмента состояния задач. Кроме этого, в GDT включены два системных дескриптора, описывающих две таблицы локальных дескрипторов LDT для задач 1 и 2. Сами эти таблицы в нашем примере находятся внутри сегмента данных главной задачи, хотя их можно выделить в самостоятельные сегменты.

Таблицу локальных дескрипторов описывает дескриптор системного сегмента, имеющий такой же формат, что и дескриптор TSS, и отличающийся от последнего только полем типа (рис. 71.3).

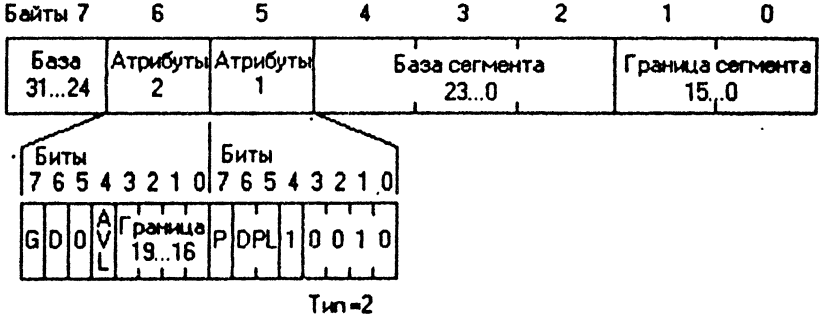


Рис.71.3. Формат системного дескриптора, описывающего LDT.

Поле границы дескрипторов LDT определяется из фактического размера LDT, а атрибут 1 имеет значение 82h: присутствующий, уровень привилегий DPL=0, дескриптор LDT (рис. 71.4).

Каждая из двух имеющихся в программе таблиц локальных дескрипторов включает три дескриптора сегментов памяти задачи: данных, команд и стека. Сами сегменты описаны в конце текста программы.

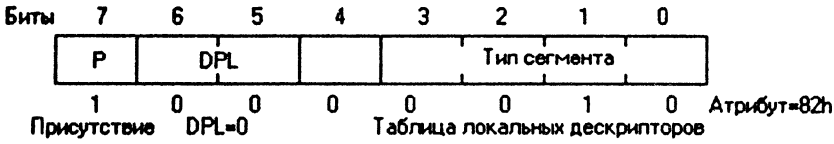


Рис. 71.4. Расшифровка значения атрибута для дескриптора LDT.

Как мы знаем, порядковый номер дескриптора в дескрипторной таблице определяет селектор, с помощью которого программа обращается к данному сегменту. Формат селектора приведен на рис. 71.5.

Рис. 71.5. Формат селектора.

Поле RPL (от Requested Privilege Level, запрошенный уровень привилегий), занимающее два младших бита селектора, может принимать значение от 0 до 3, в соответствии с числом уровней привилегий процессора. Назначение этого поля будет объяснено в следующей статье; пока примем его равным 0.

Бит TI (от Table Indicator, индикатор таблицы) равен 0, если селектор относится к глобальной таблице, и 1, если соответствующий селектору дескриптор входит в локальную таблицу. В битах 3...15 располагается порядковый номер (индекс) дескриптора в таблице дескрипторов. Индексы нумеруются с 0: 0, 1, 2 и т.д.

Сегменты, описанные в таблице глобальных дескрипторов, имели в наших программах селекторы, численно кратные 8: 0, 8, 16, 24 и т.д. Для дескрипторов, входящих в LDT, селекторы будут иметь установленным бит 2 и, соответственно, иметь значения 4, 12 и 20. При этом селекторы обеих LDT совпадают. Это не приведет к путанице, так как каждая задача будет обращаться (с помощью селекторов) только к своей локальной таблице, а под одинаковыми индексами в разных таблицах описаны физически разные сегменты.

Введение двух таблиц локальных дескрипторов требует вычисления и занесения в соответствующие дескрипторы, на этапе инициализации, во-первых, линейных адресов самих локальных таблиц (в GDT), и, во-вторых, линейных адресов всех локальных сегментов (в обеих LDT). Процедура вычисления линейных адресов уже рассматривалась.

Претерпели изменение процедуры обработки исключений. Исключения, как и аппаратные прерывания, возникают асинхронно, в непредугадываемых заранее точках программы. При передаче управления на обработчик исключения процессор изменяет только содержимое регистров CS:IP (сохраняя вектор прерванной задачи в стеке), но весь контекст задачи, т.е. содержимое сегментных и прочих регистров, оста-

ется таким, каким он был в прерванной задаче. Если обработчику исключения требуются какие-либо данные из главной задачи, как это и имеет место в нашей программе, он должен соответствующим образом настроить сегментные регистры. Это, в свою очередь, разрушит контекст прерванной задачи. Поэтому для обработчиков исключений действуют те же правила, что и для обработчиков аппаратных прерываний: при входе в обработчик следует сохранить, а перед завершением восстановить используемые им регистры. Если, однако, оформить обработчики как самостоятельные задачи и поместить в дескрипторы исключений селекторы их TSS, сохранение и восстановление контекста прерванной исключением задачи будет выполняться процессором аппаратно в процессе переключения.

В нашем случае все исключения приводят к аварийному завершению программного комплекса, поэтому сохранение регистров не требуется, однако настройка сегментного регистра DS для обеспечения обращения к сегменту данных главной задачи необходима. Засылка в DS селектора сегмента data_0 (конкретно числа 8) дает возможность обращаться затем в обработчике к строке string и использовать макрокоманду debug, которая также работает с полями данных главной задачи.

Переход на завершение задачи (последняя строка обработчика) осуществляется через двухсловную ячейку home_sel с адресом точки перехода, которую мы перенесли из сегмента данных в сегмент команд. По этой причине в команде перехода использован префикс замены сегмента (с DS на CS); можно было оставить и прежний вариант.

Как и раньше, TSS главной задачи не нуждается в инициализации; он будет автоматически заполнен контекстом задачи 0 при возврате в нее из задачи 1 или 2. В поля для сегментных регистров в TSS_1 и TSS_2 заносятся селекторы дескрипторов из локальных таблиц (численно одинаковые для обеих задач). В каждом TSS необходимо заполнить поле со смещением 60h для селектора "своей" таблицы локальных дескрипторов LDT_1 или LDT_2. Эти селекторы (64 и 72 в примере) соответствуют дескрипторам GDT, описывающим местоположение и другие характеристики локальных таблиц.

Тексты процедур task_1 и task_2 не изменились, хотя работать они будут по-другому: каждая со своим стеком и со своим сегментом данных. Внешне порядок работы с программным комплексом остался тем же: после запуска начинает работатать главная задача 0 (выводя на экран последовательные символы кодовой таблицы ASCII); нажатием клавиш <1> или <2> можно переключиться на требуемую задачу, после чего работают все три управляющие клавиши <0>, <1> и <2>, осуществляя наглядное переключение с задачи на задачу в любом порядке.

Статья 72

Уровни привилегий и защита по привилегиям

Выделение для каждой задачи, входящей в программный комплекс, отдельного локального адресного пространства (с помощью таблицы локальных дескрипторов) позволяет надежно защитить задачи друг от друга. Поскольку в процессоре имеется только один регистр LDTR, в каждый момент доступна только одна таблица локальных дескрипторов и, соответственно, только одно локальное адресное пространство. Поэтому каждая задача работает только со своими сегментами, не имея доступа к сегментам других задач. В принципе задача, если она имеет наивысший уровень привилегий, может изменить содержимое LDTR, однако в этом случае она полностью перейдет в локальное пространство другой задачи и потеряет доступ к собственным сегментам. С другой стороны, сегменты, описанные в GDT, доступны всем задачам, которые могут как обращаться к глобальным полям данным, так и вызывать подпрограммы, входящие в глобальные сегменты команд. Таким образом, концепция глобальных и локальных дескрипторов позволяет выделять каждой задаче любые ресурсы памяти, надежно защищенной на аппаратном уровне от поползновений других задач.

Учитывая исключительную важность защиты системных ресурсов в многозадачном режиме, в процессорах Intel предусмотрено еще два механизма защиты: страничная организация памяти и система уровней привилегий. В настоящей статье будут рассмотрены основы защиты задач с помощью уровней привилегий.

Каждому сегменту программы придается определенный уровень привилегий, указываемый в поле DPL (Descriptor Privilege Level, уровень привилегий дескриптора) его дескриптора. Уровни привилегий указываются во всех дескрипторах: памяти, системных и шлюзах. Уровень привилегий, указанный в дескрипторе, назначается всем объектам, входящим в данный сегмент. Так, если сегменту команд назначен уровень 3, то все процедуры этого сегмента имеют уровень защиты 3. Точно так же все данные, входящие в сегмент данных, имеют тот уровень защиты, который указан в дескрипторе этого сегмента.

Уровень привилегий выполняемого в данный момент сегмента команд называется текущим уровнем привилегий CPL (Current Privilege Level). Он определяется полем RPL селектора сегмента команд (см. рис.

71.5), загружаемого в CS. Вся система привилегий основана на сравнении CPL выполняемой программы с уровнями привилегий DPL сегментов, к которым она обращается.

Всего процессор различает 4 уровня привилегий от 0 (максимальные привилегии) до 3 (минимальные). Чем больше численное значение уровня привилегий, тем меньшими привилегиями обладает данный сегмент. Для того, чтобы избежать двусмысленности при сравнении привилегий сегментов, мы будем преимущественно называть уровни с большими привилегиями внутренними, с меньшими - внешними. Эти определения связаны с принятым изображением уровней в виде концентрических колец, называемых кольцами защиты (рис. 72.1). Во внешнем кольце располагаются сегменты, которым присвоен уровень 3; ближе к центру располагаются кольца с большими привилегиями и меньшими численными значениями уровней защиты.

Рис. 72.1. Уровни привилегий и кольца защиты.

В кольце 0 с наивысшим уровнем привилегий разумно располагать наиболее ответственную часть операционной системы - ее ядро; в кольцах 1 и 2 - остальные программы операционной системы и программного обеспечения; наконец, наименее привилегированное кольцо 3 отдается прикладным программам пользователей.

Основные правила защиты по привилегиям сводятся к следующим:

- данные из сегмента с некоторым уровнем привилегий могут быть получены только программой того же или более внутреннего уровня;

- процедура из сегмента с некоторым уровнем привилегий может быть вызвана только программой того же или более внешнего уровня, причем вызов процедуры другого уровня должен осуществляться не непосредственно, а через шлюз вызова;

- каждый уровень привилегий имеет свой стек, поэтому при переключении на другой уровень не может произойти разрушение исходного стека;

- все команды, воздействующие на механизмы сегментации и защиты (lgdt, lidt, lldt, ltr и ряд других), являются привилегированными и могут использоваться только в процедурах уровня 0, что исключает вмешательство в организацию программного комплекса со стороны прикладных программ.

Защита по уровням привилегий и защита с помощью локальных адресных пространств действуют параллельно и в какой-то мере независимо. В примере 71.1 мы защитили прикладные задачи, выделив каждой из них свое локальное адресное пространство, однако защита по уровням привилегий в этом примере отсутствовала, так как уровни привилегий всех дескрипторов комплекса были равны 0 (рис. 72.2). В результате прикладные задачи, входящие в комплекс, могли бы, например, изменить содержимое глобальных или локальных дескрипторных таблиц, т.е. вмешаться в саму организацию комплекса. Если же разместить сегменты прикладных программ на более внешнем уровне, например, уровне 3 (рис. 72.3), они, во-первых, потеряют права на выполнение привилегированных команд и, во-вторых, смогут обращаться только к тем процедурам управляющей программы, которым соответствуют шлюзы вызова, т.е. к процедурам, специально предназначенным для вызова из внешних колец. К полям данных управляющей программы, в частности, к таблицам дескрипторов или сегментам состояния задач доступ из прикладных программ будет закрыт.

Еще одним элементом организации программного комплекса, действующим независимо как от уровней привилегий, так и от разделения адресного пространства на глобальные и локальные области, является выделение системных или прикладных программ в отдельные задачи (именно этот случай изображен на рис. 72.3). В примере 71.1 управляющая и прикладные программы образовывали отдельные задачи со своими сегментами состояния задач, что обеспечило возможность использования удобных средств переключения задач с автоматическим сохранением их контекстов. Однако разделение на задачи не обязательно. Все программные элементы комплекса, расположенные, возможно, на разных уровнях привилегий и работающие в различных (глобальном и локальных) адресных пространствах, могут входить в единую задачу. При этом организация комплекса упрощается, хотя снижается его гибкость.

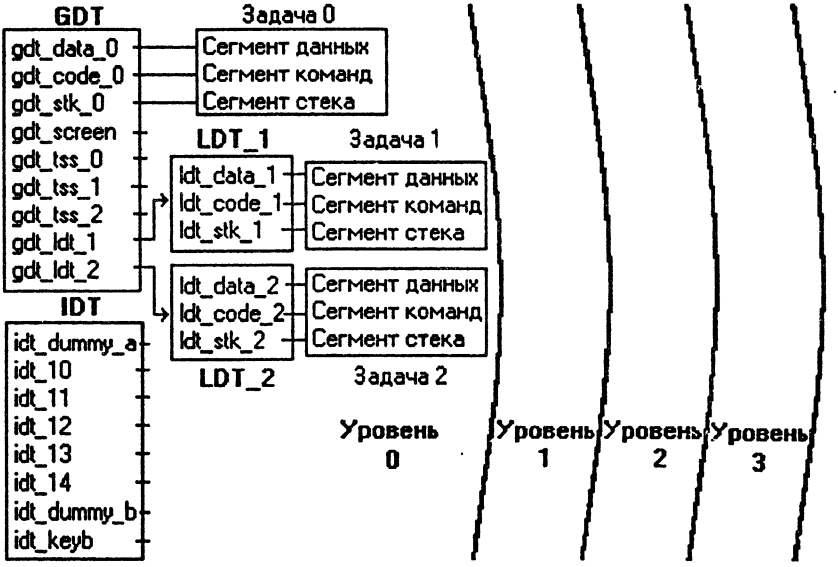


Рис. 72.2. Размещение всего программного комплекса в кольце 0.

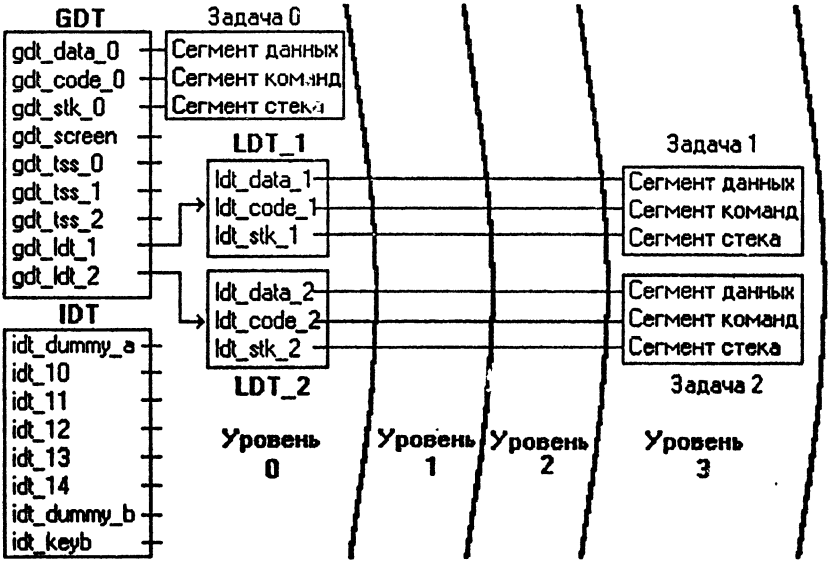


Рис. 72.3. Размещение управляющей программы в кольце 0, а прикладных программ в кольце 3.

При построении многоуровневой системы особую сложность представляет организация передачи управления с уровня на уровень. Как уже отмечалось, обычно в кольце 0 располагают программы операционной системы, управляющие вычислительным процессом; в кольце 3 размещаются прикладные программы. В этом случае сначала начинает работать управляющая программа, которая готовит операционную среду защищенного режима для себя и прикладных программ, после чего переводит процессор в защищенный режим. Затем управление передается прикладной программе, которая часть работы может выполнять самостоятельно, однако должна иметь возможность обратиться к операционной системе для вызова определенных системных функций, например, вывода на экран или в файлы, завершения работы и передачи управления системе и т.д.

Для того, чтобы разобраться в не очень простой технике передачи управления процедурам других уровней, рассмотрим сначала пример однозадачного программного комплекса, в котором все программные элементы входят в одну единственную задачу. Наш комплекс содержит управляющую программу, имитирующую некоторые функции операционной системы, и прикладную программу, для которой выделяется локальное адресное пространство. Управляющая программа вместе со всеми "системными" областями (таблицами дескрипторов, процедурами обработки исключений и проч.) располагается на уровне 0. Прикладная программа, содержащая сегменты команд, данных и стека, вынесена на уровень 3. Будучи инициирована управляющей, она выводит на экран контрольное сообщение и обращается к двум сервисным функциям управляющей программы. Первая функция является упрощенным аналогом функции 13h прерывания 10h системы BIOS (вывод на экран строки); вторая функция позволяет завершить прикладную программу и вернуться в управляющую программу (в MS-DOS эти действия выполняет функция 4Ch).

Пример 72.1. Межуровневая передача управления.

```
        ...
;Структура для описания дескрипторов сегментов
        ...
;Структура для описания шлюзов ловушек
        ...
data_ctrl segment use16  ;(1)Сегмент данных управляющей программы
;Таблица глобальных дескрипторов: gdt_null(селектор 0),
;gdt_data(8), gdt_code(16), gdt_stack(24), gdt_screen(32),
;gdt_tss(40), gdt_ldt(48)
        ...
;Таблица дескрипторов прерываний IDT
;Дескрипторы исключений dummy_exc (10 штук), exc_0ah, exc_0bh,
;exc_0ch, exc_0dh и exc_0eh
        ...
;Таблица локальных дескрипторов
```

```
ldt      label word          ;(2)
ldt_data descr <data_task_size-1,0,0,0F2h>;(3)Селектор 4, сегмент
                             ;данных прикладной программы, DPL=3
ldt_code descr <code_task_size-1,0,0,0FAh>;(4)Селектор 12+3,
                             ;сегмент команд прикладной программы
                             ;с разрешением чтения, DPL=3
ldt_stack descr <255,0,0,0F2h>;(5)Селектор 20+3, сегмент стека
                             ;прикладной программы, DPL=3
ldt_gate_1 trap <srv_1,16,5,0ECh>;(6)Селектор 28, шлюз процедуры
                             ;srv_1, входящей в управляющую программу
ldt_gate_2 trap <srv_2,16,0,0ECh>;(7)Селектор 36, шлюз процедуры
                             ;srv_2, входящей в управляющую программу
ldt_size=$-ldt               ;(8)Размер LDT
tss      dw     72 dup (0) ;(9)TSS единой задачи
pdescr   dq     0           ;(10)Псевдодескриптор
mesctrl db      ' Управляющая программа приняла управление ';(11)
mesctrl_len=$-mesctrl        ;(12)
tblhex   db     '0123456789ABCDEF' ;(13)
string db '**** ****-**** ****-**** **** ****-**** ****-****';(14)
len=$-string                 ;(15)
data_ctrl_size=$-gdt_null;(16)Размер сегмента данных
data_ctrl ends               ;(17)
code_ctrl segment 'code' use16;(18)Сегмент команд управляющей
                             ;программы
         assume CS:code_ctrl,DS:data_ctrl;(19)
begin    label word          ;(20)Начало сегмента команд
home_sel dw     home         ;(21)Адрес возврата из исключения
         dw     16           ;(22)Селектор сегмента команд
;Обработчики исключений exc_0ah, exc_0bh, exc_0ch, exc_0dh,
;exc_0eh и dummy_exc
         ...
;Сервисная функция 1 управляющей программы. Функция осуществляет
;вывод на экран строки текста. Входные параметры передаются через
;стек уровня 0 в виде 5 четырехбайтовых слов, в которых фактически
;используются только младшие половины (а в последнем параметре
;только второй байт). В приведенном ниже списке параметры
;расположены в порядке увеличения их адресов в стеке:
;Относительный адрес строки
;Сегмент строки
;Длина строки
;Позиция на экране
;Атрибут символов в AH
srv_1    proc   far          ;(23)Дальняя процедура
         call   outscr       ;(24)Вызов вложенной процедуры
         db     66h          ;(25)Возврат в вызывающую программу
         ret    20           ;(26) со снятием со стека 20 байтов
                             ;(5 двухсловных параметров)
srv_1    endp                ;(27)
;Процедура вывода на экран заданного текста
outscr   proc                ;(28)Ближняя процедура
         mov    EBP,ESP      ;(29)Текущий указатель стека
         add    EBP,10       ;(30)Сместимся к началу параметров
         mov    ESI,[BP]+0   ;(31)Относительный адрес строки
         mov    EAX,[BP]+4   ;(32)Сегмент строки
         mov    DS,AX        ;(33)Настроим DS на сегмент строки
         mov    ECX,[BP]+8   ;(34)Длина строки
         mov    EDI,[BP]+12; (35)Позиция на экране
```

```
        mov    EAX,[BP]+16;(36)Атрибут символов в AH
        cld                ;(37)Движение вперед
scr5:   lodsb              ;(38)Получим байт из строки
        stosw              ;(39)Выведем символ с атрибутом на
        loop   scr5        ;(40)Повторим ECX раз
        ret                ;(41)Возврат в srv1
outscr  endp               ;(42)
;Сервисная функция 2 управляющей программы
srv_2   proc   far         ;(43)Дальняя процедура
        mov    AX,8        ;(44)Адресуемость данных
        mov    DS,AX       ;(45)управляющей программы
        mov    SP,256      ;(46)Восстановим указатель стека
        mov    EAX,5F4B5F4Fh;(47)Выведем на экран цветную
        mov    ES:[2880],EAX;(48)сигнатуру OK
        delay  100         ;(49)Задержка
        jmp    go          ;(50)Переход на продолжение управляющей
                           ;программы (в данном случае на
                           ;завершение всей задачи)
srv_2   endp               ;(51)
main    proc               ;(52)
;Вычислим и занесем в таблицу глобальных дескрипторов линейные
;адреса сегментов управляющей программы: данных data_ctrl, команд
;code_ctrl, стека stack_ctrl, а также сегмента состояния задачи
;tss_ctrl и таблицы локальных дескрипторов gdt_ldt
        ...
;Вычислим и занесем в таблицу локальных дескрипторов линейные
;адреса сегментов прикладной программы: данных data_task в
;дескриптор ldt_data, команд code_task в дескриптор ldt_code и
;стека stack_task в дескриптор ldt_stack
        ...
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        ...
;Подготовимся к переходу в защищенный режим
        ...
;Загрузим IDTR
        ...
;Переходим в защищенный режим
        ...
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
        ...
continue:                  ;(53)
;Делаем адресуемыми данные и стек, инициализируем регистр ES
;селектором видеобуфера
        ...
;Выведем на экран сообщение управляющей программы
        mov    CX,mesctrl_len ;(54)
        mov    SI,offset mesctrl;(55)
        mov    AH,3Eh      ;(56)
        mov    DI,1920     ;(57)
scr1:   lodsb              ;(58)
        stosw              ;(59)
        loop   scr1        ;(60)
        delay  100         ;(61)Задержка для наглядности
;Подготовим вызов прикладной программы
;Загрузим регистр LDTR адресом LDT
        mov    AX,48       ;(62)Селектор LDT в GDT
```

```
        lldt   AX           ;(63)
;Инициализируем TSS задачи
        mov    tss+4,ESP    ;(64)Указатель стека кольца 0
        mov    tss+8,24     ;(65)SS кольца 0
;Загрузим TR селектором TSS задачи
        mov    AX,40        ;(66)Селектор TSS в GDT
        ltr    AX           ;(67)
;Подготовим стек для команды ret перехода на прикладную программу
        mov    EAX,23       ;(68)Селектор сегмента стека
        push   EAX          ;(69)прикладной программы в LDT
        mov    EAX,256      ;(70)Указатель стека
        push   EAX          ;(71)прикладной программы
        mov    EAX,15       ;(72)Селектор сегмента команд
        push   EAX          ;(73)прикладной программы
        mov    AX,offset task;(74)Входное значение EIP
        push   EAX          ;(75)прикладной программы
;Вызовем прикладную программу
        db     66h          ;(76)"Возврат из подпрограммы"
        retf                ;(77)в прикладную программу
;Продолжение управляющей программы после возврата из прикладной
go:     mov    AX,0FFFFh    ;(78)Диагностическое значение
home:   mov    SI,offset string;(79)
        debug               ;(80)
;Выведем на экран диагностическую строку
        ...
;Вернемся в реальный режим
        ...
;Теперь процессор снова работает в реальном режиме
return:                     ;(81)
;Восстановим операционную среду реального режима (DS, SS:SP),
;разрешим аппаратные и немаскируемые прерывания, завершим
;программу. Закроем главную процедуру main, закроем сегмент команд
;управляющей программы
        ...
stk_ctrl segment stack 'stack';(82)
        db     256 dup ('0');(83)
stk_ctrl ends               ;(84)
;Сегменты прикладной программы
data_task segment           ;(85)
data_beg=$                  ;(86)
mes1 db 16,16,16,'Работает прикладная программа',17,17,17;(87)
mes1_len=$-mes1             ;(88)
mes2 db 'Вызываем сервисную функцию 1 управляющей программы';(89)
mes2_len=$-mes2             ;(90)
data_task_size=$-data_beg;(91)
data_task ends              ;(92)
code_task segment 'code' use16;(93)
        assume CS:code_task,DS:data_task;(94)
task    proc                ;(95)
;Выведем на экран сообщение о переходе в прикладную программу
        mov    AX,4         ;(96)Селектор сегмента данных
        mov    DS,AX        ;(97)прикладной программы в LDT
a1:     mov    AH,61h       ;(98)Атрибут символов
        mov    DI,2240      ;(99)Позиция на экране
        mov    CX,mes1_len;(100)Длина строки
        mov    SI,offset mes1;(101)Адрес строки
        cld                 ;(102)Движение вперед
```

```
a3:     lodsb                   ;(103)Получим байт из строки
        stosw                   ;(104)Выведем символ с атрибутом
        loop    a3              ;(105)Повторим ECX раз
        delay   100             ;(106)Задержка для наглядности
;Подготовим параметры в стеке и вызовем сервисную функцию 1
;управляющей программы вывода на экран сообщения "средствами DOS"
        mov     AH,24h          ;(107)В AH атрибут символов
        push    EAX             ;(108)Двойное слово в стек
        mov     AX,2560         ;(109)Позиция на экране
        push    EAX             ;(110)Двойное слово в стек
        mov     AX,mes2_len;(111)Длина строки
        push    EAX             ;(112)Двойное слово в стек
        mov     AX,DS           ;(113)Сегментный адрес строки
        push    EAX             ;(114)Двойное слово в стек
        mov     AX,offset mes2;(115)Относительный адрес строки
        push    EAX             ;(116)Двойное слово в стек
        db      9Ah             ;(117)Код команды call far ptr
        dw      0               ;(118)Игнорируемое смещение шлюза
        dw      31              ;(119)Селектор шлюза вызова процедуры
                                ;srv_1 в LDT
        delay   100             ;(120)Задержка для наглядности
;Вызовем сервисную функцию 2 управляющей программы (параметров
;нет)
        db      9Ah             ;(121)Код команды call far ptr
        dw      0               ;(122)Игнорируемое смещение шлюза
        dw      39              ;(123)Селектор шлюза вызова процедуры
                                ;srv_2 в LDT
task    endp                    ;(124)
code_task_size=$-task           ;(125)
code_task       ends            ;(126)
stack_task segment 'stack';(127)
        ...                     ;Завершающие строки сегмента
```

На рис. 72.4 изображена структурная схема программного комплекса примера 72.1 с указанием состава дескрипторных таблиц, численных значений селекторов сегментов и распределения сегментов по кольцам защиты.

В таблицу глобальных дескрипторов включены дескрипторы всех трех сегментов управляющей программы, а также сегментов видеобуфера, состояния задачи и таблицы локальных дескрипторов. Сегменты программы, TSS и LDT имеют в байте атрибутов, как и раньше, значение поля DPL, равное 0, и располагаются, таким образом, в нулевом, наиболее привилегированном кольце защиты. В дескрипторе видеобуфера поле DPL содержит число 3 (рис. 72.5), т.е. сегмент видеобуфера расположен на уровне 3 с минимальными привилегиями. Это разрешает доступ к нему из программ всех четырех уровней. Для того, чтобы сделать обращение к видеобуферу практически возможным, он описан в таблице глобальных дескрипторов и размещен, таким образом, в глобальном адресном пространстве.

В единственной таблице локальных дескрипторов описаны сегменты данных, команд и стека прикладной программы, а также два шлюза вы-

зова процедур управляющей программы ldt_gate_1 и ldt_gate_2. Все сегменты прикладной программы, как и видеобуфер, имеют DPL=3 и размещаются на уровне 3.

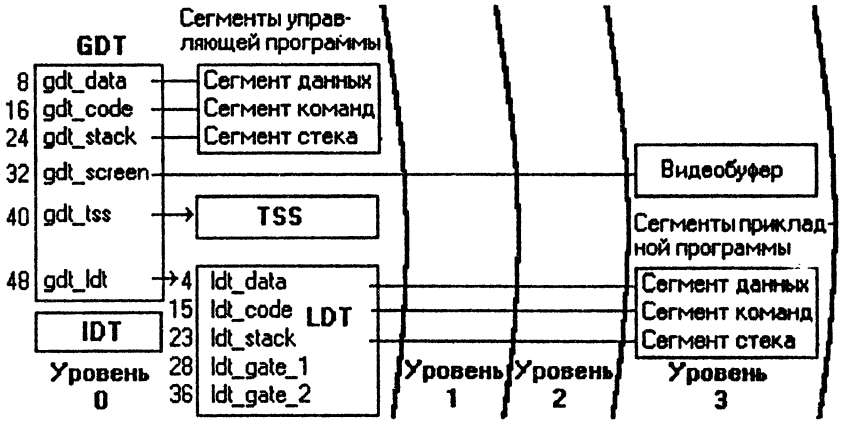


Рис. 72.4. Структурная схема программного комплекса примера 72.1.

Рис. 72.5. Расшифровка значения атрибута сегмента видеобуфера.

Шлюзы вызова, описывающие процедуры srv_1 и srv_2, содержат относительные адреса точек входа в эти процедуры (вообще говоря, 32-битовые, но в нашем случае фактически 16-битовые), селектор сегмента команд управляющей программы (16), поле счетчика (5 для первого шлюза и 0 для второго), а также байт атрибута, равный ECh (рис. 72.6).

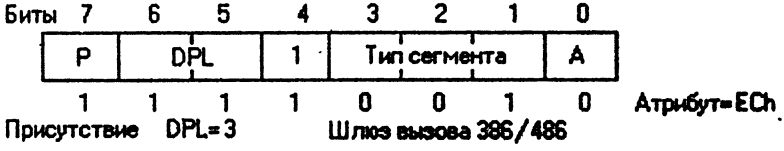


Рис. 72.6. Расшифровка значения атрибута шлюза вызова.

На первый взгляд кажется, что DPL шлюза, описывающего процедуру кольца 0, должен быть равен 0, однако для шлюзов вызова действуют особые правила назначения уровня привилегий. DPL шлюза определяет не уровень привилегий вызываемой процедуры, а то, какими привилегиями должна обладать вызывающая программа. Поскольку мы будем вызывать процедуры srv_1 и srv_2 из кольца 3, то и дескриптор шлюза должен иметь DPL, не меньший 3, т.е. просто 3.

Назначение поля счетчика будет разъяснено позже.

Как уже отмечалось, наш программный комплекс, несмотря на наличие в нем нескольких сегментов команд, расположенных к тому же на разных уровнях привилегий, представляет собой единую задачу. Поэтому в сегменте данных управляющей программы зарезервировано поле для одного сегмента состояния задачи tss. Заполнен этот сегмент будет позже, по ходу выполнения программы.

В сегмент данных включена символьная строка mesctrl для контрольного сообщения управляющей программы. Остальные поля данных были описаны в предыдущих примерах.

Сервисная функция 1, посылающая в видеобуфер строку текста, ради наглядности, а также для демонстрации внутрисегментных вызовов содержит вложенную процедуру outscr, которая, собственно, и выводит строку на экран. Сама процедура srv_1 только принимает управление из программы внешнего кольца и после отработки процедуры outscr возвращает управление назад в прикладную программу. Сервисная функция 1 требует для своей работы ряда параметров, которые ей передаются через стек. Правила передачи параметров будут рассмотрены позже.

Сервисная функция 2 предназначена для завершения работы прикладной программы и возврата в управляющую. В процедуре srv_2 заново инициализируются сегментный регистр данных и указатель стека, на экран выводится символьная строка 'OK' и после небольшой задержки управление передается на завершение всего программного комплекса (метка go).

Главная процедура main, олицетворяющая управляющую программу, прежде всего выполняет обычные действия по заполнению дескрипторных таблиц GDT и LDT, загрузке регистров GDTR и IDTR и переходу в защищенный режим.

В защищенном режиме инициализируются сегментные регистры CS (с помощью команды far jmp), DS, SS и ES, на экран выводится контрольное сообщение, после чего управляющая программа приступает к подготовке перехода на выполнение прикладной программы. Сам вызов прикладной программы осуществляется, как это ни странно, командой ret. Для того, чтобы понять, как команда ret может вызвать на выполнение процедуру, нам придется подробно рассмотреть механизм межуровневого вызова подпрограмм с помощью команды call и шлюза вызова.

Передача управления в заданную точку программы в пределах одного уровня осуществляется с помощью команд call и jmp, причем при переходе в тот же сегмент команд используются ближние формы этих команд, а при переходе в другой сегмент - дальние. Переход на другой уровень возможен только с помощью команды дальнего вызова call; команда jmp передать управление на другой уровень не может. В качестве аргумента команды call должен использоваться не фактический адрес вызываемой процедуры, а селектор шлюза вызова, который, в свою очередь, содержит адрес вызываемой процедуры. Команда дальнего прямого вызова имеет, как известно, следующий формат:

```
db      9Ah
dw      offset
dw      segment
```

Здесь 9Ah - код операции, *offset* - смещение вызываемой процедуры, *segment* - ее сегментный адрес. При вызове процедуры через шлюз вызова в поле сегмента указывается, как это и положено в защищенном режиме, селектор шлюза, а поле смещения процессором игнорируется и может содержать любое значение.

В шлюзе вызова (рис. 72.7) указывается полный адрес (селектор + смещение) вызываемой процедуры, уровень привилегий шлюза, назначение которого отмечалось выше, и (в поле счетчика) число двухсловных параметров, передаваемых вызываемой процедуре.

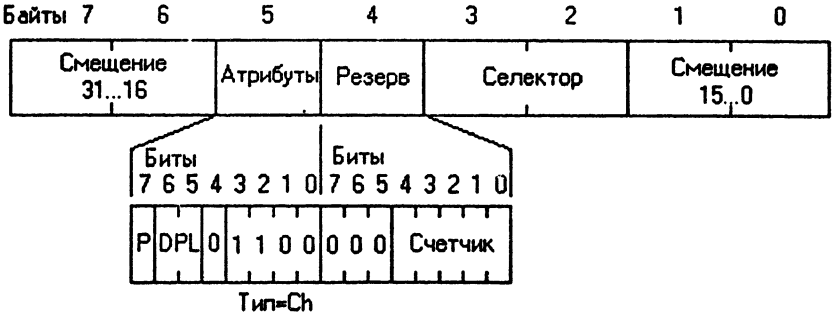


Рис. 72.7. Формат шлюза вызова.

Вызов процедуры внутреннего кольца через шлюз вызова не только передает управление на вызываемую процедуру со сменой текущего уровня привилегий, но и копирует указанное в шлюзе число двухсловных параметров из стека внешнего уровня на стек внутреннего. Кадр стека (SS:ESP) внутреннего уровня берется процессором из сегмента состояния задачи. Поэтому перед вызовом процедуры внутреннего

уровня следует, во-первых, заполнить в TSS задачи поля для SS и ESP тех внутренних уровней, процедуры которых предполагается вызывать через шлюзы вызова, и, во-вторых, передать процессору селектор TSS (с помощью команды ltr). Эти действия выполняются в рассматриваемом примере в предложениях 64-65 и 66-67.

В нашем примере для привилегированных программ используется только одно, нулевое кольцо, поэтому в TSS заполняются лишь поля для кадра стека уровня 0. Если планируется вызывать процедуры из колец 1 или 2, в TSS следует записать исходные значения SS:ESP и для этих уровней (см. формат TSS на рис. 67.3).

Процессор, инициализировав стек внутреннего уровня с помощью данных, полученных из TSS, сохраняет в новом стеке значения SS и ESP, чтобы иметь возможность после завершения внутренней процедуры переключиться на стек внешнего уровня. Далее в стек внутреннего уровня из внешнего стека копируются двухсловные параметры, число которых (от 0 до 31) задано в шлюзе вызова. Наконец, во внутренний стек помещается, как это всегда делается при дальнем вызове, адрес возврата в вызывающую программу внешнего уровня (значения CS:EIP). В результате указатель стека внутреннего уровня оказывается смещенным относительно первоначального положения на p*4+16 байтов, где p - число передаваемых параметров. На рис. 72.8 изображены оба стека, участвующие в операции вызова процедуры внутреннего уровня с передачей 5 параметров.

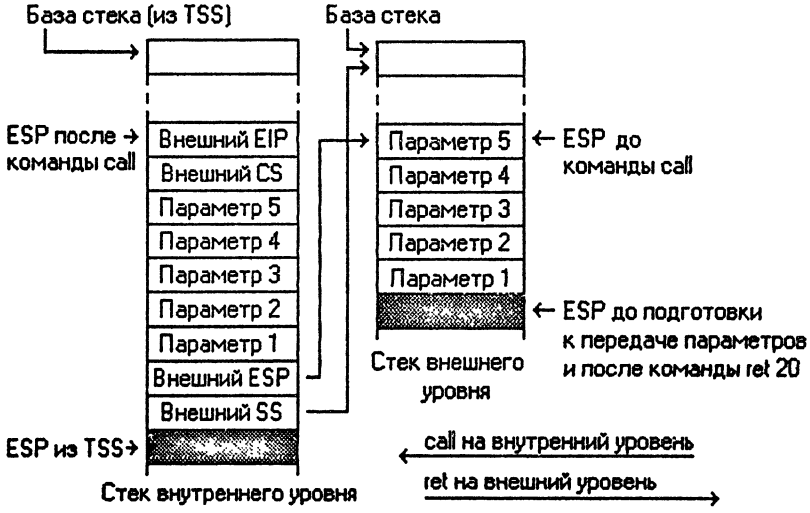


Рис. 72.8. Вызов процедуры внутреннего уровня с передачей параметров через стек.

Таким образом, вызывающая программа внешнего уровня должна поместить в стек заданное число двухсловных параметров и выполнить команду дальнего вызова с указанием в качестве сегментного адреса перехода селектор шлюза вызова (предложения 107..119).

В вызываемой процедуре внутреннего уровня параметры не следует снимать со стека, чтобы не потерять адрес возврата. В рассматриваемом примере вызываемая процедура srv_1 сразу же вызывает обычной командой ближнего прямого вызова вложенную процедуру outscr, что приводит к смещению указателя стека еще на одно слово (2 байта). В результате параметр 5 оказывается "погребенным" в стеке на глубине 10 байтов. Соответственно в процедуре outscr для адресации к этому параметру к текущему указателю стека после переноса его в базовый регистр EBP прибавляется 10 (предложения 29-30).

Команда ret ближней процедуры outscr снимает со стека адрес возврата и возвращается в процедуру srv_1. В этой процедуре выполняется команда дальнего возврата ret. Для того, чтобы в процессе возврата со стеков были сняты параметры, в качестве аргумента команды ret следует указать суммарное число байтов в параметрах (у нас 20). Для того, чтобы команда ret снимала со стека двойные слова, в 16-битовом сегменте команд ей следует предпослать префикс 66h замены размера операнда (предложения 25-26).

Вторая сервисная функция управляющей программы srv_2 не требует параметров. Соответственно поле для счетчика в ее шлюзе содержит 0 (предложение 7), при вызове этой процедуры параметры в стек не заносятся (предложения 121...123), а возврат из этой процедуры выполняется командой ret без параметров. В нашем примере возврат в прикладную программу из процедуры srv_2 не предусмотрен.

Рассмотрим оставшийся неясным вопрос, как из управляющей программы, работающей на уровне привилегий 0, перейти в прикладную программу. Как уже отмечалось, команда call допускает переход только на внутренний уровень, но не на внешний. Однако команда ret как раз наоборот, позволяет вернуться на внешний уровень. Этой командой можно воспользоваться и для первого перехода в прикладную программу внешнего уровня, если правильным образом настроить стек. Как видно из рис. 72.8, команда ret возврата на внешний уровень (при отсутствии параметров) ожидает наличие в стеке 4 двойных слов: полного указателя адреса перехода и кадра внешнего стека. Если сформировать в стеке такую структуру и выполнить команду дальнего возврата ret, управление будет передано программе внешнего уровня. Чтобы разобраться в деталях такой передачи управления рассмотрим более подробно механизмы защиты по привилегиям.

Как уже отмечалось, каждому сегменту приписывается некоторый уровень привилегий, определяемый полем DPL в его дескрипторе. Ана-

логичное поле имеется в любом селекторе, где оно называется RPL и определяет запрашиваемый уровень привилегий, т.е. уровень привилегий запросчика (не запрашиваемого сегмента!). Указание в поле RPL значения, превышающего значение CPL, позволяет как бы уменьшить привилегии выполняемой программы и тем самым запретить ей обращение к сегменту, использование которого в противном случае было бы разрешено. Такое изменение привилегий источника запроса практикуется при передаче селекторов в качестве параметров в процедуры внешних уровней. Обычно же в поле RPL селектора указывают либо значение CPL текущей программы, либо 0. В этом случае поле RPL не влияет на механизм защиты по привилегиям.

Если программа делает попытку обращения к некоторому сегменту данных (загружая в один из сегментных регистров соответствующий этому сегменту селектор), процессор сравнивает текущий уровень привилегий программы с уровнем привилегий адресуемого сегмента. Программе разрешается доступ к сегменту лишь того же или более внешнего уровня привилегий. Другими словами, при обращении к данным должно выполняться условие CPL<=DPL.

Команды дальних вызовов и переходов call и jmp (как прямые, так и косвенные), осуществляя передачу управления в другой сегмент, изменяют значение селектора, хранящегося в CS. При этом в принципе возникает возможность смены текущего уровня привилегий. Процессор следит за этим процессом и разрешает загрузку нового селектора только в том случае, если значение DPL дескриптора, связанного с этим селектором, точно равно значению CPL. Таким образом, разрешаются переходы и вызовы подпрограмм только из того же кольца, в котором находится запрашивающая программа.

Очевидно, что это ограничение носит слишком жесткий характер, так как оно делает невозможным обращение прикладных программ к сервису операционной системы. Для смягчения этого ограничения введена возможность вызова процедур из внутренних колец защиты через шлюзы вызова. Поскольку сами шлюзы обычно располагаются в сегментах нулевого кольца, они формируются операционной системой или другой управляющей программой и недоступны программам пользователя. Таким образом, доступ к средствам операционной системы жестко контролируется: прикладная программа может осуществлять вызовы только заранее обусловленных процедур внутренних колец. Обращение к процедурам из внешних колец запрещено безусловно.

Рассмотрим теперь особенности передачи управления процедуре внешнего кольца с помощью команды ret (предложения 68...77).

В предложениях 72-73 в стек загружается селектор сегмента команд прикладной программы. В процессе выполнения команды ret этот селектор будет загружен в регистр CS, в результате чего его поле RPL

определит текущий уровень привилегий CPL. Прикладная программа расположена на уровне 3 и должна работать с текущим уровнем привилегий CPL, равным 3. Поэтому поле RPL в загружаемом селекторе тоже должно иметь значение 3. Дескриптор сегмента команд находится в таблице локальных дескрипторов (бит селектора TI=1), и расположен на втором месте (индекс дескриптора в селекторе равен 1); в результате численное значение селектора оказывается равным 15 (1*8+4+3).

При определении значения селектора стека прикладной программы надо учитывать, что поскольку для каждого уровня в системе предусмотрен свой стек, процессор при загрузке регистра SS в процессе перехода с уровня на уровень выполняет такую же жесткую проверку привилегий, как и при вызове процедур. Поэтому для сегмента стека должно выполняться правило RPL=DPL=CPL. Это дает значение селектора сегмента стека 23 (2*8+4+3).

Загрузив в стек нулевого уровня три двойных слова (SS, ESP и EIP), можно выполнить команду ret дальнего возврата из процедуры в режиме 32-битовых операндов (префикс 66h). Так как процедура main, в которую входит эта команда, объявлена по умолчанию ближней (в операторе proc отсутствует описатель far) необходимо указание в явной форме диапазона действия команды (мнемоника retf).

Между прочим, для первого перехода на прикладную программу можно было воспользоваться и командой iret. Для нее, разумеется, требуется несколько иное заполнение стека (рис. 72.9).

Рис. 72.9. Заполнение стека для перехода на внешний уровень командой iret.

Помимо подготовки стека, для правильного выполнения команды iret необходимо позаботиться о сбросе бита NT в регистре флагов. Выше отмечалось, что при загрузке компьютера бит NT может быть установлен. В этом случае команда iret (без предшествующего аппаратного прерывания) попытается выполнить обратный межзадачный возврат через TSS, что в нашем случае приведет к аварии.

Соответствующий фрагмент программы будет выглядеть следующим образом:

```
;Сбросим бит NT для команды iret перехода на прикладную программу
        pushf               ;Перешлем содержимое регистра
        pop     AX          ;флагов через стек в AX
        and     AX,0BFFFh   ;Сбросим бит 14 (4000h)
        push    AX          ;И через стек вернем
        popf                ;в регистр флагов
;Подготовим стек для команды ret перехода на прикладную программу
        mov     EAX,23      ;Селектор сегмента стека
        push    EAX         ;прикладной программы в LDT
```

```
        mov     EAX,256     ;Указатель стека
        push    EAX         ;прикладной программы
        pushfd              ;EFLGS в стек
        mov     EAX,15      ;Селектор сегмента команд
        push    EAX         ;прикладной программы
        mov     AX,offset task;Входное значение EIP
        push    EAX         ;прикладной программы
;Вызовем прикладную программу
        db      66h         ;"Возврат из подпрограммы"
        iret                ;в прикладную программу
```

Статья 73

Задачи в кольцах защиты

В предыдущей статье для демонстрации основных принципов защиты по привилегиям была использована программа, представляющая собой, несмотря на относительную сложность, одну задачу с единственным сегментом состояния TSS. При построении программных комплексов, в которых имеются четко выраженные управляющие (системные) и прикладные элементы, естественнее системные программы оформить в виде управляющей задачи, а прикладные программы выделить в одну или несколько прикладных задач. Управляющую задачу можно разместить в нулевом кольце защиты, а прикладные, например, в кольце 3.

Внесем в пример 72.1 изменения, необходимые для организации в программе двух задач - управляющей и прикладной. Изменения затронут прежде всего механизмы взаимодействия управляющих и прикладных элементов. Для передачи управления прикладной задаче можно воспользоваться средствами переключения задач с сохранением их контекстов; с другой стороны, вызов системных функций из прикладной задачи естественнее реализовать по-прежнему с помощью шлюзов вызова.

Пример 73.1. Задачи и привилегии.

```
        ...
;Структура для описания дескрипторов сегментов
        ...
;Структура для описания шлюзов ловушек
        ...
data_ctrl segment use16  ;(1)Сегмент данных управляющей задачи
;Таблица глобальных дескрипторов: gdt_null (селектор 0),
```

```
;gdt_data(8), gdt_code(16), gdt_stack(24), gdt_screen(32)
        ...
gdt_tss_ctrl descr <103,,,89h>;(2)Селектор 40, дескриптор TSS
                               ;управляющей задачи
gdt_tss_task descr <103,,,89h>;(3)Селектор 48, дескриптор TSS
                               ;прикладной задачи
gdt_ldt descr <ldt_size-1,,,82h>;(4)Селектор 56, дескриптор LDT
;Таблица дескрипторов прерываний IDT. Дескрипторы исключений
;dummy_exc (10 штук), exc_0ah, exc_0bh, exc_0ch, exc_0dh и exc_0eh
        ...
;Таблица локальных дескрипторов LDT. Дескрипторы сегментов
;прикладной задачи ldt_data (селектор 4), ldt_code (селектор 15),
;ldt_stack (селектор 23)
        ...                    ;См. пример 72.1)
ldt_gate_1 trap <srv_1,16,5,0ECh,0>;(5)Селектор 28, шлюз процедуры
                               ;srv_1, входящей в управляющую задачу
tss_ctrl dw    72 dup (0) ;(6)TSS управляющей задачи
tss_task dw    72 dup (0) ;(7)TSS прикладной задачи
start_task dw  0,48       ;(8)Селектор TSS прикладной задачи
pdescr  dq     0          ;(9)Псевдодескриптор
mesctrl db     ' Управляющая задача приняла управление ';(10)
mesctrl_len=$-mesctrl     ;(11)
tblhex  db     '0123456789ABCDEF';(12)
string  db '**** ****-**** ****-**** ****-**** ****';(13)
len=$-string              ;(14)
        ...               ;Завершающие строки сегмента
code_ctrl segment 'code' use16;(15)Сегмент команд управляющей
                               ;задачи
        assume CS:code_ctrl,DS:data_ctrl;(16)
home_sel dw    home       ;(17)Адрес возврата из исключения
        dw     16         ;(18)Селектор сегмента команд
;Обработчики исключений exc_0ah, exc_0bh, exc_0ch, exc_0dh,
;exc_0eh и dummy_exc
        ...
;Сервисная функция 1 управляющей задачи. Функция осуществляет
;вывод на экран строки текста. См. пример 72.1, предложения
;23...42
        ...
main    proc              ;(19)
;Вычислим и занесем в GDT линейные ;адреса сегментов управляющей
;задачи: данных data_ctrl, команд code_ctrl и стека stack_ctrl,
;сегментов состояния задач: управляющей tss_ctrl и прикладной
;tss_task, а также таблицы локальных дескрипторов gdt_ldt
        ...
;Вычислим и занесем в LDT линейные адреса сегментов прикладной
;задачи: данных data_task, команд code_task и стека ;stack_task
        ...
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        ...
;Инициализируем необходимые поля TSS управляющей задачи
        mov    tss_ctrl+60h,56        ;(20)Селектор LDT
;Инициализируем TSS прикладной задачи
        mov    tss_task+4Ch,15        ;(21)CS
        mov    tss_task+20h,offset task;(22)IP
        mov    tss_task+50h,23        ;(23)SS
        mov    tss_task+38h,256;(24)SP
        mov    tss_task+54h,07        ;(25)DS
```

```
        mov    tss_task+48h,35        ;(26)ES
        mov    tss_task+60h,56        ;(27)LDT
        mov    tss_task+4,256 ;(28)ESP0
        mov    tss_task+8,24  ;(29)SS0
;Подготовимся к переходу в защищенный режим
        ...
;Загрузим IDTR
        ...
;Переходим в защищенный режим
        ...
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
        ...
continue:                      ;(30)
;Делаем адресуемыми данные и стек, инициализируем регистр ES
        ...
;Выведем на экран сообщение управляющей задачи
        mov    CX,mesctrl_len;(31)
        mov    SI,offset mesctrl;(32)
        mov    AH,3Eh      ;(33)
        mov    DI,1920     ;(34)
scr1:   lodsb              ;(35)
        stosw              ;(36)
        loop   scr1        ;(37)
        delay  100         ;(38)Задержка для наглядности
;Загрузим TR селектором TSS управляющей задачи
        mov    AX,40       ;(39)Селектор TSS в GDT
        ltr    AX          ;(40)
;Выполним переключение на прикладную задачу через ее TSS
        call   dword ptr start_task;(41)
;После завершения прикладной задачи командой iret управление
вернулось сюда.
;Доберемся до адресного пространства прикладной задачи
        mov    AX,4        ;(42)Будем адресовать прикладную
        mov    FS,AX       ;(43)задачу через FS
        add    FS:pos,mes_len*2+2;(44)Сменим позицию на экране
        inc    FS:mes      ;(45)Заменим цифру 1 на 2
        mov    tss_task+20h,offset task;(46)Исправим поле EIP в
                           ;TSS прикладной задачи для ее
                           ;повторного вызова
;Выполним повторное переключение на прикладную задачу через ее TSS
        call   dword ptr start_task;(47)
;Продолжение управляющей задачи после возврата из прикладной
go:     mov    AX,0FFFFh  ;(48)Диагностическое значение
home:   mov    SI,offset string;(49)
        debug              ;(50)
;Выведем на экран диагностическую строку
        ...
;Вернемся в реальный режим
        ...
;Теперь процессор снова работает в реальном режиме
return:                    ;(51)
;Восстановим операционную среду реального режима (DS, SS:SP),
;разрешим аппаратные и немаскируемые прерывания, завершим
;программу. Закроем главную процедуру main и сегмент команд
;управляющей задачи
```

```
stk_ctrl segment stack 'stack'        ;(52)
        ...                   ;256 байт

;Сегменты прикладной задачи
data_task       segment       ;(53)
data_beg=$                    ;(54)
mes db '1 ',16,16,16,'Работает прикладная задача',17,17,17;(55)
mes_len=$-mes                 ;(56)
pos     dw      2240          ;(57)Позиция сообщения mes1
data_task_size=$-data_beg     ;(58)
data_task ends                ;(59)
code_task segment 'code' use16;(60)
        assume CS:code_task,DS:data_task;(61)
task    proc                  ;(62)
;Вызываем сервисную функцию управляющей задачи
;Подготовим параметры в стеке (5 двойных слов)
        mov     AH,24h        ;(63)В AH атрибут символов
        push    EAX           ;(64)Двойное слово в стек
        mov     AX,pos        ;(65)Позиция на экране
        push    EAX           ;(66)Двойное слово в стек
        mov     AX,mes_len    ;(67)Длина строки
        push    EAX           ;(68)Двойное слово в стек
        mov     AX,DS         ;(69)Сегментный адрес строки
        push    EAX           ;(70)Двойное слово в стек
        mov     AX,offset mes;(71)Относительный адрес строки
        push    EAX           ;(72)Двойное слово в стек
        delay   100           ;(73)Задержка для наглядности
        db      9Ah           ;(74)Код команды call far ptr
        dw      0             ;(75)Игнорируемое смещение
        dw      31            ;(76)Селектор шлюза вызова srv_1
        iret                  ;(77)
task    endp                  ;(78)
        ...                   ;Завершающие строки сегмента
stack_task segment 'stack';(79)
        ...                   ;Завершающие строки сегмента
```

Предлагаемый пример мало отличается от примера 72.1; ниже описываются только внесенные изменения.

Наличие двух задач требует описания в полях данных внутреннего кольца двух TSS (предложения 6-7), а в таблице глобальных дескрипторов - двух дескрипторов этих сегментов (предложения 2-3). Дескриптор LDT сдвинулся в GDT на одну позицию и его селектор стал равен 56.

Для упрощения из управляющей программы удалена функция srv_2, а из таблицы дескрипторов прерываний - шлюз этой процедуры.

Для обеспечения переключения на прикладную задачу в сегмент данных введена ячейка start_task, содержащая во втором слове селектор TSS прикладной задачи (48). Первое слово этой ячейки игнорируется.

Слегка изменен текст контрольного сообщения управляющей задачи (предложение 10) - слово "программа" заменено словом "задача".

На этапе инициализации дескрипторных таблиц необходимо заполнить оба дескриптора сегментов состояния - дескриптор gdt_ctrl для управляющей задачи и gdt_task для прикладной.

Существенным элементом подготовки задач является заполнение их сегментов состояния. Как и в предыдущих примерах, большую часть TSS управляющей задачи можно не инициализировать, так как процессор заполнит его текущим контекстом при переключении на прикладную задачу и воспользуется потом этим контекстом при возврате в управляющую. Однако селектор LDT, для которого в TSS предусмотрено слово по адресу 60h, процессору взять неоткуда. Если управляющая задача не обращается к локальному адресному пространству, это поле ей и не требуется. Однако в нашем примере управляющая задача обращается к сегменту данных кольца 3 (см. предложения 42..45), занося в сегментный регистр FS селектор 4, принадлежащий локальному адресному пространству. В то же время при возврате из прикладной задачи в управляющую мы потеряем адресуемость LDT, если из TSS загрузится исходное нулевое значение селектора. Поэтому в соответствующее поле TSS управляющей задачи необходимо заранее занести селектор LDT (предложение 20)

Что же касается TSS прикладной задачи, то в нем необходимо заполнить все поля, значение которых мы хотим передать в прикладную задачу при переключении на нее (см. предложения 21..29) Вообще говоря, строго необходима только инициализация поля кадра стека нулевого уровня (эти данные понадобятся процессору при выполнении команды iret обратного переключения с уровня 3 на уровень 0); однако удобно занести в TSS и другие элементы контекста, чтобы не заниматься инициализацией в прикладной задаче.

Если управляющая задача будет обращаться к локальным полям данных, необходимо загрузить в LDTR селектор таблицы локальных дескрипторов. У нас обращение к сегменту data_task осуществляется только после возврата в управляющую задачу из прикладной. В этом случае явное заполнение LDTR не требуется, так он будет загружен из соответствующего поля TSS в процессе переключения задач.

После перехода в защищенный режим выполняются обычные действия по инициализации сегментных регистров, выводу контрольного сообщения (предложения 31...37) и объявления текущего сегмента состояния задачи (предложения 39-40) После этого можно переключиться на прикладную задачу (предложение 41)

Прикладная задача, работая на уровне 3, выводит на экран (с помощью единственной в этом примере функции управляющей программы) контрольное сообщение и командой iret осуществляет возврат на уровень 0 в управляющую задачу. Для демонстрации доступности программам уровня 0 прикладных полей данных модифицируем в управляющей задаче две ячейки прикладной программы: позицию выводимой строки (ячейка pos) и один символ в ней (ячейка mes). Для этого следует настроить на поля данных прикладной задачи один из свободных

сегментных регистров. В примере для этого используется регистр FS, в который загружается селектор дескриптора сегмента data_task из LDT (предложения 42-43). Далее выполняется модификация ячеек pos и mes.

Перед повторным переключением на прикладную задачу необходимо занести в TSS прикладной задачи относительный адрес точки входа, затертый в процессе возврата в управляющую задачу с помощью команды iret.

В процедуре task в текущий стек (т.е. стек уровня 3) загружаются необходимые параметры и командой дальнего вызова через селектор шлюза вызова управление передается процедуре srv_1. Процесс передачи параметров через стек уровня 3 и получения их из стека уровня 0 подробно описывался в предыдущей статье. В нашем примере процедура task вызывается дважды, причем в промежутке между вызовами управляющая программа "залезает" в ее поля данных и изменяет содержимое ячеек pos и mes. В результате при правильной работе программы на экран выводится сообщение, привведенное на рис. 73.1.

```
FFFF

 Управляющая задача приняла управление

1 ►►►Работает прикладная задача◄◄◄ 2 ►►►Работает прикладная задача◄◄◄
```

Рис. 73.1. Вывод программы 73.1.

Статья 74

Режим виртуального МП 8086

Как должно быть ясно из предыдущих глав, реальный и защищенный режимы процессоров фирмы Intel абсолютно не совместимы друг с другом. В этих режимах действуют разные схемы преобразования сегментного адреса в физический, а также разные алгоритмы обслуживания аппаратных и программных прерываний. Операционная система MS-DOS, являясь системой реального режима, не может работать в защищенном режиме. Таким образом, для использования преимуществ защищенного режима надо писать специальные программы, для того же, чтобы на процессорах 80286 и выше выполнять программы реально-

го режима, процессор следует переключать в этот режим (или, точнее говоря, не переводить его в защищенный режим).

Начиная с МП 386, процессоры фирмы Intel обеспечивают еще одну возможность выполнения программ реального режима (программ МП 8086) - так называемый виртуальный режим МП 8086, или, как его часто сокращенно называют, V86. Режим V86 обеспечивает выполнение программ МП 8086 в условиях действия механизмов управления памятью, защиты и переключения задач, характерных для защищенного режима. Однако для того, чтобы в режиме V86 выполнять программы, написанные для МП 8086, в памяти должна находиться специальная управляющая программа - монитор виртуальной машины. Эта программа, во-первых, обеспечивает настройку системных областей и вызов программ МП 8086 с одновременным переключением режима, а во вторых, обслуживание программных и аппаратных прерываний так, что хотя прерывания обрабатываются по правилам защищенного режима, тем не менее для программ МП 8086 действуют обычные для них правила написания обработчиков прерываний. Монитор виртуальной машины должен находиться во внутреннем кольце защиты и работать, соответственно, на уровне привилегий 0, программы МП 8086 - в самом внешнем кольце и работать на уровне 3.

Аппаратные средства обслуживания задач, включаемые в процессоры фирмы Intel начиная с МП 386, позволяют организовать в режиме V86 одновременное выполнение многих задач МП 8086, причем каждая задача выполняется с полным доступом ко всем ресурсам МП 8086 (мегабайт адресного пространства, система прерываний), как если бы она была единственной задачей в системе. При этом режим виртуального МП 8086 не запрещает выполнение, наряду с задачами МП 8086, задач защищенного режима МП 386/486. Система защиты позволяет надежно изолировать все задачи друг от друга и от монитора виртуальной машины, который в таком многозадачном режиме должен содержать в себе средства переключения задач по заданному алгоритму.

Другой вариант выполнения на процессорах 80x86 задач МП 8086 - работа в реальном режиме. В этом случае механизмы защиты, страничной организации памяти и многозадачности не действуют, что делает невозможным параллельное выполнение процессором нескольких задач.

В настоящей статье будет рассмотрен простейший монитор виртуальной машины, который выполняет настройку системных таблиц и вызов задачи МП 8086 с переключением процессора в режим V86. В дальнейших статьях будут изложены вопросы обратного перехода в защищенный режим, обслуживания прерываний и эмуляции средств операционной системы реального режима.

Для того, чтобы упростить и сделать более наглядными тексты предлагаемых задач, в них внесены некоторые усовершенствования.

Так, фрагмент вычисления линейного адреса сегмента и заполнения соответствующих полей дескриптора выделены в макрокоманду load_addr, которая помещена в макробиблиотеку MAC.MAC. Текст этой макрокоманды, имеющей два параметра, приведен ниже.

```
;Вычисление 32-битового линейного адреса сегмента seg_addr и
;загрузка его в дескриптор descr в таблице дескрипторов
load_addr macro descr,seg_addr
        and     EAX,0FFFFh ;Обнуление старшей половины EAX
        mov     AX,seg_addr;Сегментный адрес -> в EAX
        shl     EAX,4      ;Преобразование в линейный адрес
        mov     descr.base_l,AX;Занесение в дескриптор младшего
                           ;слова линейного адреса
        rol     EAX,16     ;Обмен половин EAX
        mov     descr.base_m,AL;Занесение в дескриптор битов
                           ;16...23 линейного адреса
        endm
```

В предыдущих статьях для вывода на экран содержимого регистров процессора или ячеек памяти использовалась макрокоманда debug (см. статью 63), в которой имелось обращение к символьной таблице преобразования tblhex. Эту строку приходилось включать в сегмент данных программы. При исследовании режима виртуального МП 8086 такая методика неудобна, так как из задачи V86 нельзя обратиться к полям данных монитора; нельзя также продублировать строку с тем же символическим именем в сегменте данных задачи V86. Проблему можно решить, включив таблицу преобразования непосредственно в макрокоманду debug и изменив текст этой программы так, чтобы учесть перенос полей данных из сегмента данных в сегмент команд. Эта видоизмененная макрокоманда debug названа нами debugv:

```
debugv  macro
        local  j,tbl       ;Объявление локальных имен
;При вызове: AX=преобразуемое число, SI=адрес строки с результатом
;преобразования (одно 4-разрядное 16-ричное число)
        jmp    j           ;Обойдем поле данных
tbl     db '0123456789ABCDEF';Таблица преобразования
j:      push   AX          ;Сохраним регистры
        push   BX
        push   CX
        push   DX
        push   AX          ;Сохраним наше число в стеке
        and    AX,0F000h   ;Выделим старшую четверку битов
        shr    AX,12       ;Сдвиг право на 3 четверки битов
        push   DS          ;Сохраним DS
        push   CS          ;Загрузим CS
        pop    DS          ;в DS для команды xlat
        mov    BX,offset tbl;BX=адрес таблицы трансляции
        xlat               ;Команда табличной трансляции
        pop    DS          ;Восстановим DS
        mov    [SI],AL     ;Результат - в строку
        pop    AX          ;Вернем в AX исходное число
```

```
        push    AX              ;И отправим его обратно в стек
        and     AX,0F00h        ;Выделим вторую четверку битов
        shr     AX,8            ;Сдвиг вправо на 2 четверки битов
        inc     SI              ;Сдвинемся к следующему байту
        push    DS              ;Далее аналогично для остальных
        push    CS              ;четверок битов
        pop     DS
        mov     BX,offset tbl;BX=адрес таблицы трансляции
        xlat                    ;Команда табличной трансляции
        pop     DS
        mov     [SI],AL
        pop     AX
        push    AX
        and     AX,0F0h         ;Выделим третью четверку битов
        shr     AX,4            ;Сдвиг вправо на 1 четверку битов
        inc     SI              ;Сдвинемся к следующему байту
        push    DS
        push    CS
        pop     DS
        mov     BX,offset tbl   ;BX=адрес таблицы трансляции
        xlat                    ;Команда табличной трансляции
        pop     DS
        mov     [SI],AL
        pop     AX
        push    AX
        and     AX,0Fh          ;Выделим младшую четверку битов
        inc     SI              ;Сдвинемся к следующему байту
        push    DS
        push    CS
        pop     DS
        mov     BX,offset tbl   ;BX=адрес таблицы трансляции
        xlat                    ;Команда табличной трансляции
        pop     DS
        mov     [SI],AL
        pop     AX              ;Восстановим стек
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        endm
```

Команда jmp, с которой начинается текст макрокоманды, позволяет обойти поле данных (таблицу трансляции). Перед каждым выполнением команды xlat, которая требует в регистрах DS:BX адрес таблицы трансляции, выполняется сохранение содержимого DS и засылка в DS адреса текущего сегмента команд из CS.

В дополнение к макрокоманде debugv в файл DEBUG.MAC включена макрокоманда auxv, упрощающая процедуру обращения к макрокоманде debugv:

```
auxv    macro data,offs,string
        push    AX              ;Для большей универсальности
        push    SI              ;сохраним используемые регистры
        mov     AX,data         ;Источник выводимого числа
        mov     SI,offset string+offs;Адрес в строке
```

```
        debug
        pop     SI          ;Восстановим
        pop     AX          ;регистры
        endm
```

Как видно из текста этой макрокоманды, теперь для заполнения требуемого поля произвольной строки string содержимым того или иного регистра или ячейки памяти достаточно написать, например, (для строки string1):

```
        auxv    SS,10,string1
```

или

```
        auxv    GS:mem1,15,string1
```

Наконец, напишем макрокоманду lineout, с помощью которой можно будет выводить на экран контрольные и диагностические строки текста. Будем считать, что регистр ES настроен на начальный адрес видеобуфера (в защищенном или реальном режимах), а адрес выводимой строки, ее длину, местоположение на экране и атрибут выводимых символов будем задавать с помощью параметров макрокоманды. Предусмотрим в макрокоманде автоматический инкремент строки экрана, куда выводится данное сообщение. В дальнейшем, когда мы будем реализовывать многократные переходы в защищенный режим из виртуального, это средство даст возможность выводить на экран информацию из одной и той же точки программы многократно без затирания предыдущего вывода. Чтобы обезопасить себя от выхода за пределы экрана, если вызовов нашей макрокоманды окажется слишком много, предусмотрим после заполнения экрана переход на его первую строку.

```
lineout macro line,pos,attr,len
        local a,b
;Макрокоманда для вывода в видеобуфер строки текста. Формальные
;параметры: line - смещение выводимой строки (сегментный
;адрес предполагается в DS), pos - смещение слова в программе, в
;котором хранится позиция (в байтах) на экране при первом выводе
;(при последующих выводах сообщение будет сдвигаться вниз на одну
;строку), attr - атрибуты символов строки, len - длина выводимой
;строки (не более 80 символов)
        push    AX          ;Сохраним
        push    CX          ;в стеке
        push    SI          ;используемые
        push    DI          ;регистры
        mov     SI,offset line;DS:SI->выводимая строка
        mov     CX,len      ;Длина строки в байтах
        mov     AH,attr     ;Атрибут
        mov     DI,pos      ;ES:DI->позиция на экране
a:      lodsb               ;Прочитаем очередной символ строки
        stosw               ;Выведем его в видеобуфер
        loop    a           ;Повторим это len раз
        add     pos,160     ;Инкремент позиции строки
        cmp     pos,24*160  ;Дошли до предпоследней строки экрана?
        jb      b           ;Нет, продолжим инкремент строк
```

```
        mov     pos,0       ;Да, продолжим с первой строки
b:      pop     DI          ;Восстановим
        pop     SI          ;из стека
        pop     CX          ;используемые
        pop     AX          ;регистры
        endm
```

Предлагаемый ниже пример содержит две задачи: монитор виртуальной машины и прикладная задача МП 8086. Для общности обе эти задачи имеют отдельные сегменты команд, данных и стека, причем сегменты задачи МП 8086 оформлены так, что в физическом адресном пространстве они будут расположены вместе, за сегментами монитора (рис. 74.1). С этой целью всем сегментам задачи МП 8086 назначен один (произвольный) класс abc. Компоновщик, встретив в объектном модуле сегменты одного класса с различными именами, размещает их в памяти друг за другом. Такое расположение сегментов в памяти в нашем случае не имеет принципиального значения и предусмотрено только с целью упрощения анализа адресов в процессе отладки.

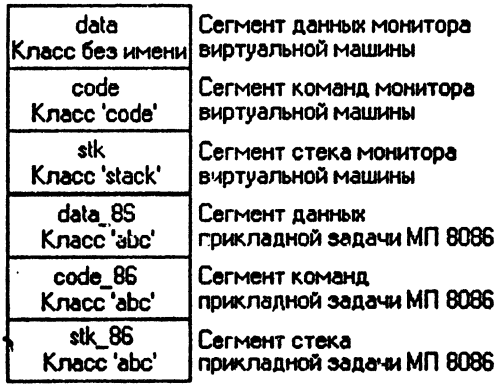


Рис. 74.1. Размещение в памяти сегментов примера 74.1.

Как и все предыдущие программы, пример 74.1 рассчитан на запуск в среде операционной системы MS-DOS. После загрузки управление получает главная процедура main монитора виртуальной машины, который мы для краткости будем в дальнейшем называть задачей 386. В этой процедуре осуществляется построение таблиц дескрипторов (глобальной и прерываний), подготовка возврата в реальный режим и переключение в защищенный режим. В защищенном режиме выполняется настройка полей данных для переключения в режим виртуального МП 8086 и переключение на задачу МП 8086 (или кратко задачу V86) с одновременной сменой режима работы процессора. Задача V86 выводит на экран контрольное и диагностическое сообщения и завершается сбросом процессора. Обратный переход в защищенный режим в данном примере не предусмотрен.

Пример 74.1 служит только для изучения процесса перехода в режим V86, и не имеет никакой практической ценности. Поскольку в нем пока не реализована эмуляция системы прерываний реального режима, задача V86 не может ни обрабатывать аппаратные прерывания, ни вызывать функции DOS или BIOS. В сущности, она не может делать ничего. Поэтому контрольное сообщение выводится на экран путем прямого обращения к видеобуферу.

Пример 74.1. Переход в режим виртуального МП 8086

```
include debug.mac
include mac.mac
.386P
;Структура для описания дескрипторов сегментов
        ...
;Структура для описания шлюзов прерываний
        ...
data    segment use16     ;(1)Сегмент данных задачи 386
;Таблица глобальных дескрипторов GDT
gdt_null descr <>         ;(2)Селектор 0
gdt_data descr <data_size-1,,,92h>;(3)Селектор 8, данные 386
gdt_code descr <code_size-1,,,9Ah>;(4)Селектор 16, команды 386 с
                          ;разрешением исполнения и чтения
gdt_stk descr <stk386-1,0,0,92h>;(5)Селектор 24, стек 386
gdt_screen descr <4095,8000h,0Bh,92h>;(6)Селектор 32, видеобуфер
gdt_tss_386 descr <103,0,0,89h>;(7)Селектор 40, TSS задачи 386
gdt_tss_86 descr <103,0,0,8Bh>;(8)Сел-р 48,TSS задачи V86, занятый
gdt_size=$-gdt_null       ;(9)
;Таблица дескрипторов всех 256 прерываний и исключений IDT
trap 6 dup (<exc_xx>)     ;(10)Дескр-ры общего обработчика
trap    <exc_06>          ;(11)Дескриптор обработчика 6
trap    <exc_xx>          ;(12)Дескриптор общего обработчика
trap    <exc_08>          ;(13)Дескриптор обработчика 8
trap    <exc_xx>          ;(14)Дескриптор общего обработчика
trap    <exc_10>          ;(15)Дескриптор обработчика 10
trap    <exc_11>          ;(16)Дескриптор обработчика 11
trap    <exc_12>          ;(17)Дескриптор обработчика 12
trap    <exc_13>          ;(18)Дескриптор обработчика 13
trap    232 dup (<exc_xx>);(19)Дескр-ры общего обработчика
pdescr  dq    0           ;(20)Псевдодескриптор
mes db  27,'[31;42m  Вернулись в реальный режим!
',27,'[0m$';(21)
string  db    '**** ****_**** ****_**** ****_****' ;(22)
string_len=$-string       ;(23)
string_pos dw  160*10     ;(24)
home_sel dw    home,16    ;(25)Адрес. возврата из исключения
tss_386 db     104 dup (0);(26)Сегмент состояния задачи 386
tss_86  db     104 dup (0);(27)Сегмент состояния задачи V86
        ..                ;Завершающие строки сегмента
;Сегмент команд задачи 386
text    segment 'code' use16;(28)
        assume CS:text,DS:data;(29)
exc_06  proc              ;(30)Начало процедуры обработчика
        mov    AX,8       ;(31)Инициализация регистра DS
        mov    DS,AX      ;(32)селектором данных задачи 386
```

```
        mov     AX,32       ;(33)Инициализация регистра ES
        mov     ES,AX       ;(34)селектором видеобуфера
        mov     AX,6        ;(35)Номер исключения для пересылки
        jmp     dword ptr home_sel;(36)в диагностическую строку
exc_06  endp                ;(37)Конец процедуры обработчика
;Процедуры обработчиков остальных прерываний exc_06, exc_08,
;exc_10, exc_11, exc_12, exc_13 и exc_xx отличаются только кодом
;номера прерывания, засылаемого в регистр AX для диагностики
        ...
;Главная процедура монитора виртуальной машины (задачи 386)
main    proc                ;(38)
;Определим линейные адреса сегментов и загрузим их в GDT
        xor     EAX,EAX     ;(39)Обнулим старшую половину EAX
        mov     AX,data     ;(40)Настроим регистр DS
        mov     DS,AX       ;(41)на сегмент данных задачи 386
        load_addr gdt_data,DS;(42)Линейный адрес сегмента данных
                            ;в дескриптор задачи 386
        load_addr gdt_code,CS;(43)То же для сегмента команд
        load_addr gdt_stk,SS;(44)То же для сегмента стека
        load_addr gdt_tss_386,DS;(45)В дескриптор TSS задачи 386
                            ;линейный адрес сегмента данных, в
                            ;котором расположен TSS
        add     gdt_tss_386.base_1,offset tss_386;(46)Прибавим
                            ;смещение TSS
        load_addr gdt_tss_86,DS;(47)То же для дескриптора TSS
        add     gdt_tss_86.base_1,offset tss_86;(48)задачи V86
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        ...
;Установим флаг NT, объявив тем самым нашу задачу вложенной
        pushf               ;(49)FLAGS в стек
        pop     AX          ;(50)Перенесем в AX
        or      AX,4000h    ;(51)Установим флаг NT
        push    AX          ;(52)Новое слово флагов в стек
        popf                ;(53)Из стека в регистр FLAGS
;Инициализируем сегмент состояния задачи tss_386
        mov     word ptr tss_386,48;(54)Связь на tss_86
;Инициализируем сегмент состояния задачи V86
        mov     word ptr tss_86+4Ch,code_86;(55)CS V86
        mov     word ptr tss_86+20h,offset v86;(56)IP точки входа
                            ;в задачу V86
        mov     word ptr tss_86+50h,stk_86;(57)SS V86
        mov     word ptr tss_86+38h,stk86;(58)SP V86
        mov     word ptr tss_86+54h,data_86;(59)DS V86
        mov     word ptr tss_86+48h,0B800h;(60)ES V86 (подготовим
                            ;заранее)
        mov     dword ptr tss_86+24h,23000h;(61)EFLAGS (VM=1, NT=0,
                            ;IOPL=3, IF=0)
        mov     word ptr tss_86+8,24;(62)SS уровня 0
        mov     word ptr tss_86+4,stk0;(63)SP уровня 0
        ...
;Загрузим IDTR
        ...
;Переходим в защищенный режим
        ...
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
```

```
continue:                ;(64)
;Делаем адресуемыми данные, стек и видеобуфер
        ...
;Загрузим регистр задачи TR селектором TSS задачи 386
        mov    AX,40     ;(65)Селектор tss_386
        ltr    AX        ;(66)
;Перейдем в режим виртуального МП 8086
        iret             ;(67)
;Завершим программу, как и раньше (весь последующий текст до метки
;return будет выполняться только при возникновении по ходу
;программы исключения)
go:     mov    AX,0FFFFh ;(68)Диагностическое значение
;Выведем на экран диагностическую строку
home:   auxv   AX,0,string;(69)
        lineout string,string_pos,74h,string_len;(70)
;Вернемся в реальный режим
        mov    AL,0FEh   ;(71)Переходим в реальный режим
        out    64h,AL    ;(72)засылкой кода FEh в порт 64h
        hlt              ;(73)Останов в ожидании сброса
;Теперь процессор снова работает в реальном режиме
return: mov    AX,data   ;(74)
        mov    DS,AX     ;(75)
        mov    AX,stk    ;(76)
        mov    SS,AX     ;(77)Инициализируем стек заново
        mov    SP,200h   ;(78)
;Разрешим аппаратные и немаскируемые прерывания, завершим
программу. Закроем главную процедуру main и сегмент команд text
        ...
;Сегмент стека задачи 386
stk386=400h              ;(79)Размер стека задачи 386
stk0=200h                ;(80)Размер стека уровня 0
stk segment stack 'stack';(81)
        db stk386 dup (^);(82)
stk     ends             ;(83)
;Сегмент данных задачи V86
data_86 segment 'abc' use16;(84)
mes_86 db 4,4,4,' Работаем в режиме виртуального 8086 ',4,4,4;(85)
mes_86_len=$-mes_86      ;(86)Длина сообщения
mes_86_pos dw 160*12     ;(87)Позиция сообщения на экране
string1 db     '**** ****_**** **** ****_**** ;(88)
string1_len=$-string1    ;(89)Длина сообщения
string1_pos dw 160*14    ;(90)Позиция сообщения на экране
data_86_size=$-mes_86    ;(91)Размер сегмента данных
data_86 ends             ;(92)
;Сегмент команд задачи V86
code_86 segment 'abc' use16;(93)
        assume CS:code_86;(94)
v86     proc             ;(95)
;Выведем на экран контрольную строку
lineout mes_86,mes_86_pos,1Bh,mes_86_len ;(96)
;Выведем на экран диагностическую строку
lineout string1,string1_pos,31h,string1_len ;(97)
;Сбросим процессор и перейдем в реальный режим
        ...
v86     endp             ;(98)
c_86_size=$-v86          ;(99)
code_86 ends             ;(100)
```

```
;Сегмент стека задачи v86
stk86=300h                  ;(101)Размер стека задачи V86
stk_86  segment 'abc'       ;(102)
        db      stk86 dup ('&');(103)
stk_86  ends                ;(104)
        end     main        ;(105)
```

Как и все предыдущие примеры, программа начинается с объявления текстов структур для описания дескрипторов сегментов памяти (descr) и дескрипторов шлюзов прерываний (trap). В структуре trap заранее заполнены поля селектора сегмента, в котором находятся обработчики прерываний (селектор 16), и атрибута дескриптора (8Eh, т.е. присутствующий, тип - шлюз прерывания).

В таблицу глобальных дескрипторов GDT включены 7 дескрипторов: обязательный нулевой gdt_null, дескрипторы gdt_data, gdt_code и gdt_stk для описания, соответственно, сегментов данных, команд и стека задачи 386, дескриптор gdt_screen видеобуфера, а также две дескриптора gdt_tss_386 и gdt_tss_86 для описания сегментов состояния наших двух задач - монитора виртуальной машины и прикладной задачи виртуального МП 8086. Для описания сегментов задачи V86 дескрипторы не предусмотрены, так как она будет выполняться в режиме виртуального МП 8086, в котором отсутствует понятие дескриптора.

В глобальных дескрипторах сегментов памяти указаны фактические границы соответствующих сегментов, а также их атрибуты: 92 для сегментов данных и стека и 9A для сегмента команд. Обычно сегмент команд имеет атрибут 92, где код 2 обозначает разрешение исполнения. Однако в новой макрокоманде debugv предусмотрено обращение к полю данных tbl, находящемуся в сегменте команд. Для того, чтобы в защищенном режиме разрешить не только исполнение, но и чтение полей сегмента, ему следует присвоить атрибут 9A (см. табл. 66.1). В дескрипторе видеобуфера, кроме того, явным образом определен линейный адрес (B8000h).

Сегменты состояния задач TSS имеют в данном примере минимальный обязательный размер 104 байт и, соответственно, границу 103. Атрибут дескриптора TSS задачи 386 равен 89h (присутствующий, тип 9 - свободный сегмент состояния задачи МП 386 и 486), а для дескриптора TSS задачи V86 указан атрибут 8Bh (присутствующий, тип B - занятый сегмент состояния задачи МП 386 и 486). Почему этот TSS еще до начала выполнения задачи объявлен занятым, будет ясно из дальнейшего рассмотрения.

Таблица дескрипторов прерываний содержит 256 шлюзов прерываний и исключений. Фактически в наших программах могут возникать только исключения 6 (недопустимый код операции), 8 (двойное нарушение), 10 (недопустимый TSS), 11 (отсутствие сегмента), 12 (ошибка обращения к стеку) и 13 (нарушение общей защиты). Поэтому в про-

грамме предусмотрены отдельные обработчики только для этих исключений; все остальные исключения, если даже они и возникнут, будут обслуживаться общим обработчиком exc_xx. В программе не предусмотрено содержательной обработки исключений, которые при правильной работе программы возникать не должны. Однако в процессе отладки программы исключения могут создаваться в силу каких-либо ошибок (неправильный адрес памяти, незаконная команда и др.). Кроме того, в следующей статье мы займемся исследованиями режима V86, в процессе которых нам потребуется "отлавливать" возникающие исключения и определять их номер. Обработчики исключений данного примера очень просты и предназначены лишь для вывода номера возникшего исключения.

В практически одинаковых процедурах обработчиков (см., например, обработчик исключения 6, предложения 30...44) выполняется инициализация сегментных регистров DS и ES соответствующими селекторами, в регистр AX заносится номер исключения, после чего командой дальнего перехода управление передается через поле home_sel в точку home в конце программы. Дальний переход здесь остался от предыдущих примеров; его можно было заменить на ближний, так как и обработчик исключений, и задача 386 входят в один сегмент команд. В точке home (предложение 69) вызываются макрокоманды auxv и lineout для вывода на экран диагностической строки string. Далее процессор переводится в реальный режим и вся программа завершается.

Инициализация сегментных регистров в обработчике исключений необходима из-за того, что передача на него управления может произойти в процессе выполнения задачи V86, которая написана по правилам программ реального режима. Когда процессор выполняет эту задачу, сегментные регистры содержат не селекторы, а сегментные адреса, что недопустимо для защищенного режима, в котором работает обработчик. К тому же в задаче V86 сегментный регистр DS настроен на сегмент данных этой задачи, а диагностическая строка string расположена в сегменте данных задачи 386.

В сегменте данных data зарезервировано место под два сегмента состояния задач TSS (предложение 27), которые будут заполнены необходимой информацией по ходу инициализации.

Главная процедура main задачи 386 начинается, как и в предыдущих примерах, с инициализации сегментного регистра DS (пока в реальном режиме). Стоит отметить, что регистры SS:SP инициализируются автоматически системой на этапе загрузки программы в результате того, что в определение сегмента стека (предложение 81) включен описатель stack (тип объединения). Естественно, что тип объединения должен отсутствовать в описании сегмента стека задачи V86, иначе при запуске программы регистры SS:SP будут инициализированы неправильно.

Здесь же обратим внимание на то, что размер сегмента стека для удобства последующих ссылок на него указан в символической форме (stk386).

Далее в глобальные дескрипторы сегментов задачи 386 заносятся отсутствующие пока в них линейные адреса соответствующих сегментов памяти. Для этого используется макрокоманда load_addr (предложения 42...44). Для настройки дескрипторов TSS в них сначала макрокомандой load_addr заносится линейный адрес сегмента данных data (предложения 45 и 47), а затем к этому адресу прибавляется смещение соответствующего TSS внутри сегмента данных (предложения 46 и 48).

В предложениях с 49 по 53 выполняется установка флага вложенной задачи NT и настройка обоих сегментов состояния задач, что необходимо для перехода в режим V86. Этот переход (который будет выполнен из защищенного режима, а пока у нас реальный режим) можно осуществить разными способами. Он может быть выполнен в рамках одной задачи, а может сопровождать переключение задач. В любом случае для перевода процессора в режим V86 надо установить флаг виртуальной машины VM в регистре флагов EFLAGS (бит 17, что соответствует числу 20000h). Однако этот флаг невозможно установить непосредственно. Команд работы с флагом VM не существует, а попытка установить флаг через стек, как это часто делается для других флагов, с помощью последовательности команд

```
db      66h         ;Префикс замены размера операнда
pushf               ;Отправим содержимое EFLAGS в стек
pop     EAX         ;Извлечем его оттуда в EAX
or      EAX,20000h  ;Попробуем установить флаг VM
push    EAX         ;Новое слово флагов в стек
db      66h         ;Префикс замены размера операнда
popf                ;Из стека в EFLAGS
```

не приведет к желаемому результату, так как при выгрузке нового содержимого EFLAGS из стека состояние флага VM не изменится.

Состояние флага VM можно изменить командой iret, если она выполняет выход из обработчика прерываний нулевого уровня привилегий, и при этом в регистре флагов сброшен бит вложенной задачи NT (иначе команда iret будет выполнять процедуру переключения задач), а в слове флагов, которое она извлекает из стека, установлен флаг VM. В этом случае команда iret вернет управление в прерванную ранее процедуру уже в режиме виртуального МП 8086.

Другой способ установки флага VM предполагает переключение задач. Если выполняется переключение на задачу, для которой определен TSS 386, в поле для слова флагов которого (смещение 24h, см. рис. 67.3) установлен бит VM, то в процессоре устанавливается режим виртуального МП 8086. При этом переключение задач можно выполнить команда-

ми jmp или call через шлюз задачи или через TSS (см. статью 67), а можно достичь того же с помощью команды iret, если должным образом настроить оба участвующие в процессе переключения TSS, а также регистр флагов процессора. Рассмотрим сначала второй способ.

Как известно, команда iret при сброшенном флаге вложенной задачи NT выполняет обычный возврат в прерванную процедуру, снимая со стека три (в 16-разрядном режиме) или 6 (в 32-разрядном) слов, содержащих вектор возврата: флаги, CS и IP (или EIP). Если же флаг NT установлен, т.е. текущая задача является вложенной, та же команда iret выполняется принципиально по-иному, осуществляя возврат в исходную задачу. Конечный пункт этой операции, т.е. задача, на которую надлежит выполнить переключение, устанавливается с помощью слова связи, находящегося в TSS текущей задачи со смещением 0 от его начала. В этом слове хранится селектор дескриптора, описывающего TSS исходной задачи, на которую и выполняется переключение (см. рис. 67.4). При обычном переключении задач в защищенном режиме флаг VM и в самом регистре EFLAGS, и во всех его копиях (в TSS или в стеках) сброшен, и переключение задач осуществляется без изменения режима работы процессора.

Переход в режим виртуального МП 8086 с помощью команды iret требует ряда подготовительных операций.

Для того, чтобы команда iret осуществляла переключение задач, а не возврат через стек в прерванную процедуру, следует установить флаг NT (предложения 49...53). Правда, как уже отмечалось, в исходном состоянии флаг NT установлен, так что эти действия можно считать излишними.

В поле связи текущего TSS заносится селектор TSS той задачи, на которую надо выполнить переключение (предложение 54). В этом случае команда iret, выполняя "возврат в исходную задачу", узнает из этого поля, на какую задачу надо переключиться (рис. 74.2). Остальные поля TSS заполнять нет необходимости, они будут заполнены текущим контекстом задачи 386 при переключении на задачу V86.

TSS задачи V86 следует подготовить более основательно. При переключении на эту задачу в регистры процессора будут загружены данные из соответствующих полей TSS. Поэтому целесообразно заполнить заранее поля всех регистров, начальное значение которых известно. Прежде всего это относится к полям регистров CS:EIP (их заполнение строго обязательно) и SS:ESP (предложения 55...58). В нашем случае разумно также подготовить регистры DS и ES (предложения 59-60). Обратите внимание на то, что в поля для сегментных регистров заносятся не селекторы, не имеющие смысла в режиме V86, а привычные нам сегментные адреса, например, начальный адрес видеобуфера B800h.

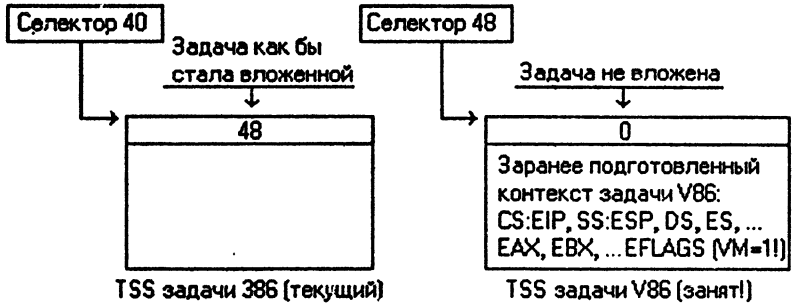


Рис. 74.2. Подготовка TSS задач перед переходом в режим V86.

Важнейшим элементом подготовки к переходу в режим МП 8086 является установка флага VM в образе EFLAGS в TSS (предложение 61). Если при переключении на задачу V86 флаг виртуальной машины окажется сброшен, процессор останется в защищенном режиме. Наконец, если предполагается переход из режима V86 назад в защищенный режим, необходимо заполнить поля TSS для кадра стека уровня 0 (предложения 62-63). При переходе в защищенный режим регистры SS:ESP будут загружены из этих ячеек. В настоящем примере такой переход не предусмотрен, однако он может произойти в результате определенных ошибок в программе, и отсутствие разумных значений в полях для SS:ESP уровня 0 приведет к дополнительным неприятностям. Для стека уровня 0 резервируется верхняя половина (с меньшими адресами) стека задачи 386, для чего этот стек сделан относительно большого размера (400h байт, предложение 79).

Далее по ходу программы выполняются уже знакомые нам действия по подготовке к переходу в защищенный режим (запрет прерываний, загрузка адреса возврата в ячейку BIOS и проч., загрузка регистра IDTR и, наконец, переключение режима процессора.

В защищенном режиме после инициализации сегментных регистров выполняется засылка селектора TSS задачи 386 в регистр задачи TR. Тем самым задача 386 объявляется текущей и возникает возможность ее переключения.

Предложение 57 с командой iret является ключевым. Именно эта команда осуществляет переключение на задачу V86 с одновременным переходом в режим виртуального МП 8086.

В "прикладной" задаче V86 с помощью прямого обращения к видеобуферу выполняется вывод контрольной строки mes_86, а за ней - пустой пока диагностической строки string1. Далее сбросом процессора осуществляется переход в реальный режим и завершение задачи. Вид экрана дисплея в случае правильной работы программы приведен на

рис. 74.3. Из рисунка видно, что запуск программного комплекса осуществлялся в среде оболочки DOS Norton Commander; программный комплекс имел имя p.exe и запускался командой p. Для того, чтобы диагностический и контрольный вывод программы появлялся на чистом экране, перед запуском программы была выполнена команда DOS CLS.

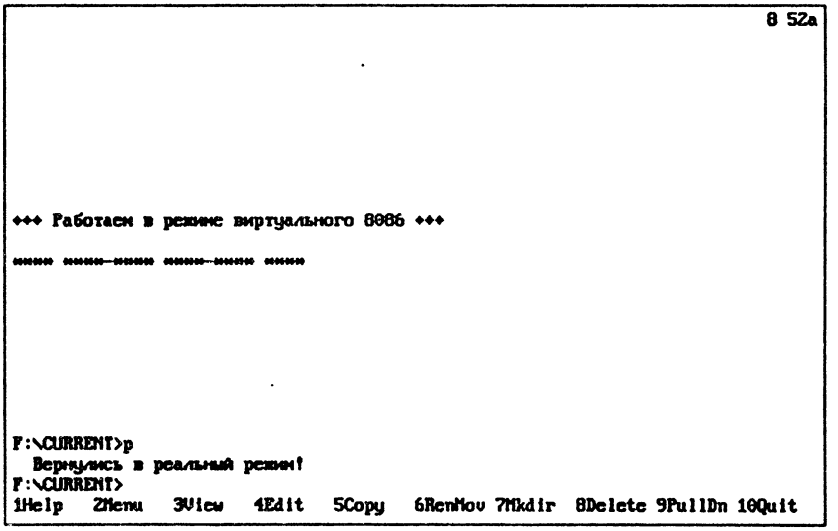


Рис. 74.3. Вывод примера 74.1.

Статья 75

Исследование режима виртуального МП 8086

Воспользуемся описанной в предыдущей статье программой в качестве основы для изучения режима виртуального МП 8086.

В рассмотренном примере мы перевели процессор в режим V86 с помощью команды iret, предварительно установив связь текущего TSS с TSS задачи V86, который был заранее объявлен занятым (иначе в контролируемую им задачу нельзя было бы "вернуться"). Рассмотрим те-

перь другой способ переключения в режим V86 - с помощью команды перехода jmp на TSS активизируемой задачи.

Включим в состав сегмента данных задачи 386 поле с адресом TSS задачи V86:

```
jmp_tss dw      0,48
```

Здесь 48 - селектор интересующего нас TSS, а второе слово адреса, предназначенное для смещения, в данном случае может иметь любое значение, так как не используется командой перехода.

Изменим описание дескриптора TSS задачи V86, объявив TSS свободным:

```
gdt_tss_86 descr <103,0,0,089h>;Селектор 48 TSS V86, свободный
```

Наконец, заменим команду iret перехода в режим V86 (предложение 67 примера 74.1) на команду

```
        jmp     dword ptr jmp_tss
```

Программа будет работать точно так же, как и раньше.

Для того, чтобы получить дополнительные подтверждения переключения в режим V86, попробуем определить, какой TSS является текущим. Вообще говоря, это нетрудно сделать с помощью команды str (store task register, запись регистра задач). Включим команду

```
        str     AX
```

в самый конец программы V86, перед переходом в реальный режим (после предложения 97). Вывод программы (рис. 75.1) говорит о том, что в процессе выполнения программы возникло исключение с номером 6.

Обратившись к табл. 64.1, видим, что это исключение недопустимого кода команды. Мы получили гораздо более веское подтверждение перехода в режим V86, чем предполагали! В самом деле, команда str, так же, как и другие команды того же типа (ltr, lgdt, lidt и др.), предназначенные для использования в операционных системах, а не в прикладных программах, могут работать только в защищенном режиме. В режимах реальных адресов эти команды не имеют смысла, интерпретируются процессором (не транслятором!), как недопустимый код и неминуемо приводят к исключению недопустимого кода команды.

Второй, не менее фундаментальный вывод заключается в том, что исключение, возникшее в режиме V86, переключает процессор в защищенный режим. Действительно, диагностическая строка с номером исключения получена с помощью нашего обработчика исключения exc_06, а он может работать только в защищенном режиме (достаточно взглянуть на строки загрузки сегментных регистров селекторами). Возбуждение аппаратного прерывания или исключения (в частности, с по-

мощью команды программного прерывания int) является единственной возможностью перевода процессора из виртуального режима в защищенный.

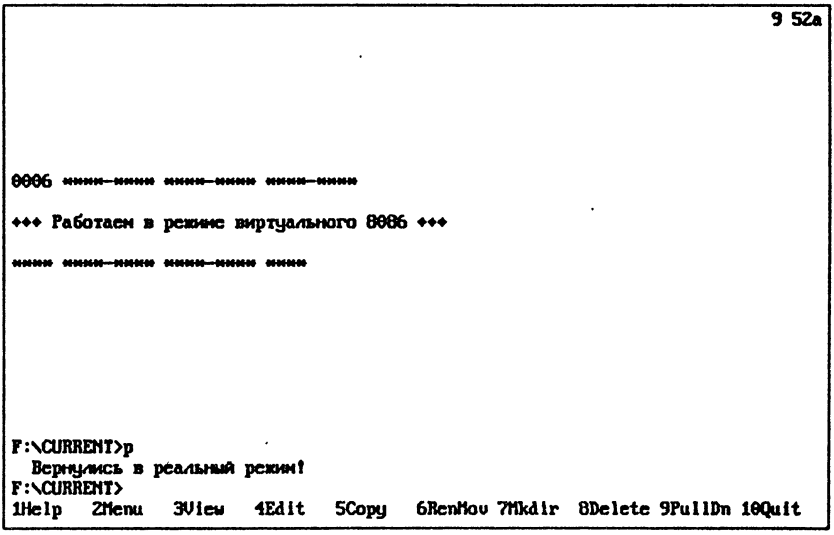


Рис. 75.1. В режиме V86 зафиксировано исключение с номером 6.

Теперь наша программа работает сначала в реальном режиме, затем переходит в защищенный, далее по команде jmp переключается в режим V86, оттуда, встретившись с "непосильной" командой str, возвращается в защищенный режим и, наконец, путем сброса процессора возвращается в исходный реальный режим, в котором и завершает свою работу. При этом мы имеем возможность выводить на экран диагностические строки как из защищенного режима, так и из режима V86. Воспользуемся этим обстоятельством для анализа содержимого сегментных регистров в обоих режимах.

Оставим в программе "неправильную" команду str AX и включим перед командой перехода в режим V86 (предложение 67) строки

```
auxv   DS,5,string
auxv   CS,10,string
auxv   SS,15,string
auxv   ES,20,string
```

а в самом конце, перед командой сброса процессора строки

```
auxv   DS,5,string1
auxv   CS,10,string1
```

```
auxv   SS,15,string1
auxv   ES,20,string1
```

Возможный вывод программы показан на рис. 75.2. Видно, что в защищенном режиме в сегментных регистрах содержатся селекторы сегментов (числа 8, 10h=16, 18h=24 и т.д.), а в режиме V86 - обычные сегментные адреса.

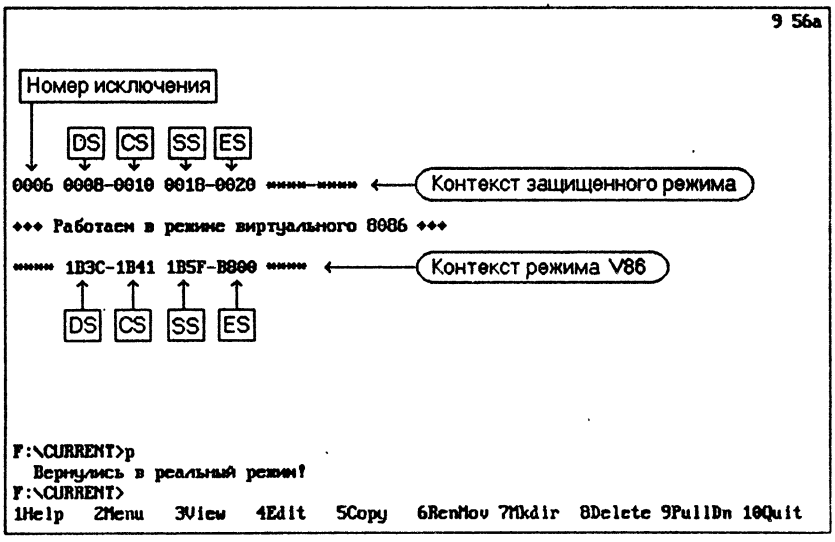


Рис. 75.2. Контексты виртуального и защищенного режимов.

Проверим теперь высказанное выше утверждение, что любое исключение, в том числе команда int, вызывает переход из режима V86 в защищенный. Удалим из задачи V86 команду str AX, и поставим на ее место команду int с каким-либо числовым аргументом, например,

```
int    00h
```

Запустив программу, убедимся в возникновении исключения с номером Dh=13 (нарушение общей защиты). При желании можно заменить аргумент команды int и повторить эксперимент. Всегда будет возникать исключение с номером 13. Получается так потому, что задача виртуального МП 8086 работает на третьем, наименее привилегированном уровне, и ей запрещено прямое обращение к уровню 0, на котором выполняется монитор виртуальной машины.

Между прочим, переход на выполнение обработчика прерываний, входящего в задачу 386, и работающего на уровне привилегий 0, в дан-

ном случае не означает смену текущей задачи. Определим, какая задача является текущей. Для этого достаточно в обработчик исключения 13 добавить строки (перед засылкой в AX кода номера исключения)

```
str    AX
auxv   AX,25,string .
```

Мы увидим, что в регистре задачи TR, содержимое которого как раз и определяет, какая задача считается текущей, по-прежнему содержится селектор TSS задачи V86 (число 30h=48). Получается, что команда int, вызвав исключение и перевод процессора в защищенный режим, не сменила текущую задачу.

Итак, прерывания и исключения, возбуждаемые в режиме виртуального МП 8086, обрабатываются в защищенном режиме. Можно ли, закончив эту обработку, вернуться на продолжение прерванной задачи реального режима? Чтобы научиться выполнять такой переход, надо рассмотреть детали перехода из виртуального режима в защищенный.

Как было показано выше, переход из режима V86 в защищенный происходит, когда процессор фиксирует прерывание или исключение, в частности, команду int с любым номером. Независимо от причины перехода, процессор извлекает из TSS текущей задачи реального режима кадр стека уровня 0 (ячейки TSS со смещениями 4 и 8) и сохраняет в этом стеке часть контекста задачи, именно, сегментные регистры GS, FS, DS, ES и SS, указатель стека ESP, регистр флагов EFLAGS, текущее содержимое CS, а также содержимое указателя команд EIP (рис. 75.3).

Если для данного исключения имеется код ошибки, он сохраняется в стеке на последнем месте. В этом случае в стек заносится в общей сложности 10 четырехбайтовых слов, т.е. 40 байт. Если код ошибки отсутствует (как это имеет место, например, для исключения 6 недопустимого кода операции), то в стеке сохраняется на 4 байта меньше.

Возврат из обработчика прерывания уровня 0 в прерванную программу уровня 3 осуществляется командой iret. Однако правильный переход в задачу виртуального МП 8086 произойдет лишь в том случае, если в слове флагов, хранящемся в стеке уровня 0, установлен флаг виртуального режима VM. Поэтому перед выполнением команды iret следует проанализировать содержимое ячейки EFLAGS с стеке. Если флаг VM сброшен, это означает, что вызов обработчика произошел в результате перехода не из режима V86, а из защищенного режима (что может иметь место в тех случаях, когда один и тот же обработчик предназначен для обслуживания исключений обоих режимов).

Второе условие правильного возврата в прерванную задачу заключается в выполнении команды iret с 32-разрядной адресацией:

```
db     66h
iret
```

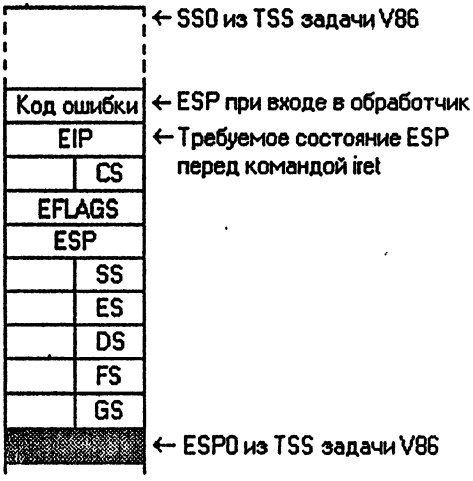


Рис. 75.3. Формат стека уровня 0 после перехода в защищенный режим.

В этом случае при загрузки из стека слова флагов будет восстановлено установленное состояние флага VM, что является обязательным условием режима V86. Обычная 16-разрядная команда iret восстановит только младшее слово флагов, и бит VM в регистре EFLAGS останется сброшенным.

Еще одно важное соображение касается слова с кодом ошибки. Перед выполнением команды возврата в режим V86 необходимо определить, имеется ли на стеке это слово. Если слово с кодом ошибки имеется, его необходимо снять со стека, чтобы указатель стека указывал на адрес возврата. Определить наличие слова с кодом ошибки можно по состоянию указателя стека. При наличии слова ошибки ESP=ESP0-40; при его отсутствии ESP=ESP0-36. Здесь ESP - текущее состояние указателя стека при входе в защищенный режим (в обработчик исключения), а ESP0 - исходное состояние указателя стека уровня 0, записанное заранее в TSS задачи виртуального МП 8086.

Полезную информацию можно получить, анализируя код ошибки. Формат этого слова для исключения 13 нарушения общей защиты был приведен на рис. 64.2. В нем бит 0 устанавливается, если исключение возникло в процессе обработки другого исключения или внешнего прерывания, бит 1 отводится под индикатор таблицы дескрипторов прерываний, а бит 2 является индикатором LDT. В битах 3...15 находится индекс дескриптора того сегмента, в котором возникло исключение. В нашем случае в коде ошибки будет установлен бит 1, что же касается индекса дескриптора, то при возбуждении исключения 13 по команде int в биты 3...15 помещается значение числового аргумента этой команды. Таким образом, хотя команда int с любым аргументом приводит к исключению 13, однако, анализируя код ошибки, можно определить конкретное значение ее аргумента.

Проанализируем содержимое стека уровня 0 в обработчике исключения 13. Для этого в обработчик exc_13 после инициализации регистров DS и ES включим строки

```
mov    EBP,ESP     ;Настроим BP на текущую вершину стека
mov    EAX,[BP]        ;Код ошибки
auxv   AX,5,string     ;Выведем в строку string
mov    EAX,[BP+4]      ;EIP
auxv   AX,10,string    ;Выведем в строку string
mov    EAX,[BP+8]      ;CS
auxv   AX,15,string    ;Выведем в строку string
mov    AX,[BP+14]      ;EFLAGS, старшая половина
auxv   AX,20,string    ;Выведем в строку string
mov    EAX,[BP+16]     ;ESP
auxv   AX,25,string    ;Выведем в строку string
mov    EAX,[BP+20]     ;SS
auxv   AX,30,string    ;Выведем в строку string
```

Возможный вывод программы приведен на рис. 75.4. Исключение в режиме V86 было создано с помощью команды int 10.

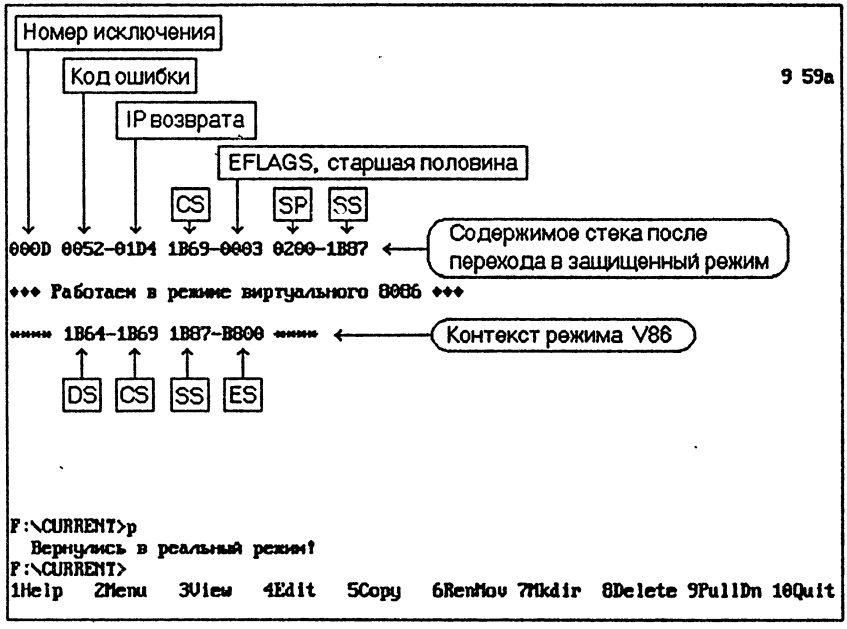


Рис. 75.4. Содержимое стека уровня 0 после перехода в защищенный режим и контекст режима V86

Проанализируем результаты нашего эксперимента. Код ошибки (верхнее слово стека) равен 52h. Это соответствует установленному биту 1 (ошибка при обращении к таблице прерываний) и числу 0Ah в поле индекса (аргумент команды int).

В следующем слове стека располагается относительный адрес возврата в прерванную программу (задачу V86). Здесь мы сталкиваемся с еще одним важным обстоятельством. Обычно в качестве адреса возврата из подпрограммы или обработчика прерываний выступает адрес очередной команды. Однако при возникновении нарушения в стеке сохраняется адрес той команды, которая вызвала данное исключение. Это в принципе дает возможность, устранив неполадку, выполнить команду повторно. В нашем случае команда int (которая сама относится не нарушениям, а к ловушкам) возбуждает нарушение общей защиты 13, и поэтому в стеке сохраняется адрес самой команды int. Посмотрите листинг трансляции вашей программы. Смещение 01D4h - это как раз адрес команды int 10 Таким образом, перед возвратом в режим V86 необходимо исправить содержимое стека, увеличив на 2 поле для EIP возврата (команда int nn занимает 2 байт).

В следующей ячейке стека хранится сегментный адрес реального режима задачи V86. Он, естественно, совпадает с соответствующим адресом в расположенной ниже строке контекста задачи V86.

В старшей половине слова флагов установлен бит 2, соответствующий флагу VM. Как уже отмечалось, перед возвратом из обработчика в виртуальный режим необходимо убедиться в том, что этот флаг установлен.

Содержимое указателя стека совпадает с его дном (200h). Это естественно, если просмотреть текст задачи V86. Перед переходом в защищенный режим она ничего не сохраняла в стеке.

Различия в сегментных адресах режима V86 на рис. 75.2 и 75.4 связаны с увеличением размеров сегмента команд задачи 386 и, как следствие, смещению задачи V86 в памяти в область больших адресов.

Теперь мы можем приступить к решению задачи возврата из защищенного режима в режим виртуального МП 8086. Модифицируем сначала только программу обработчика исключения 13, чтобы получить возможность после перехода в защищенный режим (с помощью команды int nn) вернуться назад и продолжить задачу V86. Новый текст процедуры exc_13 приведен ниже.

```
exc_13   proc
         cmp    SP,stk0-40 ;Есть ли на стеке код ошибки?
         jne    no_code    ;Нет, снимать не надо
         add    SP,4       ;"Снимем" со стека слово ошибки
no_code: push   EAX        ;Сохраним используемые
         push   EBP        ;в обработчике регистры (8 байт)
         mov    BP,SP      ;Установим базовый регистр
```

```
        add     BP,8        ;Скомпенсируем его на 2 push
        mov     EAX,[BP+8] ;Получим EFLAGS
        bt      EAX,17      ;Флаг VM установлен?
        jc      vm_ok       ;Да, можно продолжать
;Нет, что-то неверно, завершим программу, выведя сначала на экран
;диагностическую строку string. В этом случае сохраненные регистры
;(и стек) не восстанавливаем
        mov     AX,8        ;Инициализируем регистры
        mov     DS,AX       ;DS и ES для правильной
        mov     AX,32       ;дальнейшей работы
        mov     ES,AX       ;защищенного режима
        mov     AX,13       ;Номер исключения
        jmp     dword ptr home_sel;На выход
;Все верно, завершим обработчик,вернувшись в прерванную задачу V86
vm_ok:  add     word ptr [BP],2         ;Передвинемся за команду int
        pop     EBP         ;Восстановим сохраненные
        pop     EAX         ;ранее регистры
        db      66h         ;Замена размера операнда
        iret                ;Возврат в задачу V86
```

Помимо изменения процедуры обработчика исключения 13, надо еще предусмотреть контроль возврата в режим V86. Это можно сделать очень просто, включив команду int с любым аргументом после вывода на экран контрольного сообщения, но до вывода диагностической строки, т.е. перед предложением 97 примера 74.1. Если программа написана правильно, на экране должны появиться обе строки вывода из задачи V86 (рис. 75.5). Если же в программе что-то будет не так, то весьма вероятно, что на экран вместо контекста задачи V86 будет выведена строка string из задачи 386.

Итак, в режиме V86 команда int, независимо от значения аргумента, приводит к исключению с номером 13, которое, в свою очередь, переводит процессор в защищенный режим и вызывает соответствующий обработчик. Вообще, как было показано выше, возбуждение любого исключения (например, в результате попытки выполнить запрещенную команду) приводит к переключению в защищенный режим. Однако, кроме исключений, переходы из виртуального режима в защищенный инициируются также аппаратными прерываниями. В примере 74.1 аппаратные прерывания нам не досаждали по той причине, что в слове флагов в TSS задачи V86 был сброшен флаг IF, и задача V86 выполнялась при запрещенных прерываниях. В задаче же 386 аппаратные прерывания были запрещены командай cli. Разрешим аппаратные прерывания в задаче V86 и проведем несколько экспериментов по их изучению.

Обработка аппаратных прерываний потребует обращения к аппаратуре компьютера. Для того, чтобы в защищенном режиме получить доступ к пространству портов, необходимо снабдить TSS задачи V86 картой ввода-вывода и описать в этой карте, к каким именно портам разрешается доступ. Для простоты можно открыть доступ ко всем портам.

```
                                                                    10 10a
♦♦♦ Работаем в режиме виртуального 8086 ♦♦♦

•••••• 1B3F-1B44 1B62-B800 ••••••

F:\CURRENT>p
  Вернулись в реальный режим!
F:\CURRENT>
1Help  2Menu  3View  4Edit  5Copy  6RenMov 7Mkdir 8Delete 9PullDn 10Quit
```

Рис. 75.5. Вывод программы в случае успешного возврата из обработчика исключения в задачу V86.

Карта ввода-вывода располагается с адреса 68h от начала TSS и может содержать до 64 кбит, по одному биту на каждый порт. Бит 0 соответствует порту 0, бит 1 - порту 1 и т.д. Если бит сброшен, обращение к данному порту разрешено, если установлен - запрещено. Смещение начала карты ввода-вывода, т.е. число 68h, засылается в слово TSS по адресу 66h. Длина карты ввода-вывода не оговаривается, признаком ее завершения является слово, содержащее FFFFh.

Встретив команду ввода-вывода, процессор проверяет карту ввода-вывода, чтобы определить, разрешено ли программе обращение к указанному в команде порту. Если к порту обратиться нельзя, возбуждается исключение общей защиты. Таким образом, исключение 13 в виртуальном режиме может возникнуть не только при выполнения команды int, но и в результате попытки обращения к запрещенному для этой задачи порту (вспомним, что для каждой задачи предусматривается свой TSS и, следовательно, своя карта ввода-вывода).

Если команда ввода-вывода выполняется в защищенном режиме, то процессор перед обращением к карте ввода-вывода сравнивает текущий уровень привилегий CPL с уровнем привилегий ввода-вывода IOPL (биты 12-13 регистра флагов). Если CPL меньше или равен IOPL, операция ввода-вывода выполняется безотносительно к содержимому карты ввода-вывода. Проверка карты ввода-вывода осуществляется только в

том случае, если обнаруживается, что CPL>IOPL. Наши программы защищенного режима выполняются с CPL=0; значение же IOPL по умолчанию устанавливается равным 3. Таким образом, доступ ко всем портам оказывается разрешен, и в сегмент состояния задачи 386 нет необходимости включать карту ввода-вывода.

Если, однако, задача, содержащая команды ввода-вывода, выполняется в режиме виртуального МП 8086, то проверка карты ввода-вывода осуществляется независимо от уровня IOPL, и TSS такой задачи должен содержать карту ввода-вывода.

С учетом вышесказанного описание TSS задачи V86 будет теперь выглядеть следующим образом:

```
tss_86  dw      52 dup (0) ;Сегмент состояния задачи V86
iomap   db      8192 dup (0);Карта ввода-вывода
ioend   dw      0FFFFh      ;Завершающий код
tss_86_size=$-tss_86        ;Размер TSS
```

Изменится также поле границы в дескрипторе этого TSS:

```
gdt_tss_86 descr <tss_86_size-1,0,0,8Bh>;Селектор 48 TSS задачи
V86
```

Для того, чтобы задача V86 выполнялась при разрешенных прерываниях, необходимо установить флаг IF в поле флагов TSS этой задачи:

```
        mov     tss_86+24h,23200h;VM=1, NT=0, IOPL=3, IF=1
```

Аппаратное прерывание вызывает исключение с номером, равным номеру вектора, посылаемого из контроллера прерываний в процессор. Наиболее доступными для экспериментов являются прерывания от таймера и клавиатуры. Поскольку мы не перепрограммировали контроллер прерываний, эти устройства будут генерировать прерывания с векторами 8 и 9, соответственно. Поэтому в программу следует включить, кроме уже имеющихся в ней обработчиков исключений 6, 8, 10, 11, 12 и 13 еще и обработчик исключения 9.

Фиксация процессором прерывания некоторого уровня приводит к аппаратному блокированию в контроллере прерываний последующих прерываний этого и всех нижележащих уровней. Для снятия блокировки в обработчике аппаратного прерывания необходимо генерировать сигнал конца прерывания EOI, для чего служит последовательность команд (для ведущего контроллера, к которому подключены и таймер, и клавиатура)

```
        mov     AL,20h
        out     20h,AL
```

В результате текст обработчика исключений 8 будет выглядеть так:

```
exc_08  proc                ;Обработчик прерывания от таймера
        mov     AX,8 .
```

```
        mov     DS,AX
        mov     AX,32
        mov     ES,AX
        mov     AL,20h      ;Сигнал
        out     20h,AL      ;EOI
        mov     AX,8
        jmp     dword ptr home_sel
exc_08  endp
```

Сложнее дело обстоит с обработкой прерываний от клавиатуры. В этом обработчике, помимо разблокирования контроллера прерываний, необходимо еще получить из порта 60h скен-код нажатой клавиши, после чего разрешить контроллеру клавиатуры дальнейшую работу. Последнее выполняется кратковременной установкой бита 8 в порте 61h:

```
exc_09  proc                ;Обработчик прерываний от клавиатуры
        mov     AX,8
        mov     DS,AX
        mov     AX,32
        mov     ES,AX
        in      AL,60h      ;Снимем скен-код
        in      AL,61h      ;Прочитаем порт B
        or      AL,80h      ;Установим бит 8
        out     61h,AL      ;и в порт
        and     AL,7Fh      ;Сбросим бит 8
        out     61h,AL      ;и в порт
        mov     AL,20h      ;Сигнал
        out     20h,AL      ;EOI
        mov     AX,9
        jmp     dword ptr home_sel
exc_09  endp
```

Наша программа V86 очень коротка и выполняется за ничтожное время. Для того, чтобы успеть зафиксировать прерывание от таймера и, тем более, успеть нажать клавишу, в нее необходимо включить достаточно большую программную задержку (например, между выводом контрольной и диагностической строк). Для этого можно воспользоваться макрокомандой delay:

```
        delay 100
```

Включив в программу описанные изменения, запустим ее. Сразу же на экран выведется сообщение о регистрации исключения 8 (рис. 75.6).

```
8008 ---- ---- ---- ---- ---- ----

+++ Работаем в режиме виртуального 8086 +++
```

Рис. 75.6. Вывод программы с сообщением об исключении 8.

Для большей убедительности (в конце концов, мало ли откуда взялось исключение 8!) попробуем отработать прерывание от клавиатуры.

Однако для этого надо ухитриться нажать на клавишу до первого сигнала от таймера, т.е. в течение 1/18 с после запуска программы. Поскольку это практически невозможно, придется замаскировать в контроллере прерываний уровень 0 прерываний от таймера. Соответствующие программные строки можно включить как в процедуру main (до перехода в виртуальный режим), так и в самое начало процедуры v86:

```
in      AL,21h
or      AL,1
out     21h,AL
```

Однако после завершения задачи маска сама по себе не снимется и таймер останется заблокированным. Поэтому в задаче 386 перед переходом в реальный режим (после выхода из обработчика) следует предусмотреть снятие маски:

```
in      AL,21h
and     AL,0FEh
out     21h,AL
```

Запустим программу и после вывода на экран контрольного сообщения задачи V86 нажмем любую клавишу. На экране немедленно появится сообщение об исключении 9 (рис. 75.7). Таким образом, действительно, аппаратное прерывание вызывает исключение с номером, равным номеру его вектора.

```
0009 ----- ----- ----- ----- ----- -----

*** Работаем в режиме виртуального 8086 ***
```

Рис. 75.7. Вывод программы с сообщением об исключении 9.

Вернемся еще раз к вопросу о карте ввода-вывода. В защищенном режиме в наших условиях (CPL<IOPL) она не анализируется, и все операции ввода-вывода через порты разрешены. В режиме V86 в данном варианте программы мы установили карту ввода-вывода с разрешением обращения ко всем портам, что и дало нам возможность пользоваться командами in и out. Однако в примере 74.1 в задаче V86 тоже использовалась команда out (с ее помощью выполнятся переход в реальный режим), хотя карты ввода-вывода в ней не было. Правильная работа программы в этом случае объясняется тем, что хотя мы и не разрешили явным образом обращение к портам, но и не запретили его. Для того, чтобы оповестить процессор об отсутствии карты ввода-вывода, следует в поле для ее адреса (ячейка 66h в TSS) поместить адрес, больший или равный границе TSS, например

```
mov     tss_86+66h,tss_86_size
```

В этом случае все команды ввода-вывода, встретившиеся в режиме V86, будут возбуждать исключение общей защиты.

Статья 76

Эмуляция MS-DOS в режиме виртуальногоМП 8086

Важнейшей чертой программ реального режима является широкое использование средств DOS и BIOS. Практически невозможно представить себе программу, которая не обращалась бы, по ходу своей работы, к тем или иным функциям операционной системы. Поэтому режим виртуального МП 8086 может иметь право на существование лишь в том случае, если в нем правильно выполняются вызовы DOS.

Обращения к функциям DOS и BIOS осуществляются с помощью команд программных прерываний int. Между тем, в режиме V86 все команды int приводят к возникновению исключения общей защиты и передачи управления в обработчик этого исключения, который работает в защищенном режиме. Следовательно, важнейшей функцией обработчика исключения 13 должно быть получение числового аргумента вызвавшей его команды int (номера вектора программного прерывания) и передача управления через этот вектор требуемой программе операционной системы, которая, очевидно, должна выполняться опять в режиме V86. Всю эту процедуру иногда называют "отражением" прерывания на MS-DOS (или другую операционную систему реального режима), или "эмуляцией" DOS в режиме виртуального МП 8086.

Однако переключение из задачи V86 в защищенный режим может быть вызвано не только программным, но и аппаратным прерыванием. В этом случае возбуждается исключение с номером, равным номеру вектора аппаратного прерывания. Обработчики этих исключений (с 8h по Fh и с 70h по 77h) должны, определив номер вектора аппаратного прерывания, передать управление требуемой программе (обычно - обработчику BIOS), обеспечив тем самым эмуляцию обработки не только программных, но и аппаратных прерываний.

Наконец, программы BIOS, вызываемые (непосредственно или косвенно, посредством вызовов DOS) из прикладной программы, обращаются к аппаратуре компьютера и управляют ею через порты с помощью команд ввода-вывода in и out. Монитор виртуальной машины

должен обеспечить условия для успешного выполнения команд обращения к портам.

Разработаем программу монитора виртуальной машины, включающего в себя средства отражения аппаратных и программных прерываний на соответствующие программные средства DOS и BIOS.

В качестве основы воспользуемся примером 74.1, внеся в него необходимые дополнения, в частности, увеличив число глобальных дескрипторов. Структура программного комплекса примера 76.1 приведена на рис. 76.1.

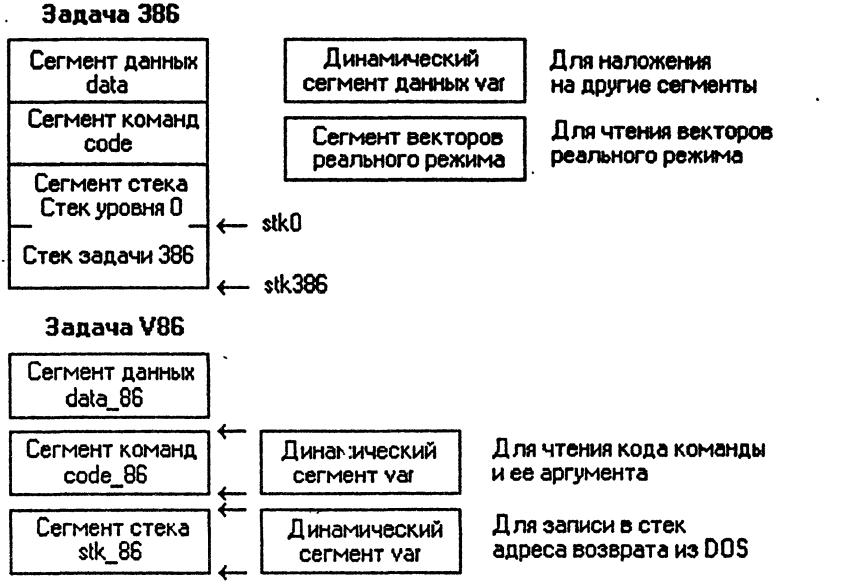


Рис. 76.1. Структура программного комплекса примера 76.1.

Программный комплекс, как и в предыдущем примере, включает две задачи - монитор виртуальной машины (задача 386) и прикладную программу (задача V86). Каждая из задач состоит из сегментов данных, команд и стека. Для задачи 386 предусмотрен стек относительно большого объема (stk386=400h байт), в котором верхняя половина (от смещения stk0=200h) отдана стеку уровня 0, используемому при переключении в защищенный режим из виртуального. Кроме этих 6 сегментов, в таблице глобальных дескрипторов предусмотрен дескриптор, описывающий область векторов реального режима. Через этот дескриптор программы защищенного режима, конкретно, обработчики ис-

ключений, получают доступ к векторам реального режима и, следовательно, к системным программам DOS и BIOS.

Монитор виртуальной машины должен иметь также доступ к запускаемой из него прикладной программе. Во-первых, при возбуждении исключения монитор обращается к сегменту команд задачи V86 с тем, чтобы выяснить, является источником исключения команда int, и если да, каков ее числовой аргумент. Далее, в процессе эмуляции DOS монитор помещает в стек задачи V86 адрес возврата для правильной отработки команды iret, завершающей вызываемое системное средство.

Сегменты прикладной программы, выполняясь в режиме V86, не имеют дескрипторов. Для получения доступа к этим сегментам в мониторе предусмотрен динамический сегмент данных, который будет накладываться на требуемый участок памяти, занимаемый прикладной программой. Как показано на рис. 76.1, наложение динамического сегмента на сегмент команд задачи V86 дает возможность прочитать код команды, вызвавшей исключение, наложение же динамического сегмента на прикладной стек позволяет записать в стек требуемую информацию.

Рассмотрим сначала в самых общих чертах взаимодействие монитора и прикладной программы в процессе обращения к средствам DOS (рис. 76.2).

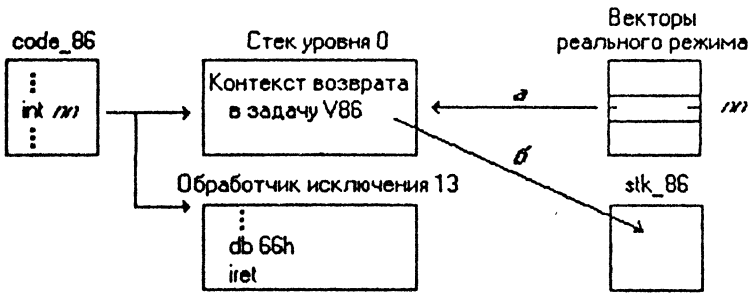


Рис. 76.2. Взаимодействие элементов комплекса при обработке программного прерывания.

Как было показано в предыдущей статье, команда int с любым аргументом, встретившаяся в прикладной программе, осуществляет переключение в защищенный режим, передавая управление через таблицу дескрипторов прерываний обработчику исключения 13. При этом, в отличие от процедуры реального режима, когда в стеке сохраняется вектор возврата (флаги, CS и IP), при переключении в защищенный режим из виртуального стек прикладной задачи совершенно не затрагивается.

Вместо этого в стеке того уровня, на который передается управление, в данном случае в стеке уровня 0, сохраняется контекст прерываемой задачи. В контекст задачи входит содержимое всех сегментных регистров, кадр прикладного стека, вектор возврата и код ошибки (см. рис. 75.3).

Обработчик исключения 13 должен переключить процессор назад в режим V86, передав управление программе, адрес которой хранится в векторе реального режима с номером, равным аргументу команды int. Это переключение осуществляется с помощью "32-разрядной" команды iret, которая использует контекст, находящийся в стеке уровня 0, восстанавливая из него сегментные регистры, кадр стека и CS:IP.

Таким образом, перед выполнением команды iret обработчик исключения 13 должен заменить в контексте возврата содержимое ячеек для CS и IP, взяв их новое содержимое из вектора реального режима (операция а на рис. 76.2).

Программа DOS (или BIOS), на которую передается управление, завершается командой iret. Эта команда, выполняясь в режиме V86, должна снять со стека прерванной программы три слова, заполнив из них регистр флагов, CS и IP. Однако в стеке прерванной программы (задачи V86) ничего нет, поскольку команда int, осуществляя переключение в защищенный режим, сохранила вектор возврата не на "своем" стеке, а в стеке нулевого уровня. Следовательно, обработчик исключения 13 должен создать в стеке прикладной программы структуру, "законную" для команды iret реального режима, т.е. занести в него вектор возврата в задачу V86, взяв требуемую информацию из стека нулевого уровня (операция б на рис. 76.2).

Легко видеть, что аппаратное прерывание, возникшее в процессе выполнения задачи V86, также должно обрабатываться в соответствии с рис. 76.2. Таким образом, монитор виртуальной машины вместо 256 обработчиков исключений может иметь только один, хотя, как будет видно из дальнейшего обсуждения, некоторые детали обработки аппаратных и программных прерываний различаются, что должно найти отражение в программе обработчика.

Реально, однако, оказывается удобным предусмотреть в мониторе 256 обработчиков, по одному для каждого возможного исключения. Все обработчики состоят фактически из одной команды ближнего вызова call handler, где handler - адрес единого "основного" обработчика исключений и прерываний:

```
call   handler    ;Обработчик исключения 0
call   handler    ;Обработчик исключения 1
call   handler    ;Обработчик исключения 2
...
```

Команда call помещает в стек адрес возврата, который в данном случае будет зависеть от того, какая именно из 256 команд call выпол-

нила вызов основного обработчика. Анализируя адрес возврата в стеке монитора, легко определить номер обрабатываемого исключения.

Рассмотрим, не вдаваясь пока в подробности, структурную схему единого обработчика исключений (рис. 76.3).

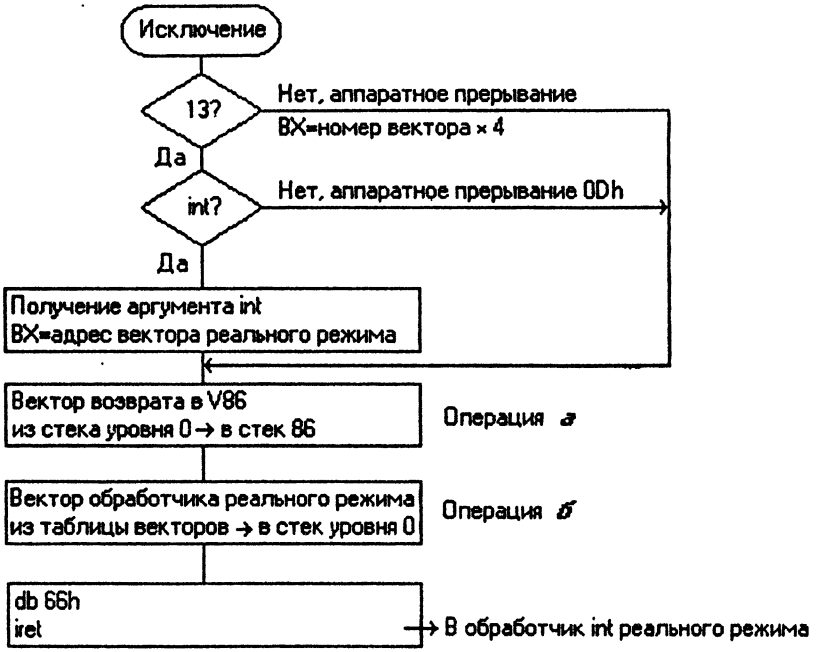


Рис. 76.3. Структурная схема обработчика исключений.

Как уже отмечалось, все программные прерывания возбуждают исключение 13, аппаратные же приводят к возникновению исключений с номерами, равными номерам векторов аппаратных прерываний. Поэтому прежде всего наш единый обработчик должен определить номер возникшего исключения и сравнить его с числом 13. Если номер исключения отличается от 13, значит, произошло аппаратное прерывание и, поскольку номер вектора уже известен, можно перейти к его обработке. Если же возникло исключение 13, надо еще определить источник прерывания: команда int с неизвестным пока аргументом или аппаратное прерывание с вектором 13. Аппаратное прерывание должно обрабатываться, как и все остальные; в случае же программного прерывания надо прочитать из сегмента команд прерванной программы значение числового аргумента команды int. Далее обработка и прерываний, и ис-

ключений осуществляется одинаково и состоит из двух операций, помеченных на рис. 76.2 буквами а и б.

Во-первых, из стека уровня 0 извлекаются "возвратные" значения IP, CS и регистра флагов и переносятся в стек задачи V86. При этом необходимо сместить на 6 байт указатель этого стека в контексте, хранящемся в стеке уровня 0, чтобы программа DOS, которой будет передан кадр стека прикладной программы, получила новое, исправленное значение указателя прикладного стека.

Во-вторых, из таблицы векторов реального режима (первый килобайт линейных адресов) извлекается вектор обрабатываемого прерывания и заносится в стек уровня 0 на место флагов, CS и IP, перенесенных в стек задачи V86.

Наконец, выполняется команда iret, которая, получив адрес возврата из стека уровня 0, передает управление обработчику данного прерывания реального режима. Возврат из этого (системного) обработчика осуществится через стек задачи V86, который мы уже подготовили для правильного выполнения этой операции.

Пример 76.1. Эмуляция DOS

```
include debug.mac
include mac.mac
.386P
;Структуры для описания дескрипторов сегментов и шлюзов прерываний
        ...
;Структура для описания формата стека уровня 0
oldstk  struc
oldip   dw      0,0             ;EIP возврата
oldcs   dw      0,0             ;CS возврата
oldfl   dw      0,0             ;Регистр флагов EFLAGS
oldsp   dw      0,0             ;ESP возврата
oldss   dw      0,0             ;SS возврата
oldes   dw      0,0             ;Сегментный регистр ES
oldds   dw      0,0             ;Сегментный регистр DS
oldfs   dw      0,0             ;Сегментный регистр FS
oldgs   dw      0,0             ;Сегментный регистр GS
oldstk  ends
data    segment use16
;Таблица глобальных дескрипторов GDT
gdt_null descr <>               ;Селектор 0
gdt_data descr <data_size-1,0,0,92h>;Селектор 8, данные 386
gdt_code descr <code_size-1,0,0,9Ah>;Селектор 16, команды 386
gdt_stk descr  <stk386-1,0,0,92h>;Селектор 24, стек 386
gdt_screen descr <4095,8000h,0Bh,92h>;Селектор 32, видеобуфер
gdt_tss_386 descr <103,0,0,89h>;Селектор 40, TSS задачи 386
gdt_tss_86 descr <tss_86_size-1,0,0,8Bh>;Сел-р 48,TSS задачи V86
gdt_00  descr  <03FFh,0,0,92h>;Селектор 56 сегмента векторов
                              ;прерываний реального режима
gdt_var descr  <0FFFFh,0,0,92h>;Селектор 64 динамического сегмента
trap    256 dup (<>)            ;256 дескрипторов прерываний
pdescr  dq      0               ;Псевдодескриптор
string  db '**** ****-**** ****-**** ****-**** 0'
```

```
string_len=$-string
string_pos dw  160*3       ;Позиция строки на экране
home_sel dw    home,16     ;Адрес аварийного возврата из исключения
tss_386  dw    52 dup (0) ;Сегмент состояния задачи 386
tss_86   dw    52 dup (0) ;Сегмент состояния задачи V86
iomap    db    8192 dup (0);Карта ввода-вывода
ioend    dw    0FFFFh     ;Завершающий код карты в-в
tss_86_size=$-tss_86      ;Размер tss-86, включая карту в-в
addr_ret dw    0          ;Ячейка для определения номера
                          ;исключения по адресу возврата
data     ends
text     segment 'code' use16
         assume CS:text,DS:data
begin    label word
;256 обработчиков по числу исключений
exc_00:  rept   256        ;Директива повторной трансляции
         call   handler    ;256 команд call handler (по 3б)
         db 0              ;Пустой байт для выравнивания
         endm              ;Конец повторяемого блока
;Единый обработчик всех исключений
handler  proc
         push   AX         ;Регистры общего назначения надо
                           ;сохранять, сегментные не надо
         mov    AX,8
         mov    DS,AX      ;Настроим DS и FS на
         mov    FS,AX      ;сегмент данных задачи 386
         mov    AX,32      ;Настроим ES на видеобуфер для
         mov    ES,AX      ;вывода диагностических сообщений
         pop    AX         ;Восстановим регистр
         inc    string+35  ;Диагностический счетчик вызовов
                           ;обработчика
         pop    addr_ret   ;Снимем со стека адрес байта
                           ;выравнивания
         cmp    SP,stk0-40 ;Есть ли на стеке код ошибки?
         jne    no_code    ;Нет, и снимать со стека нечего
         add    SP,4       ;Да, пропустим его в стека
no_code: pushad            ;Сохраним регистры
         mov    BP,SP      ;Работа со стеком уровня 0 через BP
         add    BP,32      ;Скомпенсируем на pushad
         mov    EAX,dword ptr [BP].oldfl;Получим EFLAGS
         bt     EAX,17     ;Флаг VM установлен?
         jc     vm_ok      ;Да, произошло переключение из V86
         mov    AX,7777h   ;Нет, это авария, занесем в AX
         jmp    dword ptr home_sel;условный код и на выход
vm_ok:   mov    AX,addr_ret;Заберем адрес байта выравнивания
                           ;сработавшего обработчика
         sub    AX,offset exc_00;Его смещение от точки exc_00
         and    AX,0FFFCh  ;AX=номер возникшего исключения * 4
         mov    BX,AX      ;Сохраним в BX - так удобнее
         cmp    BX,13*4    ;Было исключение 13?
         je     exc13      ;Да, на обработку всех исключений 13
         jmp    comm_1     /Все аппаратные прерывания(кроме 13)
exc13:   load_addr gdt_var,[BP].oldcs;Накладываем динамический
                           ;сегмент var на коды задачи V86
         mov    AX,64      ;Получим адресуемость сегмента
         mov    DS,AX      ;команд задачи V86 через DS (для
                           ;lodsb)
```

```
        mov    SI,[BP].oldip;SI=IP V86
        cld                  ;Чтение вперед
        lodsb                ;Получим код команды, вызвавшей
                             ;исключение
        cmp    AL,0CDh       ;Это команда int?
        je     is_int        ;Да, на обработку
        jmp    comm_1        ;Нет, аппаратное прерывание 0Dh
is_int: lodsb                ;AL=аргумент int, т.е. номер вектора
        mov    [BP].oldip,SI;В стек смещение следующей за int
                             ;команды
        sub    AH,AH         ;Очистим AH (AL=аргумент int)
        shl    AX,2          ;Получим адрес вектора
        mov    BX,AX         ;BX=адрес вектора
comm_1: mov    SI,[BP].oldsp;Получим в SI SP возврата задачи V86
        sub    SI,6          ;Зарезервируем в стеке 6 байт под
                             ;вектор возврата
        mov    [BP].oldsp,SI;Сдвинем на эту величину SP возврат
                             ;в стеке уровня 0
        load_addr FS:gdt_var,[BP].oldss;Накладываем динамический
                             ;сегмент var на стек задачи V86
        mov    AX,64         ;Получим адресуемость сегмента стека
        mov    DS,AX         ;V86 через DS (для записи в стек)
        mov    CX,[BP].oldip;IP возврата в задачу V86
        mov    DX,[BP].oldcs;CS возврата в задачу V86
        mov    DI,[BP].oldfl;Флаги возврата в задаче V86
        mov    [SI],CX       ;Занесем в стек задачи V86
        mov    [SI+2],DX     ;вектор возврата в нее же
        mov    [SI+4],DI     ;(IP, CS, FLAGS)
        and    DI,0FDFFh     ;Сбросим IF, имитируя действие int
        mov    AX,56         ;Селектор сегмента данных
        mov    DS,AX         ;векторов реального режима
        mov    AX,[BX]       ;Получим смещение из вектора
        mov    CX,[BX+2]     ;Получим сегмент из вектора
        mov    [BP].oldfl,DI;Занесем вектор обработчика DOS
        mov    [BP].oldip,AX;(из векторов реального режима)
        mov    [BP].oldcs,CX;в стек уровня 0
        popad                ;Восстановим все регистры
        db     66h           ;Префикс замены размера операнда
        iret                 ;"Возврат" в DOS
handler endp
;Главная процедура монитора виртуальной машины (задачи 386)
main    proc
;Определим линейные адреса сегментов и занесем их в GDT
        ...
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        ...
;Установим флаг NT, объявив нашу задачу вложенной
        ...
;Инициализируем сегмент состояния задачи tss_386
        mov    tss_386,48 ;Связь на tss_86
;Инициализируем сегмент состояния задачи V86
        mov    tss_86+4,stk0;Кадр стека
        mov    tss_86+8,24;уровня 0
        mov    tss_86+20h,offset v86;IP
        mov    dword ptr tss_86+24h,23200h;VM=1,NT=0,IOPL=3,IF=1
        mov    tss_86+38h,stk86;ESP
        mov    tss_86+48h,0B800h;ES
```

```
        mov     tss_86+4Ch,code_86;CS V86
        mov     tss_86+50h,stk_86;SS
        mov     tss_86+54h,data_86;DS
        mov     tss_86+66h,104;Адрес карты ввода-вывода
;Подготовимся к переходу в защищенный режим
        ...
;Заполним 256 дескрипторов исключений адресами обработчиков
        mov     CX,256      ;Счетчик шагов цикла
        mov     AX,offset exc_00;Начало массива обработчиков
        mov     BX,0        ;Смещение в массиве обработчиков
fill:   mov     idt.offs_l[BX],AX;Адрес текущего обработчика
        add     AX,4        ;Размер каждого обработчика 4 байт
        add     BX,8        ;Размер каждого дескриптора 8 байт
        loop    fill        ;Повторить 256 раз
;Загрузим IDTR
        ...
;Переходим в защищенный режим установкой флага PE
        ...
;Теперь процессор работает в защищенном режиме
;Загружаем в CS:IP селектор:смещение точки continue
        ...
continue:
;Делаем адресуемыми данные и стек, инициализируем регистр ES
        ...
;Загрузим регистр задачи TR селектором TSS задачи 386
        mov     AX,40       ;Селектор tss_386
        ltr     AX
;Перейдем в режим виртуального 8086
;       jmp     dword ptr jmp_tss
        iret                ;Переход через занятый TSS V86
;Завершим программу, как и раньше
go:     mov     AX,0FFFFh   ;Диагностическое значение
;Выведем на экран диагностическую строку
home:   auxv    AX,0,string
        lineout string,string_pos,74h,string_len
;Вернемся в реальный режим
        mov     AL,0FEh     ;Переходим в реальный режим
        out     64h,AL      ;засылкой кода FEh в порт 64h
        hlt                 ;Останов в ожидании сброса
;Теперь процессор снова работает в реальном режиме
return:
        mov     SS,AX       ;Инициализируем стек заново
        mov     SP,stk386
;Разрешим аппаратные и немаскируемые прерывания, завершим
;программу. Закроем процедуру main, закроем сегмент команд text
        ...
stk386=400h
stk0=200h
stk     segment stack 'stack'
        db      stk386 dup ('^')
stk     ends
data_86 segment 'abc' use16
string1 db      '**** ****_**** ****_**** **** '
string1_len=$-string1       ;Длина диагностической строки
string1_pos dw 160*22       ;Позиция строки на экране
data_86_size=$-mes_86
data_86 ends
```

```
code_86 segment 'abc' use16
        assume CS:code_86
v86     proc
;Заполним и выведем на экран диагностическую строку
        auxv   DS,0,string1    ;DS
        auxv   CS,5,string1    ;CS
        auxv   SS,10,string1   ;SS
        lineout string1,string1_pos,31h,string1_len
;Выведем на экран символ средствами BIOS
        mov    AH,0Eh      ;Функция вывода символа
        mov    AL,0Fh      ;Код символа
        int    10h         ;Прерывание BIOS
;Сбросим процессор и перейдем в реальный режим
        ...
v86     endp
code_86 ends
stk86=200h
stk_86  segment 'abc'
        db     stk86 dup ('&')
stk_86  ends
        end main
```

Остановимся на существенных участках программы.

В описательной части программы появилась структура oldsp, которая соответствует содержимому стека уровня 0 после перехода в него из задачи виртуального МП 8086 (см. рис. 75.3). Структура начинается с поля oldip для содержимого указателя команд и предназначена для ссылок в такой ситуации, когда базовый регистр настроен именно на это поле, т.е. со стека уже снят код ошибки (если он там был). Использование обозначений структуры oldip повышает наглядность многочисленных предложений программы, в которых выполняется обращение к стеку уровня 0.

В дескрипторе gdt_tss_86, описывающем сегмент состояния задачи V86, в размер сегмента включена карта ввода-вывода, а сам сегмент объявлен занятым. Это предопределяет способ перехода в режим V86 - с помощью команды iret.

Дескриптор gdt_00, описывающий поле векторов реального режима, полностью определен и не требует заполнения поля смещения линейным адресом, поскольку линейный базовый адрес таблицы векторов равен 0.

Дескриптор динамического сегмента данных gdt_var, который мы будем накладывать на различные участки задачи V86, имеет естественную для сегментов реального режима границу 0FFFFh и значение атрибута 92h (запись и чтение).

В сегменте данных задачи 386 следует обратить внимание на описание карты ввода-вывода iomap, вплотную прилегающей к TSS задачи V86, и на завершающий код 0FFFFh.

Сегмент команд начинается с обработчиков исключений. Для того, чтобы выровнять границы обработчиков на 4 байт, что облегчит определение их адресов, в каждый обработчик включен выравнивающий байт с произвольным кодом (0 в программе). Как было описаны выше, возбуждение любого исключения или прерывания приводит к передаче управления на общий обработчик handler, причем после выполнения 3-байтовой команды call в стеке (уровня 0) остается адрес следующего за командой call байта, в нашем случае - выравнивающего байта, т.е. для исключения 0 адрес exc_00+3, для исключения 1 - exc_00+7 и т.д. По адресу возврата в стеке легко определить номер исключения, вычтя начальное смещение exc_00, очистив два младших бита адреса и разделив полученное число на 4. Если же деление на 4 опустить, результирующее число является смещением вектора данного прерывания

Общий обработчик исключений handler работает по алгоритму, изображенному на рис. 76.3. Прежде всего сохраняется в стеке содержимое регистра AX, после чего настраиваются сегментные регистры DS и ES, первый - на сегмент данных задачи 386, второй - на видеобуфер. Вспомним, что при переключении в защищенный режим в стеке сохраняется содержимое всех сегментных регистров, но не регистров общего назначения. Поэтому перед использованием в обработчике регистров общего назначения их исходное содержимое необходимо сохранить в стеке. При этом смещается вершина стека, что может отразиться на последующем ходе программы.

В нашем обработчике после инициализации сегментных регистров и восстановления регистра AX (и исходной вершины стека) из стека извлекается и сохраняется в двухсловной ячейке addr_ret адрес байта выравнивания, соответствующего конкретному исключению. С помощью команды cmp выполняется анализ наличия в стеке слова ошибки. Как отмечалось выше, если это слово отсутствует, указатель стека смещен относительно его начального значения, указанного в TSS задачи V86, на 36 байт, если присутствует - на 40 байт. Само слово ошибки в обработчике не используется, поэтому "удаление" из стека слова ошибки осуществляется путем смещения указателя стека на 4 байт в сторону больших адресов. В результате в точке no_code состояние стека соответствует структуре oldstk.

Командой pushad все регистры сохраняются в стеке, что смещает его указатель на 32 байт. По ходу программы обработчика нам придется многократно обращаться к контексту задачи V86, сохраняемому в стеке уровня 0. Для адресации к стеку естественно использовать специально предназначенный для работы со стеком базовый регистр BP, который настраивается на то смещение в стеке, которое было вершиной в точке no_code до сохранения регистров.

Следующее действие - проверка, установлен ли в слове флагов, хранящемся в стеке, флаг виртуальной машины VM. В нашем случае сброшенное состояние этого флага говорит о серьезной ошибке, в принципе же анализ флага VM позволяет определить, откуда произошел переход в обработчик - из режима V86 или из защищенного режима (в котором флаг VM всегда сброшен).

Четыре предложения, начинающиеся с метки vm_ok, служат для идентификации исключения, конкретно, определения его номера, умноженного на 4, т.е. адреса в таблице векторов. Полученный адрес для удобства дальнейшего использования сохраняется в базовом регистре BX.

Теперь можно определить природу исключения. Если исключение имеет номер 13, оно, скорее всего, возникло в результате выполнения в режиме V86 команды int. Если исключение имеет иной номер - это аппаратное прерывание.

Обработка исключений с номером 13 выполняется, начиная с метки esc13. С помощью макрокоманды load_addr поле смещения дескриптора gdt_var заполняется линейным адресом сегмента команд задачи V86, а селектор этого сегмента (число 64) засылается в сегментный регистр DS. Для определения линейного адреса сегмента команд необходимо значение CS из задачи V86; оно хранится в стеке в составе контекста задачи V86 (в ячейке со смещением oldcs). Далее из стека извлекается и заносится в регистр SI значение указателя команд IP на момент выхода из режима V86. Если переход в защищенный режим обязан команде int, IP указывает на эту команду. Если было аппаратное прерывание, IP указывает на ту (произвольную) команду, перед которой процессор зафиксировал прерывание. Командой lodsb из сегмента команд задачи V86 считывается код команды и сравнивается с числом CDh - кодом операции int. Если коды не совпадают, можно сделать вывод, что инициатором переключения явилось аппаратное прерывание с номером 0Dh, которое следует обработать так же, как и остальные аппаратные прерывания. Если же код команды равен CDh, можно продолжить обработку исключения от команды int.

Вообще говоря, сказанное не вполне верно. Во-первых, аппаратное прерывание 0Dh могло случайно поступить точно перед командой int в программе. Поэтому наличие кода int еще не говорит однозначно о программном прерывании. Более строгий анализ источника исключения можно провести, анализируя регистр обслуживаемых запросов контроллера прерываний. Впрочем, прерывание 0Dh обычно закрепляется за вторым параллельным портом и, как правило, не используется.

Во-вторых, переключение в защищенный режим из виртуального при определенных условиях вызывают, кроме команды int, также команды, с помощью которых изменяется (или может изменяться) со-

стояние флага IF. Это команды cli, sti, pushf, popf и iret. Все они чувствительны к уровню привилегий ввода-вывода IOPL в регистре флагов. Если в текущей задаче V86 уровень IOPL<3, то эти команды не выполняются, а вызывают исключение общей защиты. Установив для задачи режима V86 значение IOPL (в слове флагов TSS задачи) меньше 3, монитор получает возможность "отлавливать" перечисленные команды и регулировать доступ виртуальных задач к реальному регистру флагов. В нашем примере IOPL=3 и описанная ситуация возникнуть не может.

Следующий шаг - считывание из сегмента команд задачи V86 аргумента команды int, который, естественно, расположен в следующем после кода операции (CDh) байте. Одновременно регистр SI настраивается на адрес следующей за int nn команды. Этот адрес заносится в соответствующее поле стека уровня 0 и будет в дальнейшем играть роль адреса возврата в задачу V86.

Аргумент команды int, т.е. номер вектора реального режима, умножается на 4 (сдвигом влево на 2 бит) и сохраняется в BX.

Начиная с метки commn_1 идет часть программы, общая и для исключения 13, и для прерываний. В любом случае при подходе к этой точке в BX находится адрес требуемого вектора реального режима.

Теперь нам надо поместить правильные адреса возврата в стек уровня 0 (чтобы команда iret нашего обработчика передала управление не в задачу V86, а в DOS или BIOS через вектор реального режима) и в стек задачи V86 (чтобы завершающая команда iret обработчика DOS или BIOS вернула управление в задачу V86). Содержимое SP задачи V86, хранящееся в контексте стека уровня 0, уменьшается на 6 (для помещения в стек флагов, CS и IP) и записывается назад в стек уровня 0.

Далее на стек задачи V86 накладывается динамический сегмент и значения FLAGS, CS и IP из контекста задачи V86 в стеке уровня 0 переносятся в стек V86. Обратите внимание на то, что сейчас регистр DS указывает на сегмент стека V86, а сегмент данных задачи 386, в котором находятся дескрипторы, адресуем только через регистр FS, который мы заранее инициализировали, имея в виду эту ситуацию.

Вектор реального режима извлекается из таблицы векторов и записывается в стек уровня 0, как вектор возврата из текущего обработчика. При этом в слове флагов сбрасывается флаг IF, чем имитируется обычное действие команды программного прерывания int, которая сбрасывает этот флаг.

После восстановления всех регистров командой popad выполняется завершающая команда iret с префиксом замены размера операнда (чтобы перенести в регистр флагов текущее значение флага VM) и управление передается в обработчик, адрес которого находился в векторе реального режима.

Что касается основной процедуры main монитора виртуальной машины, то она мало изменилась по сравнению с примером 74.1.

При инициализации TSS задачи V86 в слове флагов устанавливается бит IF. Это естественно, так как наша прикладная программа должна допускать обработку аппаратных прерываний. Далее в слово TSS со смещением 66h помещается относительный (от начала TSS) адрес карты ввода-вывода, равный у нас 104.

После подготовки к переходу в защищенный режим в дескрипторы IDT записываются смещения обработчиков исключений. Поскольку размеры дескрипторов и обработчиков известны, смещения вычисляются программным образом в цикле.

Для того, чтобы максимально использовать поле экрана, из процедуры main удалены строки вывода на экран сообщения "Вернулись в реальный режим!".

В задаче V86 с помощью предусмотренных для этого макрокоманд формируется диагностическая строка со значениями DS, CS и SS. Вывод этой строки на экран поможет нам в дальнейших экспериментах. Далее выполняется, можно сказать, коронный номер - задача виртуального МП 8086 обращается к операционной системе, в данном случае к функции 0Eh прерывания BIOS 10h обслуживания экрана. Функция BIOS выводит на экран в позицию курсора большую звездочку (код 0Fh) и смещает курсор на одну позицию (рис. 76.4).

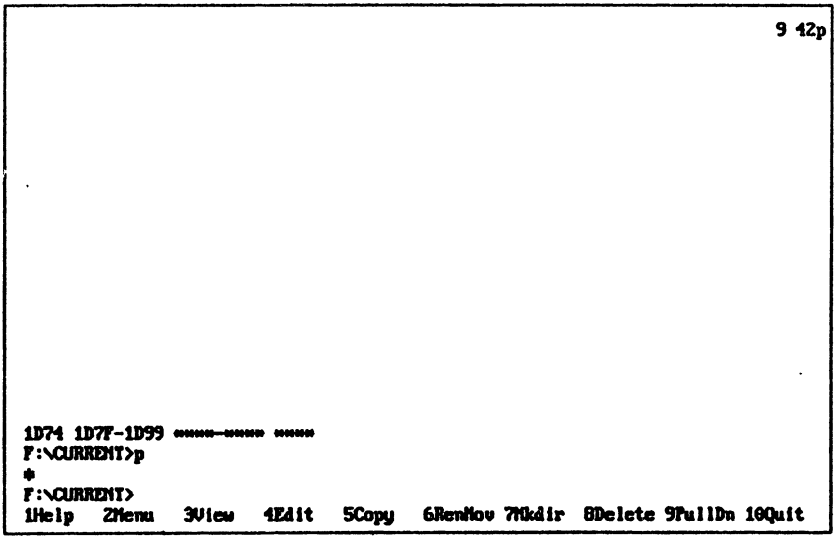


Рис. 76.4. Вывод первого варианта программы 76.1.

Проведем серию экспериментов по изучению режима V86. Прежде всего удостоверимся, что наш обработчик правильно эмулирует DOS и позволяет обращаться к ее функциям. Добавим в программу V86 строки создания на диске файла и записи в него некоторой информации, для чего в сегмент данных добавим описание имени файла и выводимый в файл текст, а в сегмент команд - последовательный вызов функций прерывания DOS 21h с номерами 3Ch, 40h и 3Eh для создания файла, записи в него и закрытия.

```
data_86 segment
        ...
fname   db 'newfile.000',0;Спецификация файла
record  db              'Строка для записи в файл'
record_len=$-record
handle  dw    0         ;Ячейка для дескриптора файла
        ...
code_86 segment
        ...
;Создадим новый файл
        mov   AH,3Ch    ;Функция создания файла
        mov   CX,0      ;Без атрибутов
        mov   DX,offset fname;Адрес имени файла
        int   21h       ;Переход в DOS
        mov   handle,AX ;Сохраним файловый дескриптор
;Запишем в файл некоторую информацию
        mov   AH,40h    ;Функция вывода
        mov   BX,handle ;Дескриптор файла
        mov   CX,mes40_len;Число выводимых байтов
        mov   DX,offset record;Адрес выводимой строки
        int   21h       ;Переход в DOS
;Закроем файл
        mov   AH,3Eh    ;Функция закрытия файла
        mov   BX,handle ;Дескриптор файла
        int   21h       ;Переход в DOS
```

Теперь включим в состав обработчика исключений фрагмент формирования и вывода на экран диагностической строки. Это позволит динамически, по ходу выполнения задачи V86, фиксировать вызовы обработчика исключений и получать информацию о них. Включать отладочные строки в текст обработчика следует с большой осторожностью, чтобы не разрушить регистры или поля данных обработчика. Выведем на экран информацию о номере исключения, коде вызвавшей его команды, втором коде команды (т.е. в случае команды int - номере вектора), адресе программы, на котором это исключение возникло.

Диагностические строки включим в обработчик после проверки флага VM:

```
vm_ok:  mov   AX,addr_ret;Заберем адрес байта выравнивания
                          ;сработавшего обработчика
        sub   AX,offset exc_00;Его смещение от точки exc_00
        and   AX,0FFFCh ;AX=номер исключения * 4
```

```
        mov    BX,AX       ;Сохраним в BX - так удобнее
;Сюда включаем диагностический фрагмент
        push   BX          ;Сохраним BX
        shr    AX,2        ;Получим номер исключения
        auxv   AX,0,string;Номер исключения в строку string
        load_addr gdt_var,[BP].oldcs;Наложим динамический сегмент
        mov    AX,64       ;var на сегмент команд V86
        mov    DS,AX       ;
        mov    SI,[BP].oldip  ;IP V86
        mov    CX,[BP].oldcs  ;CS V86
        mov    AX,0        ;Подготовим регистр AX
        cld                ;Пересылка вперед
        lodsb              ;Получим в AL код команды
        mov    BX,AX       ;Сохраним его в BX
        lodsb              ;Получим в AL второй байт команды
        mov    DX,AX       ;Сохраним его в DX
        push   ES          ;Восстановим адресуемость
        pop    DS          ;сегмента данных задачи 386
        auxv   BX,5,string     ;Код команды в строку string
        auxv   DX,10,string    ;Номер вектора в строку string
        auxv   CX,15,string    ;Сегмент команды в строку string
        mov    SI,[BP].oldip;Восстановим в SI смещение команды
        auxv   SI,20,string;Смещение команды в строку string
        lineout string,string_pos,75h,string_len;Вывод на экран
        pop    BX          ;Восстановим сохраненный регистр
;Продолжение программы обработчика
        cmp    BX,13*4     ;Было исключение 13?
        je     exc13       ;Да, на обработку всех исключений 13
        jmp    comm_1      ;Все аппаратные прерывания(кроме 13)
exc13:  load_addr gdt_var,[BP].oldcs;Накладываем сегмент var на
                           ;коды V86
        ...
```

Перед началом исследований вернем задачу V86 к ее первоначальному виду - с одним только вызовом функции 0Eh прерывания BIOS 10h. Вывод программы показан на рис. 76.5.

Обработчик исключений вывел одну строку, откуда следует, что он был активизирован один раз. Исключение с номером Dh было вызвано командой с кодом CDh и аргументом 10h, т.е. командой int 10h. Эта команда находится в сегменте с адресом 1DA5h, и из диагностической строки задачи V86 видно, что это как раз адрес ее сегмента команд. Таким образом, данное исключение вызвано командой int нашей "прикладной" программы. Судя по диагностической строке, команда int 10h имеет смещение 164h, что легко проверить по листингу трансляции.

Включим в задачу V86 после вызова функции BIOS небольшую программную задержку:

```
delay 15
```

Вывод программы показан на рис. 76.6.

Введение в программу задержки привело к тому, что обработчик исключений был активизирован на один, а семь раз.

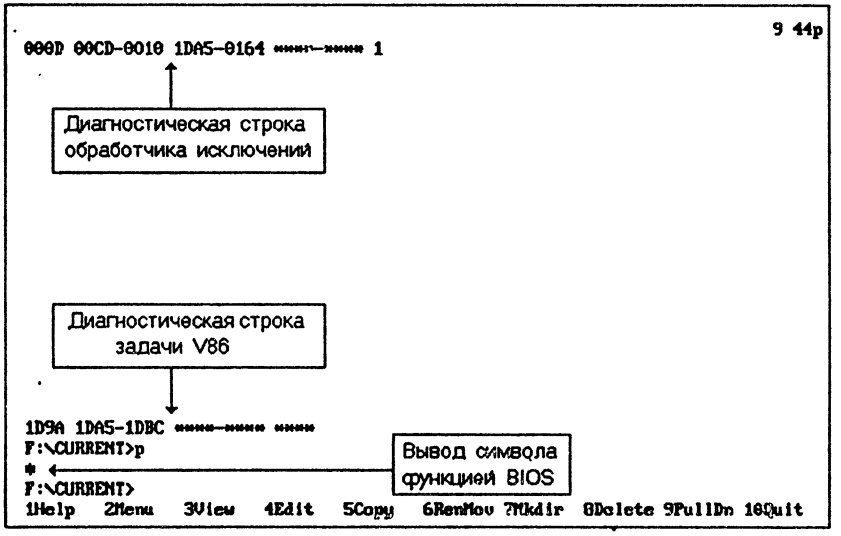


Рис. 76.5. Диагностический вывод программы при отработке прерывания BIOS.

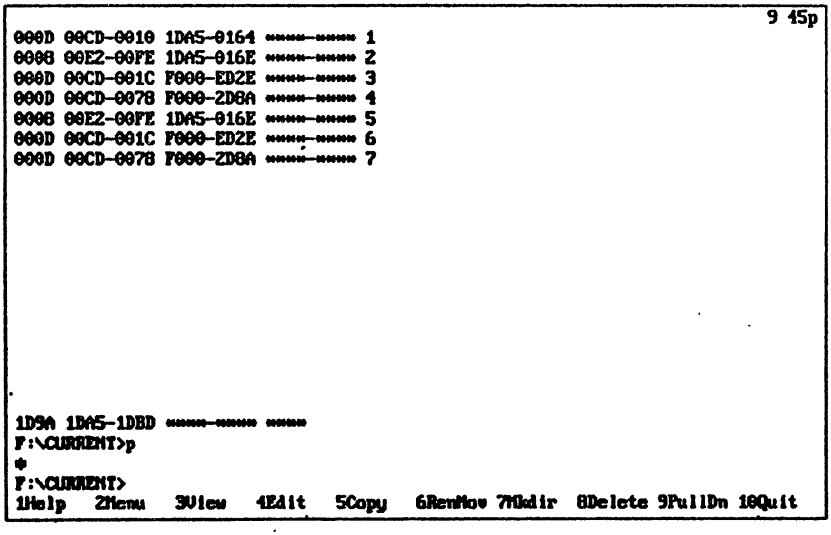


Рис. 76.6. Диагностический вывод программы при отработке прерывания BIOS и двух прерываний от таймера.

Первая диагностическая строка свидетельствует о вызове прерывания BIOS 10h из задачи V86.

Вторая строка диагностирует прерывание с номером 8 - аппаратное прерывание от таймера. Оно прервало выполнение сегмента команд задачи V86 в точке 1DA5:016E на команде с кодом E2h. Это команда loop, входящая в макрокоманду delay.

Далее обработчик был активизирован дважды командами int; первый раз - командой int 1Ch в точке F000:ED2E, а второй - командой int 78h в точке F000:2D8A. О прерывании 1Ch в нашей книге уже говорилось - это фактически заглушка, предназначенная для перехвата пользовательскими программами прерываний от таймера. Команда int 1Ch находится внутри программы BIOS обработки прерываний от таймера. Команда же int 78h находится тоже в ПЗУ BIOS, внутри программы BIOS, обрабатывающей прерывание 1Ch. Таким образом, наш обработчик исключений с системой диагностики позволяет определить, какие программные прерывания и даже из каких точек адресного пространства вызываются системными программами (в данном случае - программами BIOS) в ходе их работы.

Из рис. 76.6 видно, что при заданной задержке таймер успел сработать дважды, причем, как и следовало ожидать, тройка строк от второго прерывания таймера полностью повторяет только что рассмотренную. Ясно, что число шагов в цикле программной задержки в макрокоманде delay, необходимое для получения наглядного результата, определяется скоростью работы процессора и может на разных машинах колебаться в очень широких пределах. Приводимые примеры выполнялись на компьютере с процессором 486DX2 и частотой 50 МГц.

Увеличим задержку

```
delay 25
```

и постараемся успеть нажать на какую-либо клавишу, пока на экран выводятся диагностические строки. Вывод программы может иметь вид, показанный на рис. 76.7 (как видно из строки с системным запросом, была нажата клавиша с буквой a).

Кроме "нашего" прерывания int 10h и четырех аппаратных прерываний от таймера, в диагностических строках видно два прерывания от клавиатуры с вектором 9. Одно из них соответствует нажатию клавиши, второе - отпусканию. Видно, что программа BIOS обработки прерываний от клавиатуры вызывает внутри себя программное прерывание 15h, программа обслуживания которого тоже находится в BIOS. Это прерывание по своим функциям несколько напоминает прерывание 1Ch: оно тоже предназначено для перехвата пользователем аппаратных прерываний, в данном случае от клавиатуры, с целью анализа и фильтрации вводимых пользователем данных.

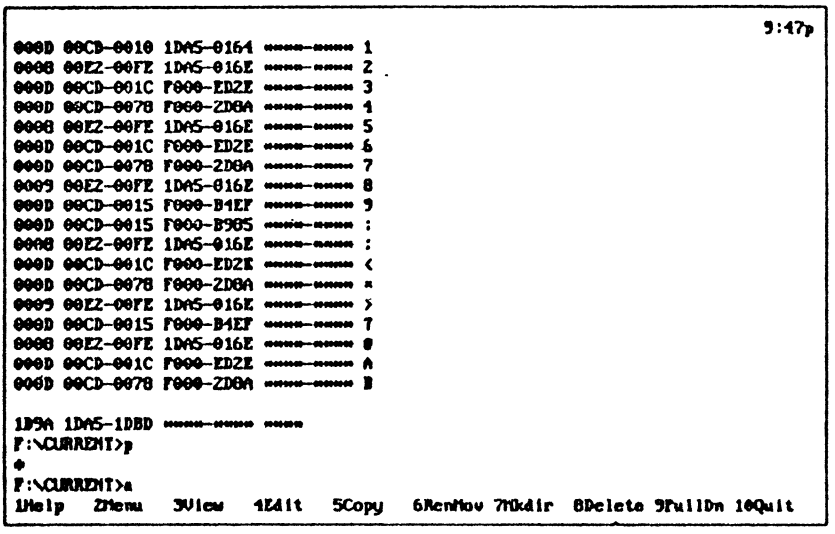
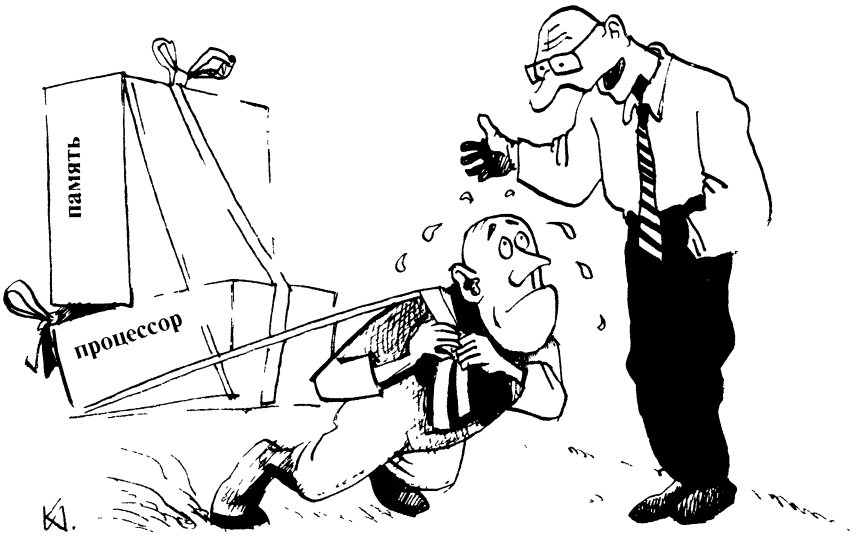


Рис. 76.7. Диагностический вывод программы при отработке прерывания BIOS, нескольких прерываний от таймера и двух прерываний от клавиатуры.

Часть 3

Программирование арифметического сопроцессора



- "С сопроцессором легче и быстрее!" -

Статья 77

Основы работы с арифметическим сопроцессором

Программы, описываемые в 1-2 частях книги, предназначались для обработки символьной информации или выполнения операций над множеством целых чисел. Именно для выполнения таких операций и были созданы микропроцессоры Intel 8086 - i486SX. Для того, чтобы выполнить операцию над действительным числом, имеющим целую и дробную части, необходимо либо использовать специальное программное обеспечение (существенно замедляющее работу программы), либо прибегнуть к услугам арифметического процессора. Арифметические процессоры, которые обычно называются сопроцессорами, обрабатывают информацию, представленную в формате действительных чисел. Фирмой Intel создан ряд сопроцессоров - от 8087 до i487, предназначенных для совместной работы с микропроцессорами 8086 - i486SX, которые в дальнейшем мы будем собирательно называть основным процессором. Наличие сопроцессора позволяет значительно ускорить работу программ, выполняющих расчеты с высокой точностью, тригонометрические вычисления и обработку информации, которая должна быть представлена в виде действительных чисел.

Начиная с модели i486DX, арифметический процессор располагается на том же кристалле, что и основной процессор. Тем не менее в дальнейшем, говоря о командах или особенностях арифметического процессора мы будем называть его сопроцессором, вне зависимости от того, каким образом он реализован.

Прежде чем перейти к составлению программ, остановимся на программной модели сопроцессора, которая показана на рис. 77.1.

Программисту доступны восемь регистров общего назначения, обозначаемых ST(0) - ST(7), и пять нечисловых регистров, которые будут рассмотрены позже. Десятибайтовые регистры ST(0) - ST(7) используются как стек, что подчеркивается их обозначением ST (STack). Они предназначены для хранения операндов и результатов арифметических операций. Первый из этих регистров, наиболее часто используемый, обычно обозначается просто SP.

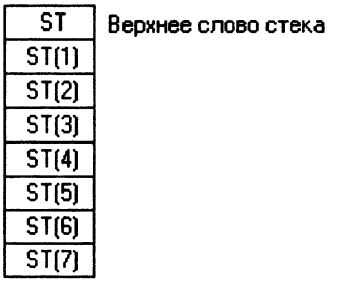


Рис. 77.1. Программная модель сопроцессора.

Вне зависимости от формата, данные, передаваемые на обработку в сопроцессор, преобразуются во внутреннее представление, занимающее 80 бит, и записываются в один из регистров общего назначения. После завершения обработки результат может быть переслан в память, а перед этим преобразован в необходимый формат. Нечисловые регистры используются преимущественно для управления работой сопроцессора.

Для использования сопроцессора надо лишь включить в программу специальные команды - команды сопроцессора. Проиллюстрируем вышесказанное на примере сложения двух целых чисел, занимающих одно слово. Заметим, что данные имеют такие же форматы, как и при работе с основным процессором.

Пример 77.1. Сложение на сопроцессоре целых чисел.

```
text    segment 'code'
assume  cs:text,ds:data
myproc  proc
        mov     AX,data     ;Инициализируем
        mov     DS,AX       ;регистр DS
        fild    x           ;Загрузим первый операнд
                            ;из ячейки памяти x в ST
        fild    y           ;Загрузим второй операнд из ячейки y
                            ;в ST, первый операнд в ST(1)
        fadd                ;Сложим операнды: результат будет
                            ;в ST, ST(1) освобождается
        fistp   z           ;Преобразуем содержимое ST в целый
                            ;формат и запишем в ячейку памяти z
        mov     ax,4C00h
        int     21h
myproc  endp
text            ends
data segment
x       dw      1           ;Первый операнд
y       dw      2           ;Второй операнд
z       dw      ?           ;Результат
data    ends
end     myproc
```

В результате выполнения данной программы в поле z будет записано число 3. Убедиться в этом можно, используя отладчик в режиме пошагового выполнения (работу с отладчиком мы рассмотрим в одной из

следующих статей), или дополнив программу фрагментом, обеспечивающим вывод содержимого поля z на дисплей.

Следует иметь в виду, что использование сопроцессора исключительно для операций над целыми числами нецелесообразно, так как эти действия он выполняет медленнее, чем центральный процессор. Это происходит из-за того, что перед выполнением операции в сопроцессоре числа всегда преобразуются во внутреннюю, действительную форму представления и реально операции выполняются над действительными числами. Так, например, команда сложения содержимого регистра AX с операндом, находящимся в памяти

```
add    AX,mem
```

выполняется за 24+n машинных тактов, а соответствующая команда сопроцессора

```
fiadd  mem
```

за 120+n тактов, где n - количество тактов, необходимое для вычисления адреса операнда mem. При записи в память производится обратное преобразование с округлением, способ которого можно выбрать.

В приведенной выше программе использованы три команды сопроцессора: fild (float integer load, загрузка целого числа в формате плавающей точки) - загрузка целой переменной из памяти в сопроцессор; fadd (float add, сложение целых чисел в формате плавающей точки) - сложение и fistp (float integer store and pop, запись числа с плавающей точкой в формате целого и операция выталкивания из стека) - пересылка результата из сопроцессора в память с преобразованием к целому формату. Рассмотрим (рис. 77.2), как будет изменяться содержимое стека сопроцессора и сегмента данных программы в процессе выполнения этих команд.

Первая команда fild x пересылает значение переменной из памяти в сопроцессор. После ее выполнения в регистр ST оказывается число из поля x. Выполнение следующей команды fild y приводит к тому, что содержимое поля y загружается в вершину стека ST. Загруженная ранее единица будет протолкнута в регистр ST(1).

Команда fadd, в которой отсутствует явное определение данных, выполняет сложение чисел, расположенных в двух верхних регистрах стека, то есть в регистре ST и в регистре ST(1), и записывает результат в ST. Содержимое ST(1) после выполнения этой команды становится неопределенным и в него теперь можно загружать новые числа. Учтите, что загрузка новых чисел в уже занятый регистр сопроцессора не допускается. Если вы попробуете это сделать, возникнет исключение.

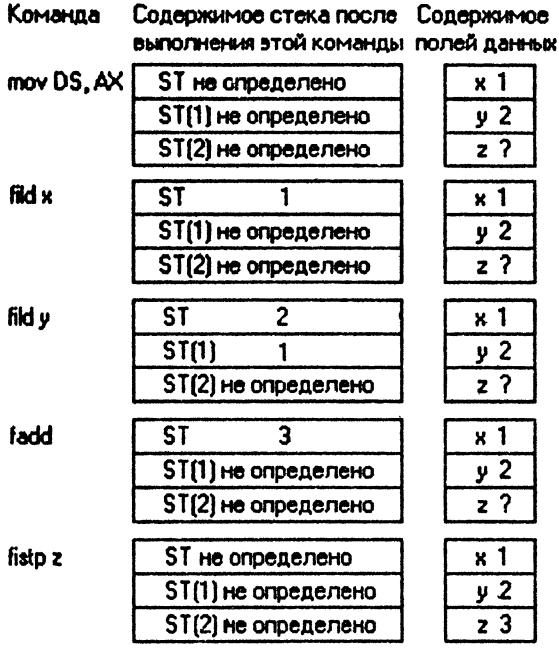


Рис. 77.2. Содержимое полей данных и стека сопроцессора при работе программы.

Команда fistp выполняет запись результата в память и выгрузку регистра ST. Она нужна для преобразования полученного результата из внутреннего представления в целый формат и пересылки его в память. Как видно из рис. 77.2, в процессе выполнения этой команды содержимое ST выталкивается из стека. Если по ходу программы требуется переписать содержимое вершины стека ST в память и оставить стек без изменений, то следует использовать команду fist (float integer store, запись числа в формате плавающей точки в память в формате целого).

Аналогично производится сложение и вычитание целых чисел, для представления которых отводится 4 или 8 байт, и вещественных чисел, занимающих в памяти, в зависимости от требуемой точности 4, 8 или 10 байтов. После выполнения операций над числами их можно записать в память в любом из определенных для сопроцессора форматов, используя соответствующие команды (список команд арифметического сопроцессора приведен в приложении 3).

Статья 78

Работа с действительными числами

Начнем с фрагмента программы, иллюстрирующего сложение действительных чисел (пример 78.1). В нем вместо команд fild, fiadd и fistp, которые мы применяли для сложения целочисленных данных, используются команды fld, fadd и fstp, а для хранения действительных чисел зарезервировано по четыре байта.

Пример 78.1. Сложение действительных чисел.

```
        fld     x           ;Загрузим первый операнд в стек
                            ;сопроцессора
        fld     y           ;Загрузим второй операнд в стек
                            ;сопроцессора
        fadd                ;Сложим их
        fstp    z           ;Перешлем результат в память
;Поля данных
x       dd      1.0         ;Действительные константы должны
y       dd      2.5         ;быть описаны в формате с десятичной
z       dd      (?)         ;точкой, например, 1.0. Описание 1
                            ;недопустимо
```

Напомним, что сопроцессоры специально предназначаются для обработки действительных чисел. Поэтому в стеке сопроцессора все числа представляются в действительном формате. Обычный, используемый по умолчанию формат представления чисел в сопроцессоре показан на рис. 78.1.

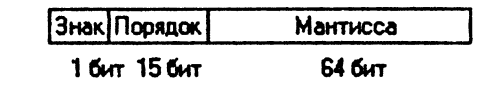


Рис. 78.1. Формат представления действительных чисел в сопроцессоре.

Приведенный выше формат носит название формата расширенной точности. Заметим, что для представления отрицательных действительных чисел дополнительный код не используется. Знак числа в любом формате определяется старшим битом. Если он равен 0, то число положительное, а если 1, то отрицательное. В расширенном формате под порядок отводится 15 бит и 64 бит под мантиссу.

Действительные числа могут быть представлены еще в двух форматах: одинарной точности (4 байт, из которых 23 бит отводятся для ман-

тиссы, 8 для порядка и один для знака числа) и двойной точности (8 байт, 52 бит и 11 бит для мантиссы и порядка соответственно).

Какой из этих форматов используется, определяет содержимое двухбитового поля PC регистра управления сопроцессора (биты 8 и 9). PC=11 соответствует режиму расширенной точности, который автоматически устанавливается при инициализации сопроцессора. При работе в остальных двух режимах результаты вычислений округляются.

Если PC = 10, то выполняется округление до одинарной точности.

Если PC = 00, то выполняется округление до двойной точности.

Следует отметить, что ухудшение точности вычислений не приводит к ускорению работы программы. Поэтому эти режимы целесообразно использовать только в тех случаях, когда требуется выполнять расчеты с пониженной точностью: одинарной или двойной. Для перехода в них используется команда

```
fldcw mem
```

(float load control word, загрузка слова управления сопроцессора), которая загружает в управляющий регистр слово из сегмента данных. (Для записи содержимого управляющего регистра в память используется команда

```
fstcw mem
```

float store control word, сохранение слова управления сопроцессора).

Все действительные числа, независимо от используемого формата, хранятся в нормализованном виде. Нормализованным называется число, целая часть которого состоит из одной не равной нулю цифры. Поскольку при использовании двоичного представления эта цифра всегда будет равна единице, то она не хранится, что позволяет сэкономить один бит памяти. В поле порядка записывается степень числа 2, на которую умножается мантисса, плюс смещение, равное 16386 для расширенной точности, 1023 для двойной точности или 127 для одинарной точности.

Рассмотрим, как представляется число 7,375 при использовании одинарной точности. Во-первых, заметим, что 7,375=4+2+1+1/4+1/8. Поэтому представление этого числа в двоичной форме имеет следующий вид: 111.011b=22*1.11011b. Таким образом, хранимая мантисса числа будет равна 11011b, а порядок 2+127=129, в двоичной форме 10000001b. Поскольку число положительно, знаковый бит равен 0.

Целиком двоичное нормализованное представление числа будет иметь вид, показанный на рис. 78.2.

Именно в таком виде и передаются числа из сопроцессора в память, если не выполнить преобразования к целому формату (естественно, с округлением). Поэтому вывод действительных чисел не является триви-

альной задачей. К счастью, содержимое регистров сопроцессора в отладчике можно посмотреть в обычной форме с плавающей точкой. Отлаживая программу, можно легко контролировать, как протекает процесс вычислений в сопроцессоре. Работе с отладчиком будет посвящена следующая статья, а сейчас еще раз напомним, что любая команда записи в стек сопроцессора выполняется аналогично команде push центрального процессора. Заметим, что стек невелик, его образуют всего 8 регистров, поэтому легко может произойти переполнение. Информация о содержимом регистров стека сведена в регистр признаков (тегов).

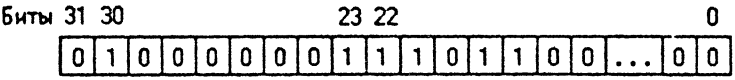


Рис. 78.2. Представление числа 7,375 в двоичной форме при использовании одинарной точности.

Регистр признаков состоит из восьми двухбитовых полей, которые обозначаются TAG0...TAG7. Каждое поле характеризует свой регистр стека (рис. 78.3).

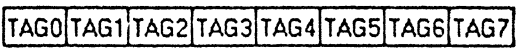


Рис. 78.3. Регистр признаков.

По содержимому поля можно судить о том, какое число хранится в регистре. Если в поле признака находится 00, то в соответствующем регистре расположено действительное ненулевое число, 01 свидетельствует о наличии в регистре нуля, 11 означает, что регистр пуст, а 10 указывает на то, что он содержит недействительное число, например бесконечность.

Если регистр не отмечен как пустой, то при попытке записи в него вырабатывается код недействительной операции (устанавливается бит 0 регистра состояния) и запись в стек не производится.

Заметим, что этот особый случай может быть замаскирован, и тогда запись будет произведена. Для этого в бит 0 регистра управления надо записать 1, иначе возникновение особого случая вызовет прерывание центрального процессора.

Статья 79

Отладка программ, работающих с сопроцессором

Проверка правильности работы программы, включающей команды сопроцессора, затрудняется необходимостью использования процедур ввода и вывода, существенно более сложных, чем при работе с целыми числами. Целесообразно контролировать процесс выполнения подобных программ, используя отладчик. До сих пор в книге рассматривалась работа с компилятором MASM и отладчиком CodeView фирмы Microsoft. Еще одной, может быть более широко используемой инструментальной средой является продукция фирмы Borland: компилятор TASM, компоновщик TLINK и отладчик TD (TurboDebugger, "турбо-отладчик"). Поскольку многим доступно программное обеспечение именно фирмы Borland, рассмотрим работу по созданию и отладке программ для сопроцессора в этой среде.

Работа с компилятором TASM и компоновщиком TLINK осуществляется так же, как и с их аналогами фирмы Microsoft. Отличие состоит в использовании ключей, перечень которых можно получить, введя команды TASM и TLINK без параметров. Для того, чтобы полностью использовать возможности отладчика, при компиляции необходимо указать ключ /ZI - включение полной отладочной информации, а при компоновке ключ /V. Соответствующие команды для подготавливаемой программы с именем P.ASM выглядят следующим образом:

```
TASM/ZI P
TLINK/V P
```

Теперь для выполнения отладки с помощью отладчика TurboDebugger 3.1 достаточно ввести команду

```
TD P.EXE
```

На экране дисплея появится начальный информационный кадр отладчика со строкой главного меню сверху и справочной строкой снизу. В основном поле кадра будет выведен текст программы, причем в точке входа расположен курсор. Им в дальнейшем отмечается команда, которая будет выполняться следующей. Пример начального информационного кадра показан на рис. 79.1.

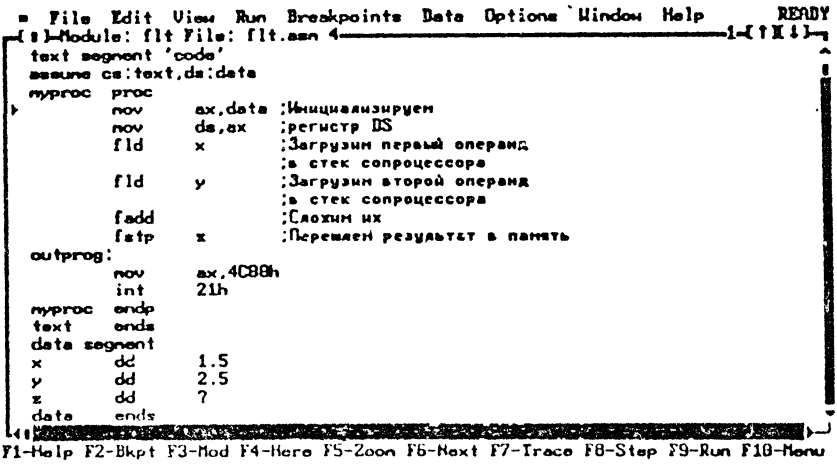


Рис. 79.1. Начальный информационный кадр отладчика TurboDebugger.

На экран также могут быть выведены окна с деассемблированным текстом программы, содержимым регистров и стека основного процессора, а также дампом заданной области памяти. При необходимости на экран можно вывести содержимое полей данных и, что особенно существенно в рассматриваемом случае, регистров сопроцессора. Для того, чтобы определить, какие окна будут выведены на экран, необходимо выбрать в строке главного меню пункт View, а в появившемся затем меню имя необходимого окна.

Работа в среде отладчика TD выполняется с помощью системы меню, как и в большинстве программных продуктов фирмы Borland. Для тех, кто не знаком с этой процедурой, приведем необходимый минимум сведений.

Выбор пункта главного меню осуществляется выделением его имени в главном меню (с помощью мыши или клавиш <Стрелка влево>, <Стрелка вправо>) и нажатия клавиши <Enter>. Всего возможен выбор из десяти пунктов: = (очистка/восстановление экрана), File (работа с файлами), Edit (редактирование), View (выбор просматриваемой информации), Run (управление процессом отладки), Breakpoints (определение точек останова), Data (определение выводимых данных), Options (задание режима работы отладчика), Window (управлением расположением и выводом окон) и Help (вывод справочной информации).

После выбора пункта на экране появляется его меню, работа с которым выполняется аналогично (разница состоит лишь в том, что перемещение по внутреннему меню пункта осуществляется с помощью кла-

виш <Стрелка вверх>, <Стрелка вниз>). Например, после выбора пункта View на экране появляется меню, определяющее возможные окна наблюдения (рис. 79.2).

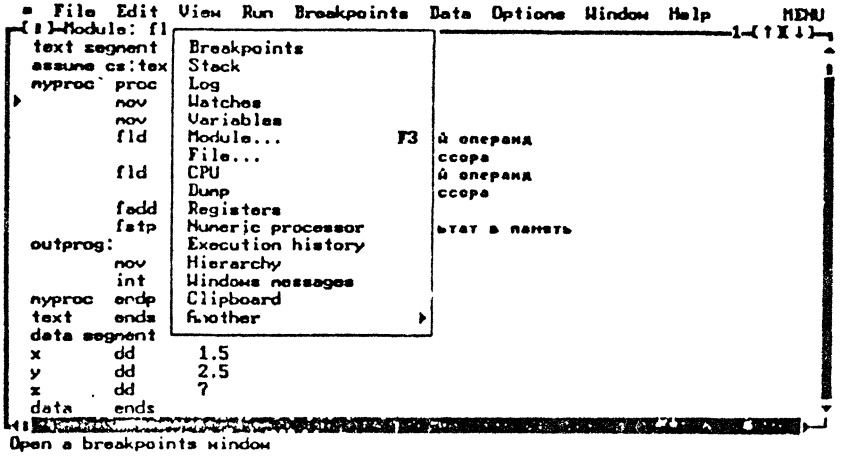


Рис. 79.2. Окно возможных режимов наблюдения.

Для перехода в главное меню необходимо нажать клавишу <F10>.

Для вывода на экран регистров сопроцессора необходимо в меню пункта View выбрать пункт Numeric processor. В информационном кадре появится окно, в котором указаны имена и содержимое регистров стека сопроцессора и флагов. При отладке полезно также вывести окно регистров основного процессора. После этого надо определить такое положение окон, чтобы они не слишком загораживали друг друга и программу (рис. 79.3.).

Для перемещения окна можно воспользоваться "горячими клавишами". Нажатие комбинации клавиш <Ctrl>/<F5> приводит к тому, что рамка активного окна изменяет цвет и ее можно перемещать в пределах информационного кадра отладчика с помощью клавиш <Стрелка вправо>, <Стрелка влево>, <Стрелка вверх> и <Стрелка вниз>. Поскольку последнее открытое окно является активным, то изменять его положение лучше сразу после открытия. Переход в режим изменения размеров окна достигается с помощью тех же "горячих клавиш", а само изменение выполняется клавишами <Стрелка вправо>, <Стрелка влево>, <Стрелка вверх> и <Стрелка вниз>, при работе с которыми надо удерживать нажатой клавишу <Shift>. Чтобы зафиксировать положение и размер окна, нажмите клавишу <Enter>.

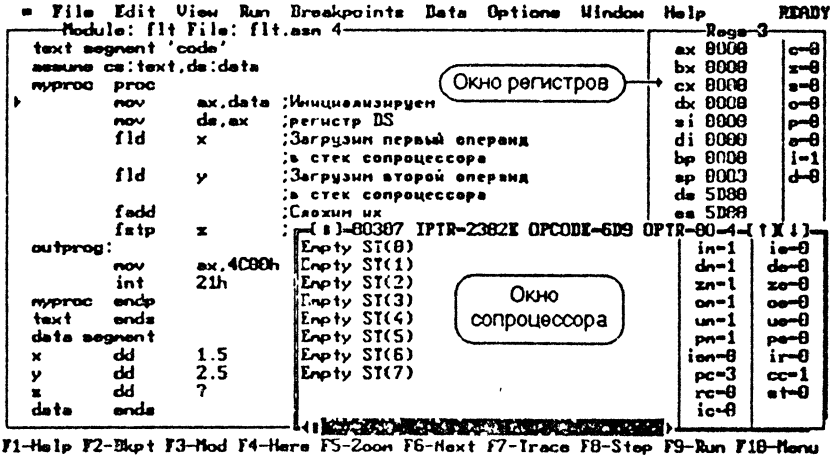


Рис. 79.3. Пример расположения окон вывода в информационном кадре отладчика TurboDebugger.

Теперь, когда экран полностью сформирован, приступим к отладке программы. Проще всего работать в режиме пошаговой отладки. Для этого используется клавиша F8, каждое нажатие которой вызывает выполнение одной команды. Изменения регистров, флагов и данных можно увидеть на экране. Отладчик Turbodebugger позволяет не только последовательно, команда за командой, выполнять программу, но и пошагово возвращаться назад (комбинация клавиш <Alt>/<F4>). Это возможно потому, что отладчик хранит последние выполненные команды. Для того, чтобы вернуться к началу программы и выполнить ее еще раз, достаточно нажать <Ctrl>/<F2>.

В процессе отладки можно, не выходя из отладчика, изменять содержимое регистров и полей памяти, а также устанавливаемые флаги. Для этого используется локальное меню, открываемое нажатием комбинации клавиш <Alt>/<F10> или нажатием правой кнопки мыши. Локальное меню контекстно чувствительно, то есть для каждого активного окна имеется свое локальное меню.

Рассмотрим, например, как изменить содержимое регистра ST. Для этого в окне сопроцессора сначала выделим данный регистр, а затем вызовем локальное меню, которое включает три пункта: Zero, Empty и Change (рис. 79.4). Напомним, что окно сопроцессора должно быть активным, поскольку локальное меню вызывается для активного окна.

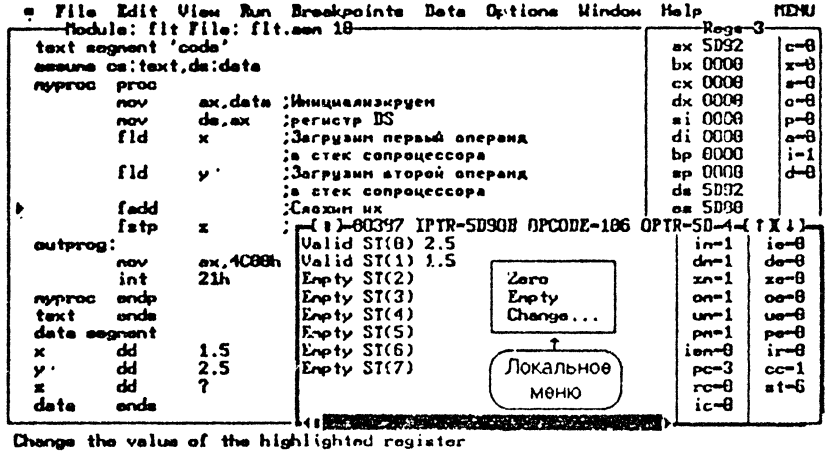


Рис. 79.4. Локальное меню окна сопроцессора.

Если мы выберем пункты Zero или Empty, то в регистр ST запишется нуль или же он освободится, то есть станет доступным для записи. Для того чтобы записать в этот регистр новое число, надо выбрать пункт Change. Это вызовет появление окна, в поле которого надо ввести требуемое число и выбрать пункт OK. После этого содержимое регистра стека сопроцессора будет изменено. Аналогично можно изменить содержимое полей памяти и регистров основного процессора. Для переключения флага необходимо его выделить, вызвать локальное меню, в котором на этот раз будет только один пункт - Toggle и подтвердить решение об изменении флага.

Чтобы при отладке следующей программы или в другом сеансе работы с той же программой не требовалось заново определять вид выводимой информации, можно запомнить текущую конфигурацию. Для этого выберите пункт меню Options, а в нем пункт Save Options. На экране появится окно, в котором необходимо определить сохраняемые параметры. Если нам необходимо сохранить выводимые окна, то в поле перед словом Layout должен быть установлен символ *. Если его нет, то он устанавливается нажатием пробельной клавиши. Переход на это поле осуществляется с помощью клавиши Tab. После завершения всех установок перейдите в поле OK и нажмите на <Enter> или выберите поле OK с помощью мыши. Теперь конфигурация запомнена. Для выхода из отладчика выберите в меню пункта File пункт Exit.

Заметим, что во всех приведенных выше примерах в процессе отладки мы работали с исходным текстом программы.

В ряде случаев бывает необходимо использовать деассемблированный текст. Для вывода его на экран в меню пункта View надо выбрать пункт CPU. Пример окна CPU с выведенным деассемблированным текстом для рассматриваемой программы показан на рис. 79.5.

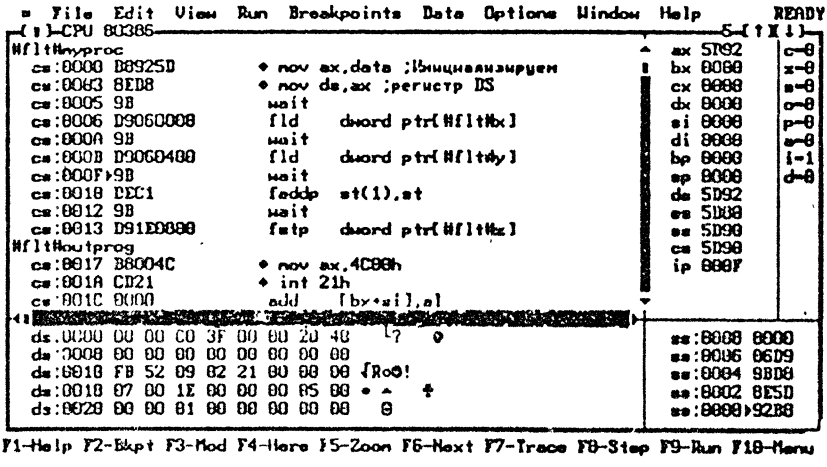


Рис. 79.5. Информационный кадр отладчика с деассемблированным текстом программы.

Как видно из рисунка, окно, выводимое на экран при выборе пункта CPU, включает пять внутренних окон. Левое верхнее окно содержит деассемблированный текст программы. Справа от него располагается окно регистров. Еще правее, в отдельном окне выведены флаги процессора. В самом низу окна CPU располагается окно данных, в котором отображается дамп заданного участка памяти. Последнее, пятое окно, содержит дамп текущей вершины стека главного процессора. Оно располагается в правом нижнем углу окна CPU.

Мы рассказали об основах работы с отладчиком TurboDebugger, без знания которых невозможно отлаживать программы, работающие с сопроцессором. Для изучения остальных возможностей отладчика можно воспользоваться встроенным справочником, который вызывается при выборе пункта Help главного меню.

Статья 80

Как определить, есть ли у вас сопроцессор?

Поскольку сопроцессором комплектуются не все ПК, то программы, использующие сопроцессор, должны сами уметь определять его наличие, что можно выполнить различными способами.. Проще всего прочитать информацию из слова области данных BIOS, расположенного по адресу 0410h. Если в прочтенном слове бит 12 установлен, то сопроцессор есть, если бит сброшен, сопроцессор отсутствует. Однако, как показывает практика, такой способ определения наличия сопроцессора ненадежен. Очевидно, это происходит из-за несоблюдения соглашений по содержанию информации в области данных BIOS его разработчиками. Поэтому мы рекомендуем пользоваться другими способами.

Первый основан на чтении информации о конфигурации компьютера и зависит от типа ПК. Так, например, в XT и некоторых AT информация о конфигурации хранится в микросхеме 8255, имеющей три однобайтовых регистра A, B и C с адресами соответствующих портов 60h, 61h и 62h. Для того, чтобы считать содержимое из порта C, в котором хранится информация о конфигурации, необходимо предварительно установить бит 3 порта B. Если в считанном из порта C байте бит 1 установлен, то сопроцессор есть, в противном случае он отсутствует. Приведем фрагмент программы, проверяющей наличие сопроцессора.

Пример 80.1. Проверка наличия сопроцессора с помощью чтения информации о конфигурации компьютера.

```
        in      AL,61h      ;Получим значение из порта 61h
        mov     BL,AL       ;Сохраним AL
        or      AL,1000b    ;Установим бит 3 (для XT, AT)
        out     61h,AL      ;Установим в порте 61h бит 3, чтобы
                            ;обеспечить чтение конфигурации
        in      AL,62h      ;Прочитаем слово конфигурации
        and     AL,00000010b;Сбросим в нем все биты, кроме 2^1
        cmp     AL,0        ;Проверим наличие сопроцессора
        je      non         ;При отсутствии выполним переход
;Вывод сообщения coopr о наличии сопроцессора
        ...
non:
;Вывод сообщения errmes об отсутствии сопроцессора
        ...
;Поля данных
```

```
errmes  db      'Сопроцессор отсутствует$'
coopr   db      'Сопроцессор имеется$'
```

Проверку наличия сопроцессора можно выполнить еще проще с помощью прерывания 11h, возвращающего информацию об оборудовании. Фрагмент программы, реализующий этот способ, показан ниже.

Пример 80.2. Проверка наличия сопроцессора с помощью прерывания 11h.

```
        int     11h             ;В AX возвращается информация об
                                ;имеющемся периферийном оборудовании
        and     AL,00000010b;Сбросим все биты, кроме бита 1
        cmp     AL,0            ;Проверим наличие сопроцессора
        je      non             ;При отсутствии выполним переход
;Вывод сообщения coopr о наличии сопроцессора
        ...
non:
;Вывод сообщения errmes об отсутствии сопроцессора
        ...
;Поля данных
errmes  db      'Сопроцессор отсутствует$'
copr    db      'Сопроцессор имеется$'
```

Приведенными способами проверка наличия сопроцессора не исчерпывается. Можно, например, попытаться инициализировать сопроцессор командой finit (float initial, инициализация сопроцессора), а затем прочесть содержимое его управляющего регистра. При инициализации сопроцессора, если он присутствует, в управляющий регистр записывается определенный код. Фрагмент программы, реализующий данный способ, показан в примере 80.3.

Пример 80.3. Проверка наличия сопроцессора с помощью чтения содержимого управляющего регистра.

```
        finit               ;Инициализируем сопроцессор
        fstcw  ctrl_reg     ;Занесем содержимое управляющего регис-
                            ;тра сопроцессора в поле данных ctrl_reg
        fwait               ;Синхронизация работы сопроцессора и
                            ;основного процессора
        and    ctrl_reg,0F3Fh;Выделим разряды по маске
                            ;0000111100111111b
        cmp    ctrl_reg,033Fh;Проверим на равенство числу 33Fh
        jne    non          ;Если не 033Fh, сопроцессора нет
;Вывод сообщения coopr о наличии сопроцессора
        ...
non:
;Вывод сообщения errmes об отсутствии сопроцессора
;Поля данных
ctrl_regdw      ?
errmes  db      'Сопроцессор отсутствует$'
copr    db      'Сопроцессор имеется$'
```

Если сопроцессор действительно присутствует, то после выполнения команды finit (более подробно эта команда будет рассмотрена в статье

85) в его управляющий регистр будет записана заранее определенная информация. Маской 0F3Fh для дальнейшей обработки выделяются разряды, управляющие точностью результатов выполнения арифметических операций (с восьмого по одиннадцатый) и маскированием исключительных ситуаций (с нулевого по пятый). Остальные разряды сбрасываются. Выделенное значение должно быть равно 033Fh или, в двоичном представлении, 0000001100111111b. Если это не так, то сопроцессора нет.

К достоинствам этого метода относится возможность определения типа сопроцессора, так как разные сопроцессоры после инициализации возвращают отличающуюся информацию, например, в 15-ом разряде регистра управления.

Статья 81

Выполнение арифметических операций

В распоряжении программиста, использующего сопроцессор, имеется около сотни команд. К ним относятся команды записи в стек сопроцессора целых и действительных чисел, сохранения полученных результатов в памяти, выполнения операций сложения, вычитания, умножения и деления, а также команды вычисления тригонометрических и некоторых других функций. Перечень команд приведен в приложении 3; сейчас мы отметим их общие особенности.

Команды сопроцессора легко узнать, так как все они начинаются с буквы f (float). В этой статье для указания на произвольную команду сопроцессора будут использоваться обозначения fcmd и fcmdp (второе для команд с извлечением из стека).

Команды сопроцессора могут иметь два, один или ни одного явно задаваемых операнда. Однако фактически большинство команд являются двухоперандными; просто в ряде случаев эти операнды используются неявно.

В качестве операндов используются регистры сопроцессора и поля памяти. При указании поля памяти могут применять все способы адресации, принятые в основном процессоре. Если, однако, один из операндов является полем памяти, то в качестве другого обязательно используется верхний регистр стека ST. В отличие от основного процессора, в командах сопроцессора не используются непосредственные опе-

ранды и, за исключением команды fstsw, не разрешается адресовать регистры основного процессора. Так же, как и для основного процессора, оба операнда одновременно не могут быть полями памяти.

Команды обычно имеют операнд-источник и операнд-приемник. После выполнения команды операнд-источник не изменяется, а значение операнда-приемника замещается результатом. Если у команды определено два операнда, то первый является приемником, а второй источником. Возможные способы использования операндов показаны в табл. 81.1.

Таблица 81.1. Способы использования операндов в командах сопроцессора.

Операнды	Формат команды	Неявные операнды	Примеры	Результат
Стек	fcmd	ST,ST(1)	fcom	ST-ST(1)
Память	fcmd mem	ST	fdiv mem	ST=ST/mem
Регистр	fcmd ST,ST(i) fcmd ST(i),ST		fsub ST,ST(4) fmul ST(3),ST	ST=ST-ST(4) ST(3)=ST(3)*ST
Регистр, съем со стека	fcmdp ST(i),ST fcmdp ST(i)	ST	fmulp ST(6),ST faddp ST(5)	ST(6)=ST(6)*ST, чтение ST(5)=ST(5)+ST, чтение
Стек, съем со стека	fcmdp	ST,ST(1)	faddp	ST(1)=ST+ST(1), чтение

Рассмотрим простую программу, использующую сопроцессор. Приведенный в примере 81.1 фрагмент обеспечивает вычисление интеграла J от функции f методом Симпсона по трем точкам. В программе реализовано вычисление по формуле

```
J=h*(f(-h)+4f(0)+f(h))/3
```

где h - шаг интегрирования.

Пример 81.1. Фрагмент программы вычисления интеграла

```
xor     SI,SI       ;(1)Очистим регистр SI
fld     f[SI]       ;(2)Загрузим f(-h)
add     SI,4        ;(3)Увеличим смещение на длину
                    ;двойного слова
fld     f[SI]       ;(4)Загрузим f(0)
add     SI,4        ;(5)Еще раз увеличим смещение
fild    k4          ;(6)Загрузим коэффициент 4
fmul                ;(7)Умножим 4 на f(0)
fadd                ;(8)Получим f(-h)+4f(0)
fld     f[SI]       ;(9)Загрузим f(h)
```

```
        fadd                ;(10)Получим f(-h)+4f(0)+f(h)
        fmul    h           ;(11)Умножим полученное значение
                            ;на величину шага интегрирования
        fild    k3          ;(12)Загрузим коэффициент 3
        fdiv                ;(13)Получим h*(f(-h)+4f(0)+f(h))/3
        fadd    j           ;(14)Добавим к полученным ранее зна-
        fstp    j           ;(15)чениям и перенесем в поле j
;Поля данных
f       dd      2.7182818,1.9477340,1.6487213;Отсчеты функции
;Коэффициенты
k3      dw      3
k4      dw      4
h       dd      0.5
;Поле данных для интеграла
j       dd      ?
```

В этом фрагменте использован целый ряд способов адресации. Рассмотрим выполняемые в нем действия более подробно. После очистки регистра SI мы заносим в стек сопроцессора первый отсчет интегрируемой функции (предложение 2). Поскольку значения отсчетов заданы в сегменте данных, для этой цели используется команда загрузки стека, операндом которой является поле памяти. Для указания смещения использован косвенный метод адресации через индексный регистр SI.

Следующее предложение 3 является командой основного процессора. Она увеличивает смещение таким образом, чтобы в предложении 4 выполнилась загрузка очередного отсчета интегрируемой функции. Затем мы снова увеличиваем смещение и переходим к вычислению интеграла. Последний загруженный отсчет представляет собой f(0), поэтому мы загружаем коэффициент 4 (предложение 6) и умножаем его на f(0), используя команду умножения без явного указания операндов (предложение 7). Это возможно, так как к моменту выполнения предложения 7 в регистре ST(1) находится f(0), а в ST коэффициент 4.

После того, как с помощью предложений 8...13 значение интеграла вычислено, а при данных значениях отсчетов и шага оно будет равно 2.02632, мы суммируем его с ранее полученными результатами (предложение 14) и пересылаем в поле j сегмента данных (предложение 15). Для пересылки используем команду формата

```
fcmdp  mem
```

обеспечивающую удаление из стека пересылаемого значения *mem*.

В рассмотренном фрагменте встречаются как команды сопроцессора, так и команды основного процессора. Рассмотрим, например, предложения 1 и 2. В предложение 1 входит команда основного процессора, а в предложение 2 - команда сопроцессора; они выполняются друг за другом и, кроме того, результат выполнения первой команды используется при выполнении второй. Поскольку сопроцессор и основной процессор могут работать параллельно, то для правильного выполнения

этого фрагмента необходимо передать команду предложения 2 на выполнение сопроцессору только после того, как выполнится команда предложения 1. Как же это организуется при работе с сопроцессором? Обратим внимание на листинг программы из примера 81.1, фрагмент которого приведен на рис. 81.1.

```
0000    Коды команд    text    segment
                       assume CS:text,DS:data;
0000         ↓         myproc proc
0000  B8 0000s         mov  AX,data
0003  8E D8            mov  DS,AX
0005  9B DB E3         finit
0008  33 F6            xor  SI,SI ;(1)Очистим регистр SI
000A  9B D9 84 0000r    fld  f[SI] ;(2)Загрузим f(-h)
000F  83 C6 04          add  SI,4  ;(3)Увеличим смещение на длину
                                   ;двойного слова
0012  9B D9 84 0000r    fld  f[SI] ;(4)Загрузим f(0)
0017  83 C6 04          add  SI,4  ;(5)Еще раз увеличим смещение
001A  9B DF 06 000Er    fild k4    ;(6)Загрузим коэффициент 4
001F  9B DE C9          fmul       ;(7)Умножим 4 на f(0)
0022  9B DE C1          fadd       ;(8)Получим f(-h)+4f(0)
0025  9B D9 84 0000r    fld  f[SI] ;(9)Загрузим f(h)
002A  9B DE C1          fadd       ;(10)Получим f(-h)+4f(0)+f(h)
002D  9B D8 0E 0010r    fmul h     ;(11)Умножим полученное значение
                                   ;на величину шага интегрирования
0032  9B DF 06 000Cr    fild k3    ;(12)Загрузим коэффициент 3
0037  9B DE F9          fdiv       ;(13)h*(f(h)+4f(0)+f(h))/3
003A  9B D8 06 0014r    fadd j     ;(14)Добавим к полученным ранее
003F  9B D9 1E 0014r    fstp j     ;(15)значениям и перенесем в j
0044  9B                fwait      ;(16)Подождем завершения команды
0045  B8 4C00          mov  AX,4C00h
0048  CD 21             int  21h
004A                    myproc endp
004A                    text ends
```

Рис. 81.1. Листинг программы примера 81.1.

Из приведенного листинга видно, что код каждой из команд сопроцессора начинается с шестнадцатеричного числа 9B. Это число является кодом команды wait, которая задерживает работу сопроцессора до тех пор, пока не закончит работу основной процессор. Таким образом, для обеспечения синхронизации ассемблер вставляет перед каждой командой сопроцессора команду wait. Это можно легко увидеть в отладчике, если пользоваться окном CPU, в которое выводится деассемблированный текст программы (рис. 81.2). Обратите, кстати, внимание на то, что в строках листинга, соответствующих командам сопроцессора, после кода 9B стоит шестнадцатеричная цифра D. Она является кодом первых четырех бит пятибитной команды esc (полный код этой команды 11011), которая обеспечивает переключение на сопроцессор, и обязательно предшествует каждой команде сопроцессора.

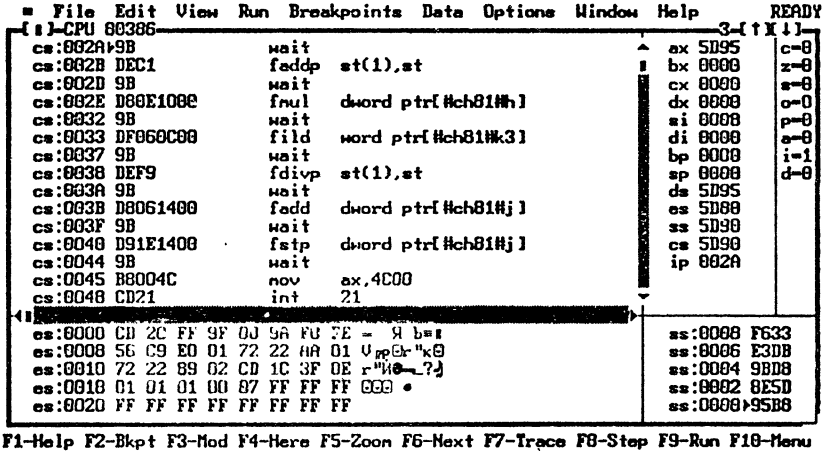


Рис. 81.2. Информационный кадр отладчика с деассемблированным текстом программы.

В поле деассемблированных команд окна CPU выводится полный адрес, код команды и ее мнемоническое обозначение. Перед каждой командой сопроцессора расположена строка с командой синхронизации wait. Обратите внимание, что команда fwait воспринята деассемблером отладчика как команда wait. Дело в том, что, хотя она начинается с буквы f и внешне выглядит как команда сопроцессора, ее код равен уже знакомому нам числу 9B (предложение 15 листинга на рис. 81.1), и, следовательно, совпадает с кодом команды wait основного процессора. Поэтому она является просто другим мнемоническим обозначением данной команды. Команда fwait применяется в тех ситуациях, когда необходимо синхронизировать действия центрального процессора и сопроцессора, то есть приостановить в них выборку команд из очереди до завершения сопроцессором текущей команды. Ее наличие приводит к тому, что центральный процессор не может обратится к операнду в памяти до тех пор, пока этот операнд не будет записан в память сопроцессором. Пример подобной ситуации показан ниже.

Пример 81.2. Использование команды fwait.

```
        fstsw  sword        ;Записываем в поле данных sword
                            ;содержимое слова состояния сопроцессора
        fwait               ;Ждем, пока сопроцессор не завершит
                            ;выполнения команды fstsw
        mov    AX,sword     ;Перешлем содержимое слова состояния
                            ;в регистр AX
;Поля данных
sword   dw     ?
```

Подобные фрагменты мы будем использовать в последующих статьях, в тех случаях, когда потребуется передать в основной процессор значения флагов слова состояния сопроцессора.

Статья 82

Использование сопроцессора для реализации операции возведения положительного числа в дробную степень

Наличие сопроцессора позволяет существенно упрощать решение задач, требующих расчетов показательных функций. Это происходит из-за того, что вычисление таких функций можно выполнить с помощью одной или нескольких команд, и не требуется составлять специальные процедуры, предназначенные, например, для вычисления квадратного корня. Рассмотрим использование сопроцессора для вычисления функции y=s*sqrt(x) и вывода ее графика на экран монитора. Вычислим 400 значений функции и выведем их на экран в виде красных точек. Для этого предварительно установим графический режим 10h (VGA с разрешением 640*350, 16 цветов), доступный при работе с подавляющим большинством современных адаптеров. Сделаем это с помощью функции 0h прерывания 10h BIOS. Затем перешлем в стек сопроцессора значение коэффициента s и вычислим значение функции. Для вывода точек на экран воспользуемся функцией 0Ch прерывания 10h BIOS. Фрагмент программы приведен в примере 82.1.

*Пример 82.1. Вычисление и вывод на экран графика функции y=a*sqrt(x).*

```
;Установка параметров
        mov     AH,0        ;Функция установки графического режима
        mov     AL,10h      ;Режим 10h-графика, 640*350 точек
        int     10h         ;Прерывание BIOS
        fild    s           ;Зашлем коэффициент s в регистр ST
        mov     CX,400      ;Число повторений цикла
;Цикл вычисления координат и вывода точек на экран
line:   push    CX          ;Сохраним счетчик цикла
        fild    x           ;Зашлем x в ST, s идет в ST(1)
        fsqrt               ;Корень из ST, результат там же
        fmul    ST,ST(1)    ;Умножим на s, результат в ST
        fistp   y           ;Перешлем содержимое ST в ячейку y
        mov     AH,0Ch      ;Функция вывода пиксела
        mov     AL,4        ;Цвет пиксела
```

```
        mov     BH,0        ;Видеостраница 0
        mov     CX,x        ;Номер столбца
        mov     DX,100
        sub     DX,y        ;Номер строки
        int     10h         ;Прерывание BIOS
        inc     x           ;Сместимся вправо
        pop     CX          ;Восстановим счетчик цикла
        loop    line        ;Повторим CX раз
;Поля данных
forcolor db     4           ;Цвет
x       dw      0
y       dw      ?
s       dw      4
```

Непосредственно для работы с сопроцессором здесь использовано всего 5 команд. С их помощью производится засылка в стек сопроцессора коэффициента s и текущего аргумента x, вычисление sqrt(x) с сохранением результата в стеке, вычисление произведения s*sqrt(x) и пересылка полученного результата в ячейку y. Применение команды fist позволяет перед пересылкой преобразовать число к целому формату. Поэтому его сразу можно использовать для указания координаты y.

Вычисление произвольной степени положительного числа представляет собой более сложную задачу. Для ее решения можно использовать команды сопроцессора f2xm1 (float 2x-1, вычисление в формате плавающей точки выражения 2x-1) и fyl2x (float y*log2x, вычисление в формате плавающей точки выражения y*log2x). Команда f2xm1 вычисляет два в степени ST минус один, при этом значение ST должно находится в диапазоне от 0.0 до 0.5. Команда fyl2x производит умножение содержимого регистра ST(1) на логарифм ST по основанию два. Таким образом, чтобы возвести число в произвольную степень, имея в наличии эти две команды, необходимо вычислить логарифм (по основанию 2) от основания степени, умножить его на показатель степени и возвести два в степень, значение которой равно получившемуся результату. Если при этом соблюдается условие для выполнения команды f2xm1, то процедура будет очень простой. Рассмотрим, например, как вычислить кубический корень из конкретного числа (пример 82.2).

Пример 82.2. Вычисление кубического корня из числа 2,5.

```
        fld1                ;Загрузим в стек сопроцессора 1
        fld     n           ;Загрузим в стек степень корня
        fdiv                ;Вычислим показатель степени,
                            ;в которую будет возводится число
        fld     a           ;Загрузим основание
        fyl2x               ;Вычислим логарифм искомого числа
        f2xm1               ;Вычислим (корень-1)
        fld1                ;Загрузим единицу
        fadd                ;Получаем корень
        fstp    rez         ;Пересылаем результат в память
```

```
;Поля данных
a       dd      2.5
n       dd      3.0
rez     dd      ?
```

В приведенном примере команда fdiv (float divide, деление чисел в формате плавающей точки) использована для того, чтобы в сегменте данных задавать показатель корня, а не дробную степень. В результате ее выполнения в регистр ST засылается ST(1)/ST. Регистр ST освобождается.

Если показатель степени, в которую надо возвести число два, чтобы получить результат, представляет собой положительное число, то перед выполнением команды f2xm1 надо выделить его целую часть. Затем следует определить диапазон, в котором лежит дробная часть. Если дробная часть превышает 0.5, то надо уменьшить ее на эту величину и домножить результат на квадратный корень. И только после этого можно выполнять команду f2xm1. Фрагмент программы, включающий операции проверки, приведен ниже.

Пример 82.3. Фрагмент программы возведения положительного числа в произвольную положительную степень.

```
        finit
        fld     n               ;Загрузим в стек показатель степени
        fld     a               ;Загрузим основание
        fyl2x                   ;Вычислим логарифм по основанию 2
        fst     ST(1)           ;Запишем его в регистр ST(1)
;Установим способ округления
        fstcw   contrl          ;Прочитаем и сохраним слово
                                ;управления сопроцессора в contrl
        fwait                   ;Подождем выполнения этой команды
        or      contrl,0C00h;Режим отбрасывания дробной части
        fldcw   contrl          ;Загрузим новое слово управления
        frndint                 ;Округляем, получая целую часть
;Если целая часть не равна 0, то передадим полученное число в
;основной процессор и вычислим 2 в степени, равной целой части
        fist    stn
        cmp     stn,0
        jz      cont
        fsub                    ;Получим дробную часть
        fwait
;На основном процессоре вычислим 2 в степени, равной целой части
        mov     CX,stn
        sub     CX,1
        mov     AX,2
        shl     AX,CL
        mov     res,AX
        fild    res             ;Загрузим результат в стек
        fstp    rezlt           ;и перепишем его в память в виде
                                ;действительного числа
;Проверим величину получившегося остатка
cont:   fst     ST(1)           ;Сохраним получившийся остаток
        fild    const2          ;Загрузим 2 в стек сопроцессора
```

```
        fmul                ;Помножим остаток на 2
        frndint             ;и отбросим дробную часть
        fistp  pj           ;Запишем результат в память
                            ;в виде дробного числа
        fwait               ;Ожидание выполнения операции
        cmp    pj,1         ;Сравним с 1
;Если остаток меньше 0.5, то перейдем к вычислению 2 в степени,
;равной остатку, иначе - домножим результат на корень из двух
        jl     cont1
        fsub   const05
        fild   const2
        fsqrt
        fmul   rezlt        ;Домножим результат на корень из двух
        fstp   rezlt        ;Запишем в ячейку rezlt
cont1:  f2xm1               ;Вычислим 2 в степени, равной остатку
        fld1                ;Выполним коррекцию
        fadd
        fld    rezlt
        fmul                ;Получим искомое число,
        fstp   rezlt        ;которое перешлем в память
;Поля данных
a       dd     125.0        ;Число, возводимое в степень
n       dd     0.33333333   ;Показатель степени
rezlt   dd     ?            ;Поле для сохранения результата
const2  dw     2            ;Используемые константы
const05 dd     0.5
res     dw     0            ;Поля для хранения промежуточных данных
stn     dw     ?
pj      dw     ?
contrl  dw     ?            ;Поле для сохранения слова
                            ;управления сопроцессора
```

Приведенную программу можно оптимизировать, выполняя сравнение в сопроцессоре и округляя получившийся после вычисления логарифма результат до ближайшего целого. Затем следует возвести 2 в это целое число и, в зависимости от знака разности между округленным и неокругленным числами, выбрать один из двух вариантов. Можно также умножить 2 в степени, равной целой части, на 2 в степени, равной остатку, или разделить на него.

Статья 83

Вычисление корня нелинейного уравнения F(x)=0

Удобство использования сопроцессора легко иллюстрируется на примере решения задачи определения корня уравнения F(x)=0 с заданной точностью delta<<1. Широко известный метод половинного деления состоит из следующих операций. Сначала вычисляются значения функции в точках, расположенных через равные интервалы на оси x. Это делается до тех пор, пока не будут найдены два последовательных значения F(x[n]) и F(x[n+1]), имеющие противоположные знаки. Если функция непрерывна, то изменение знака указывает на существование корня. Предположим, что x[n] и x[n+1] уже найдены и x[n]=a, x[n+1]=b, причем F(a)<0, а F(b)>0. Кроме того, на этом отрезке функция Y=F(x) имеет один нуль. Тогда график функции пересекает ось только в одной точке z, которую и требуется найти.

Для этого вычисляется точка c=(a+b)/2. Если F(c)>0, то заменяем точку b на точку c, иначе точкой c заменяется точка a. Процедуру продолжаем до тех пор, пока не выполнится условие |F(a)-F(b)|<delta. После этого z определяется как (a+b)/2. Эффективное решение такой задачи возможно только при наличии сопроцессора. Фрагмент программы, реализующей поиск корня уравнения x*x-p*sqrt(x)=0 на интервале [0.0001,1], приведен в примере 83.1.

Пример 83.1. Программа вычисления корня уравнения F(x)=0.

```
cont:                   ;Начало цикла
;Вычисление F(a)
        fld    a        ;a -> ST
        fsqrt           ;sqrt(x) -> ST
        fld    p        ;sqrt(x) -> ST(1), p -> ST
        fmul            ;p*sqrt(x) -> ST
        fld    a        ;p*sqrt(x) -> ST(1), a -> ST
        fld    a        ;p*sqrt(x) -> ST(2), a -> ST(1), a -> ST
        fmul            ;a*a -> ST(0), p*sqrt(x) -> ST(1)
        fadd            ;F(a) -> ST
;Вычисление F(b)
        fld    b        ;F(a) -> ST(1), b -> ST
        fsqrt
        fld    p
        fmul
        fld    b
        fld    b
```

```
        fmul
        fadd                ;F(b) -> ST, F(a) -> ST(1)
;F(a) и F(b) вычислены
        fsub                ;F(a)-F(b) -> ST
        fabs                ;|F(a)-F(b)| -> ST
        fcom    delta       ;Проверка результатов выполнения
                            ;операции (|F(a)-F(b)|<delta)
        fstsw   sw          ;Передача слова состояния в основной
                            ;процессор
        fwait
        mov     AX,sw
        and     AX,0500h    ;Выделение битов C2 и C0
        cmp     AH,1        ;Проверка наличия комбинации C2=0, C0=1
        je      ed          ;z найдено!
        ffree   ST(0)       ;Очистка ST
        fld     a           ;Вычисление z
        fld     b
        fadd
        fdiv    dlt         ;Деление на два, z -> ST
        fst     z           ;Результат в z
;Вычисление F(z)
        fsqrt
        fld     p
        fmul
        fld     z
        fld     z
        fmul
        fadd                ;F(z) -> ST
        ftst                ;F(z)>0?
        fstsw   sw          ;Передача слова состояния в основной
                            ;процессор
        fwait
        mov     AX,sw
        and     AX,8600h    ;Выделение битов C3, C2 и C1
        cmp     AH,0        ;Проверка наличия C3=0, C2=0, C1=0
        je      cont1
        ffree   ST(0)       ;Очистка ST
        fld     z
        fstp    b
        jmp     cont        ;z пока не найдено
cont1:  ffree   ST(0)
        fld     z
        fstp    a
        jmp     cont        ;z пока не найдено
;Вычисление z
ed:     fld     a
        fld     b
        fadd
        fdiv    dlt
        fst     z
;Поля данных
a       dd      0.0001
b       dd      1.0
z       dd      (?)
```

Сперва инициализируем сопроцессор с помощью команды finit. Затем последовательно вычислим значения F(a) и F(b). Для этого сначала перешлем из памяти значение переменной a в регистр ST и вычислим квадратный корень. Результат останется в ST. После этого в стек сопроцессора зашлем значение коэффициента p. Оно будет записано в вершину стека ST. К регистру, в котором находится уже вычисленный корень, мы теперь будем обращаться как к ячейке стека ST(1). Затем с помощью команды fmul (float multiply, умножение чисел в формате плавающей точки) без параметров перемножим содержимое регистров ST и ST(1). Значение регистра ST(1) после выполнения этой операции становится неопределенным, а произведение засылается в ST. После этого два раза засылаем величину a в стек. С помощью команды умножения вычислим квадрат a, который также запишем в стек. Далее с помощью команды сложения вычислим значение функции F(a).

Для вычисления F(b) используем такую же последовательность операций. Весь процесс обработки происходит аналогично, за исключением того, что в стеке сохраняется значение F(a).

После того как F(a) и F(b) вычислены, определяем их разность с помощью команды fsub (float substract, вычитание в формате плавающей точки) без явного указания операндов. Эта команда вычисляет ST(1)-ST, записывает результат в ST(1) и выполняет считывание из стека. Так как корень считается найденным, если модуль вычисленной разности меньше заданной величины delta, предварительно с помощью команды fabs (float absolute, абсолютное значение числа в формате плавающей точки), находим модуль ST. Для сравнения используем команду fcom (float compare, сравнение чисел в формате плавающей точки), которая из содержимого ST вычитает операнд, в данном случае delta. Результат операции не сохраняется, происходит лишь установка битов C0 и C2 в регистре состояния сопроцессора.

Для того, чтобы выполнить переход по результатам сравнения, необходимо использовать команды основного процессора, поскольку сопроцессор таких команд не имеет. Следовательно, необходимо передать результаты сравнения из сопроцессора в основной процессор. Для этого с помощью команды

```
fstsw sw
```

передадим содержимое регистра состояния в ячейку памяти sw, а затем проанализируем значения битов C0 и C2. Перед выполнением команды

```
mov   AX, sw
```

используем команду fwait для синхронизации работы основного процессора и сопроцессора.

После того, как регистр флагов центрального процессора установлен, проверим, найден ли диапазон, содержащий корень. И если этот диапазон найден, то переходим по метке ed на фрагмент, вычисляющий корень. Точность полученного решения будет равна |F(z)| .

В противоположном случае вычисляем точку z и выполняем переприсваивание: если знак F(z) совпадает со знаком F(a), то a заменяем на z, иначе на z заменяем b. Как и ранее, после проверки знака, для выполнения перехода передаем в сопроцессор содержимое слова состояния сопроцессора.

Статья 84

Процедура рисования окружности

Без использования сопроцессора практически невозможно обойтись в тех областях, где применяются процедуры машинной графики. Особенно это важно при разработке программ, реализующих режимы мультипликации и анимации. Если вычисление тригонометрических функций реализовывать на основном процессоре, то программа будет выполняться недопустимо медленно даже при выводе сравнительно несложных изображений. Применение сопроцессора здесь необходимо.

Рассмотрим программу, которая обеспечивает вывод на экран окружности заданного радиуса. Заметим, что сложность такой программы существенно зависит от того, с каким процессором мы работаем. Мы предполагаем, что в нашем распоряжении имеется процессор 80386 с сопроцессором или процессор i486DX. Если бы мы работали с компьютером, оснащенным сопроцессором более ранней версии, то программа оказалась бы сложнее. Дело в том, что только начиная с модели 80387 сопроцессоры обеспечивают выполнение команд вычисления sin() и cos().

Координаты точек окружности будем вычислять с шагом в один градус. Таким образом нам понадобится реализовать цикл по независимой переменной от 0 до 360 градусов. Заметим, что перед вычислением аргумент должен быть преобразован в радианы. Текст программы приведен в примере 84.1.

Пример 84.1. Вычисление координат точек, расположенных на окружности и вывод их на экран.

```
.386
data    segment
;Поля данных
x360    dd      180.0       ;Константа перевода градусы-радианы
x36     dw      360         ;Число точек на окружности
forcolordb      10          ;Салатовый цвет
;Координаты центра окружности
xc      dw      320
yc      dw      175
;Значения радиуса по осям
rx      dw      100
ry      dw      70
;Переменные
x       dw      ?           ;Текущие координаты точки окружности
y       dw      ?
angl    dw      1           ;Текущее значение угла
data    ends
text    segment use16
        assume CS:text,DS:data
;Подпрограмма изменения цвета пиксела
point   proc
        push    CX
        mov     CX,xc
        mov     AH,0Ch
        mov     AL,forcolor
        mov     BH,0
        fld     yc1
        fistp   yc
        mov     DX,yc
        fld     xc1
        fistp   xc
        mov     CX,xc
        sub     CX,x
        sub     DX,y
        int     10h
        pop     CX
        ret
point   endp
;Главная процедура
main    proc
;Подготовка данных
        mov     AX,data     ;Инициализация
        mov     DS,AX       ;регистра DS
        mov     AH,0h       ;Установка графического режима
        mov     AL,10h      ;Режим 10h
        int     10h         ;Прерывание BIOS
        mov     CX,x36      ;Число шагов построения окружности
        finit               ;Инициализация сопроцессора
        fldpi               ;Загрузка в стек числа pi
        fld     x360        ;Загрузка в стек числа 360,
        fdiv                ;pi/360, результат в ST
        fstp    x360        ;Сохранение в памяти коэффициента
                            ;перевода градусов в радианы
;Вычисление координат точек и вывод рисунка
```

```
do:     fld     x360        ;Коэффициент градус-радианы в стек
        fild    angl        ;Очередное значения утла в стек
        fmul                ;Перевод в радианы
        fsincos             ;sin(x) -> ST(1), cos(x) -> ST(0)
        fild    ry          ;Загрузка радиуса по координате y
        fmul                ;Вычисление координаты y
        fistp   y           ;Запись ее в память в формате целого
                            ;числа с извлечением из стека
        fild    rx          ;Загрузка радиуса по координате x
        fmul                ;Вычисление координаты x
        fistp   x           ;Запись ее в память в формате целого
                            ;числа с извлечением из стека
        fwait               ;Ожидание завершения работы сопроцессора
        call    point       ;Вывод точки на экран
        inc     angl        ;Приращение утла
        loop    do          ;Цикл
;Задержка до нажатия клавиши
        mov     AH,8
        int     21h
        mov     AX,4C00h    ;Выход в DOS с кодом ошибки 0
        int     21h
main    endp
text    ends
stk     segment stack 'stack'
        dw      128 dup(?)
stk     ends
end     main
```

Приведенная программа начинается с директивы .386, которая разрешает компиляцию всех команд процессоров 80386/87. Следует иметь в виду, что если в вашей программе необходимо применить какую-либо из команд арифметического процессора, а использование этой команды возможно только начиная с версии процессора 80x86, вы должны включить в программу директиву .x86. Поскольку при этом разрешается использование 32-разрядных регистров процессоров 386+, то необходимо определить размер сегмента. Мы это сделали, включив в описание сегмента команд атрибут use16.

Перед выполнением вычислений мы устанавливаем графический режим с помощью функции 0h прерывания 10h BIOS, инициализируем сопроцессор и вычисляем коэффициент для пересчета угла из градусов в радианы. Дело в том, что аргументы команд, вычисляющих тригонометрические функции, должны быть заданы в радианах, а приращение углов удобнее задавать в градусах. Для загрузки в стек числа pi можно использовать команду fldpi (float load pi, загрузка числа pi), одну из набора специальных команд, загружающих константы. Перечень этих команд приведен в приложении 4. После того, как число pi загружено в стек, загружаем из памяти эквивалентное значение в градусах и с помощью команды деления вычисляем коэффициент.

Вся остальная часть программы практически состоит из одного цикла, в котором производится вычисление значений синуса и косину-

са, определение координат точки окружности и изменение цвета пиксела с указанными координатами на экране. Заметим, что значение радиуса по горизонтальной оси отличается от значения радиуса по вертикальной. Это происходит из - за того, что в графических режимах разрешения по этим осям различны.

Полученную программу легко преобразовать для вывода других геометрических фигур, например спирали. В этом случае потребуется ввести в ее текст дополнительную подпрограмму, обеспечивающую непрерывное изменение радиуса. В результате у нас получится следующая программа (пример 84.2).

Пример 84.2. Фрагменты программы вычисления и вывода на экран спирали.

```
.386
;Сегмент данных
        ...
;Дополнительно введенные переменные
del     dd      0.9985      ;Коэффициент сжатия спирали
delx    dd      1.0000004   ;Коэффициент перемещения по оси x
dely    dd      0.999997    ;Коэффициент перемещения по оси y
forcolordb      9           ;Голубой цвет
rx      dd      100.0       ;У этих переменных изменен тип, не
ry      dd      70.0        ;забудьте изменить соответствующие
                            ;команды
xc1     dd      320.0       ;Дополнительные переменные для
yc1     dd      175.0       ;преобразования координат центра
;Сегмент команд
text    segment use16
        assume CS:text,DS:data
;Подпрограмма изменения координат центра и радиуса
coord   proc
        fld   rx
        fmul  del
        fstp  rx
        fld   ry
        fmul  del
        fstp  ry
        fild  xc1
        fmul  delx
        fistp xc1
        fild  yc1
        fmul  dely
        fistp yc1
        ret
coord   endp
;Подпрограмма изменения цвета пиксела (см. пример 84.1)
point   proc
        ...
point   endp
;Главная процедура
main    proc
;Подготовка данных
        ...
;Вывод точки на экран
```

```
        call   point
        inc    angl
;Изменение координат и радиуса
        call   coord
        loop   Do
;Задержка до нажатия клавиши
        mov    AH,8
        int    21h
        mov    AX,4C00h   ;Выход в DOS с кодом ошибки 0
        int    21h
main    endp
text    ends
stk     segment stack 'stack'
dw      128 dup (?)
stk     ends
end     main
```

В этой программе использована дополнительная подпрограмма coord, обеспечивающая непрерывное уменьшение размера радиуса и изменение координат центра окружности. Каждая из указанных выше величин записывается в стек сопроцессора. Затем она умножается на действительны коэффициент, после чего полученное значение снова записывается в память и выполняется чтение из стека, для того, чтобы избежать его переполнения.

Попробуйте самостоятельно составить программу, использующую вычисление тригонометрических функций, например, выводящую на экран график функции Y = Sin(a*x+b). Для этого вам надо будет воспользоваться командой fsin.

Статья 85

Управляющие регистры сопроцессора

Рассмотрим группу регистров, обеспечивающих управление работой сопроцессора. Эти регистры часто называют управляющими или нечисловыми регистрами; к ним относятся: управляющий регистр, регистр состояния, регистр признаков, указатель команды и указатель операнда (рис. 85.1).

Для работы с этими регистрами в сопроцессоре используется 30 команд, которые могут начинаться либо с двух букв fn, либо с одной буквы f, например, fnstcw и fstcw. Если такая команда начинается с

буквы f (float, обработка чисел с плавающей точкой), то перед тем, как передать ее на выполнение сопроцессору, центральный процессор проверяет, занят ли сопроцессор. Для этого проверяется бит занятости (B) регистра состояния, а также биты ошибок. Если команда начинается с букв fn (float no wait, обработка чисел с плавающей точкой без ожидания), то она передается на выполнение сопроцессору без каких либо предварительных проверок.

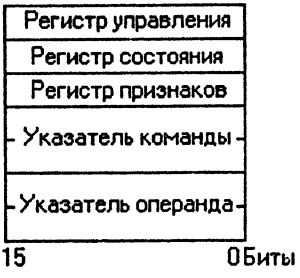


Рис. 85.1. Нечисловые регистры сопроцессора.

С одной из команд работы с нечисловыми регистров, именно, с командой инициализации сопроцессора finit, мы уже знакомы. Она устанавливает начальные значения в регистре состояния, управляющем регистре и регистре признаков. Регистр признаков, содержащий сведения о данных, находящихся в каждом из числовых регистров сопроцессора, был уже рассмотрен в статье 78. Команда finit засылает во все поля этого регистра значения 11b. Это означает, что все числовые регистры сопроцессора свободны, и в них можно записывать данные. Таким образом, команда finit может использоваться для очистки сразу всех числовых регистров.

Если же требуется очистить определенный числовой регистр, например, ST(i), то надо использовать команду

```
ffree ST(i)      ;float free, освобождение регистра
```

Эта команда записывает в i-ое поле регистра признаков 11b, тем самым помечая числовой регистр ST(i) как свободный.

Содержимое регистра состояния сопроцессоров 80287+ показано на рис. 85.2.

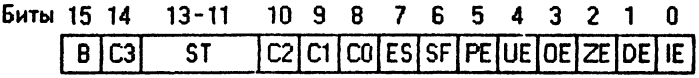


Рис. 85.2. Структура регистра состояния сопроцессора.

Биты с 0 по 5 представляют собой флаги особых случаев. Они устанавливаются при возникновении следующих ошибок:

IE (Invalid Operation) - недействительная операция;

DE (Denormalized Operand) - денормализованный операнд;

ZE (Zero Divide) - деление на нуль;

OE (Overflow) - переполнение;

UE (Underflow) - антипереполнение;

PE (Precision) - потеря точности.

Бит 6 содержит флаг стека SF (Stack Flag), а бит 7 слова состояния сопроцессоров 80287+ содержит флаг суммарной ошибки ES (Summary Error), который устанавливается при возникновении незамаскированного особого случая. В сопроцессоре 8087 этот бит содержит флаг IR (Interrupt Request) - запроса прерывания при возникновении незамаскированного особого случая.

Биты C0, C1, C2 и C3 (Condition Code)- коды условий. Они определяются по результату выполнения команд сравнения и команд нахождения остатка.

В поле ST (Stack Top Pointer) содержится номер числового регистра, являющегося вершиной стека. Бит занятости B (Busy) устанавливается, если сопроцессор выполняет команду или происходит прерывание от основного процессора. Если сопроцессор свободен, то бит занятости сбрасывается.

При инициализации сопроцессора все флаги, за исключением ST и ES, значения которых не определяются, сбрасываются.

Для тех, кто работает на IBM PC/XT, напомним еще раз, что регистр состояния сопроцессора 8087 отличается содержимым седьмого бита, в котором вместо флага ES (суммарная ошибка) хранится флаг IR (запрос прерывания).

Для того, чтобы получить информацию о содержимом полей регистра состояния или изменить их используются следующие команды:

fstsw (float store state word) и fnstsw mem - записать слово состояния сопроцессора в память;

fstsw AX и fnstsw AX - записать слово состояния сопроцессора в регистр AX (только для сопроцессоров 80287+);

fclex (float cleare exceptions) и fnclex - сбросить все флаги ошибок, а также биты ES и B;

fincstp (float increment stack pointer, увеличить указатель стека с плавающей точкой) - увеличить указатель вершины стека числовых регистров на 1;

fdecstp (float decrement stack pointer, уменьшить указатель стека с плавающей точкой) - уменьшить указатель вершины стека числовых регистров на 1.

Напомним, что, хотя содержимое полей кодов условий устанавливается при выполнении команд сопроцессора, выполнить переход можно только путем анализа установленного кода в основном процессоре. Для этого надо предварительно переписать в ячейку памяти содержимое слова состояния. Таким образом, если условием завершения итераций является выполнение неравенства $|p| < eps$, где p - текущий параметр, а eps - заданная величина, то проверку можно выполнить так:

Пример 85.1. Проверка условия выхода из цикла.

```
...
fld    p          ;Поместим p в стек сопроцессора
fabs              ;Вычислим модуль p
fcom   eps        ;Сравним с eps, результат в битах C0-C2
                  ;регистра состояния
fstsw  AX         ;Запишем регистр состояния
                  ;сопроцессора в регистр AX
and    AX,0700h   ;Выделим биты C0-C2
cmp    AH,1h      ;Если p<eps,то только C0=1. Проверим это
je     output     ;Если это так, выйдем из цикла
jmp    cont       ;Иначе продолжим вычисления
...
```

Структура регистра управления сопроцессоров 80287+, отдельные поля которого были нами рассмотрены ранее, показана на рис. 85.3.

Биты	15-13	12	11-10	9-8	7	6	5	4	3	2	1	0
	XXX	XX	RC	PC	XX	XX	PM	UM	OM	ZM	DM	IM

Рис. 85.3. Структура регистра управления сопроцессора.

Биты 0 - 5 для всех сопроцесоров являются масками недействительных состояний. Если бит маски для какого-либо недействительного состояния, например, переполнения - сброшен, то возникновение этого состояния вызывает прерывание центрального процессора. Если этот бит установлен, то прерывание не вырабатывается, а в качестве результата формируется особое значение (в рассматриваемом случае код бесконечности). Используются следующие маски особых случаев:

IM (Invalid Operation Mask) - маска недействительной операции;

DM (Denormalized Operand Mask) - маска денормализованного результата;

ZM (Zero Divide Mask) - маска деления на нуль;

OM (Overflow Mask) - маска переполнения;

UM (Underflow Mask) - маска антипереполнения;

PM (Precision Mask) - маска особого случая при неточном результате.

В сопроцессорах 80287 и выше поля в битах 6 и 7 не используются. В сопроцессоре 8087 в поле 7 содержится маска недействительных прерываний IEM (Interrupt Enable Mask). Если IEM = 0, то прерывания центрального процессора не будет даже при возникновении незамаскированной ошибки.

Содержимое поля PC (Precizion Control, биты 8 и 9) определяет точность вычислений в сопроцессоре:

11b - используется расширенная точность;

10b - результат округляется до двойной точности;

00b - результат округляется до одинарой точности.

Двухбайтовое поле RC (rounding control, биты 10 и 11) определяет режим округления при выполнении операций с вещественными числами:

00b - производится округление к ближайшему числу. Этот режим устанавливается при инициализации сопроцессора;

01b - производится округление в направлении к отрицательной бесконечности;

10b - производится округление в направлении к положительной бесконечности;

11b - производится округление в направлении к нулю.

В сопроцессорах 8087 и 80287 содержимое бита 12 предназначено для управления трактовкой понятия бесконечности; этот бит называется IC (infinity control). Данные сопроцессоры могут работать в двух режимах: проективном (IC=0) и аффинном (IC=1). В проективном режиме существует только одна бесконечность, которая не имеет знака. В аффинном режиме определено две бесконечности: положительная и отрицательная. В этом режиме допускается выполнение арифметических операций с бесконечностями. Сопроцессоры 80387+ работают только в режиме аффинной арифметики.

После выполнения команды finit в регистре управления устанавливается режим работы с расширенной точностью и округления к ближайшему числу. Все биты масок обработки особых случаев устанавливаются в 1, следовательно, все особые случаи будут замаскированы.

Для работы с регистром управления используются следующие команды сопроцессора:

fstcw (fnstcw) mem - записать содержимое управляющего регистра сопроцессора в ячейку памяти mem;

fldcw mem - записать содержимое ячейки памяти mem в управляющий регистр сопроцессора;

Рассмотрим, каким образом можно определить и при необходимости изменить установленный режим округления. Фрагмент программы, осуществляющей вывод на экран информации об установленном режиме округления приведен в примере 85.2.

Пример 85.2. Фрагмент программы, анализирующий содержимое поля RC регистра управления сопроцессора.

```
.386
data segment
;Выводимые сообщения
mes00    db     'Округление до ближайшего целого$'
mes01    db     'Округление в сторону уменьшения$'
mes10    db     'Округление в сторону увеличения$'
mes11    db     'Отбрасывание разрядов$'
```

```
;Резервируем слово для хранения регистра управления
clwd    dw    ?
        ...
data    ends
text    segment use16
assume  CS:text,DS:data
;Подпрограмма вывода сообщения
outt    proc
        mov   AH,09h
        int   21h
        ret
outt    endp
;Основная программа
main    proc
        mov   AX,data
        mov   DS,AX
        ...
;Проверка установленного режима округления
        fstcw clwd        ;Запишем содержимое регистра
                          ;управления в ячейку clwd
        mov   AX,clwd     ;Перенесем содержимое clwd в AX
        and   AX,0C00h    ;Выделим биты, управляющие режимом
                          ;округления и для удобства сравнения
        shr   AH,2        ;сдвинем AH вправо на два бита
;Определяем, какой режим и выводим соответствующее сообщение
        cmp   ah,0h
        jg    m01
        mov   DX,offset mes00
        jmp   ot
m01:    cmp   ah,1h
        jg    m02
        mov   DX,offset mes01
        jmp   ot
m02:    cmp   ah,2h
        jg    m03
        mov   DX,offset mes10
        jmp   ot
m03:    mov   DX,offset mes11
ot:     call  outt
        ...
main    endp
text    ends
end     main
```

Для того, чтобы установить необходимый режим округления, достаточно считать текущее слово управления, изменить необходимым образом содержимое поля RC и затем записать новое слово в регистр управления.

С помощью отладчика Turbodebugger (см. статью 79) можно просматривать и изменять флаги регистра состояния и маски регистра управления непосредственно в процессе отладки программы. Вся эта информация, пример которой приведен на рис. 85.4, выводится в правой части окна сопроцессора.

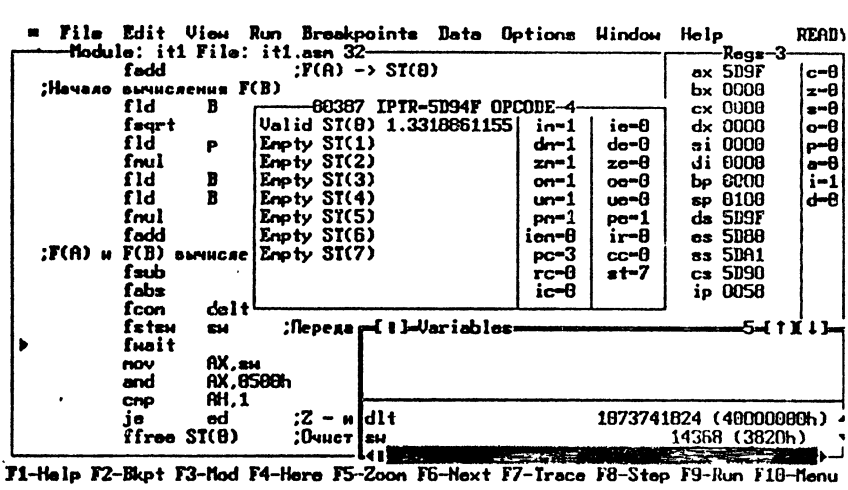


Рис. 85.4. Вывод информации о содержимом регистров состояния и управления при отладке программы.

Приведенные здесь флаги и поля битов нам уже знакомы, отметим только, что содержимое всех кодов условий представлено одним параметром CC (Condition Code). Для изменения флагов и битов масок достаточно при активном окне сопроцессора выбрать необходимый флаг, войти в меню окна и переключить выбранный флаг. Некоторые поля принимают более двух значений. Для них выполняется операция увеличения на единицу. Если перед выполнением операции поле имело максимальное значение, то оно обнулится.

Последние два регистра - указатель команды и указатель операнда (рис. 85.5).

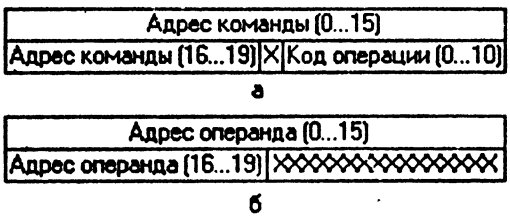


Рис. 85.5. Структура регистров указателя команды (а) и указателя операнда (б).

Указатель команды содержит двадцатиразрядный адрес команды, вызвавшей особый случай, а также код выполнявшейся операции. Если при возникновении особого случая использовался операнд, его адрес записывается в регистр указателя операнда. Приведенные форматы соответствуют 16-разрядному реальному режиму работы процессора (при определении сегмента команд используется атрибут use16).

Следующая группа команд позволяет оперировать со всеми регистрами управления сразу:

fstenv mem (float store environment) - записать содержимое всех нечисловых регистров сопроцессора в память;

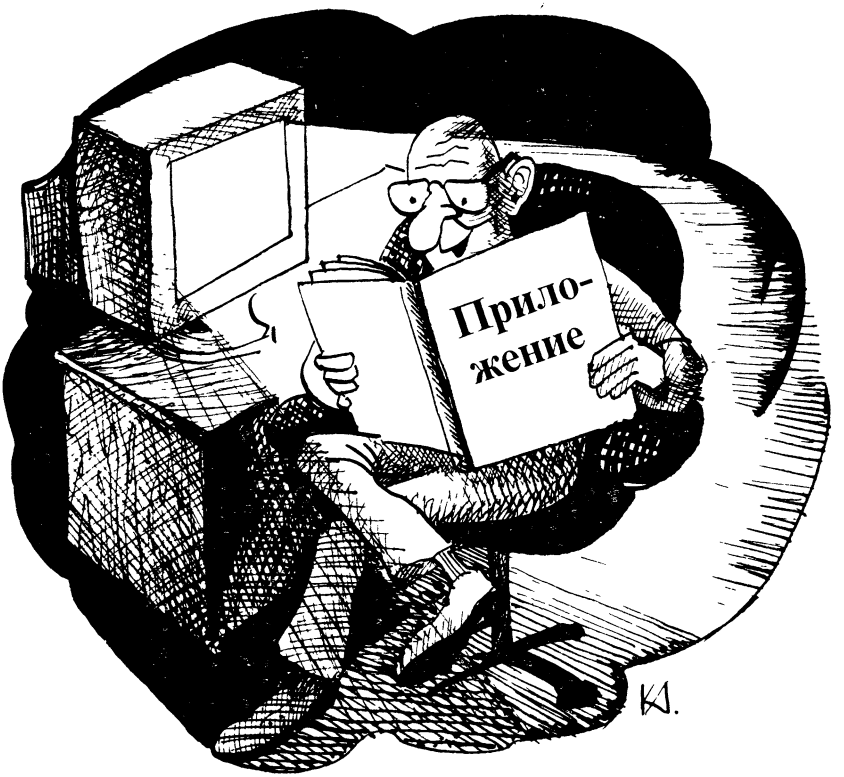
fldenv mem (float load environment) - восстановить содержимое всех нечисловых регистров сопроцессора из памяти;

При работе в 16-разрядном реальном режиме область mem имеет формат, показанный на рисунке 85.1.

Если возникает необходимость сохранить содержимое всех регистров, как числовых, так и нечисловых, то для этой цели можно использовать команду fsave all (float save, сохранить среду сопроцессора). Для восстановления ранее запомненного состояния используется команда frstor all (float store, восстановить среду сопроцессора). При работе в 16-разрядном режиме all представляет собой 94-байтное поле памяти, имеющее структуру, показанную на рис. 85.6.

Рис. 85.6. Структура поля памяти, предназначенного для сохранения всех регистров сопроцессора в 16-разрядном режиме работы.

Приложения



Приложение 1

Основные 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.

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).

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.

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.

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 операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда еще и непосредственное значение, однако не допуска-

ется определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака.

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 операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда еще и 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.

AND Логическое И

Команда AND осуществляет логическое (побитовое) умножение первого операнда на второй. Исходное значение первого операнда (приемника) теряется, замещаясь результатом умножения. В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами.

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 смещение точки возврата в текущем программном сегменте и модифицирует IP так, чтобы в нем содержалось смещение точки перехода в том же программном сегменте. Необходимое для вычисления этого смещения расстояние 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) смещение точки возврата в текущем программном сегменте. Далее модифицируются регистры IP и CS:

в IP помещается смещение 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.

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.

CLC Сброс флага переноса

Команда CLC сбрасывает флаг переноса CF в регистре флагов.

CLD Сброс флага направления

Команда CLD сбрасывает флаг DF в регистре флагов, устанавливая прямое (в порядке возрастания адресов) направление выполнения операций со строками.

CLI Сброс флага прерываний

Команда CLI сбрасывает флаг IF в регистре флагов, запрещая все аппаратные прерывания. Прерывания будут оставаться запрещенными до установки флага IF командой sti. На программные прерывания (команду int) флаг не действует.

CMC Инвертирование флага переноса

Команда CMC изменяет значение флага CF в регистре флагов на обратное.

CMP 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 операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или слова-

ми и представлять числа со знаком или без знака. Обычно вслед за командой CMP стоит одна из команд условных переходов, анализирующих состояние флагов процессора (je - переход, если равно, jne - переход, если не равно и т.д.).

CMPS 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 содержимым). Они сравнивают по одному элементу каждой строки, фактически осуществляя 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. 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/REPNZ (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 (если DF=1) элементами строк, на которых закончились операции сравнения.

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.

DAA Десятичная коррекция в регистре AL после сложения

Команда DAA корректирует результат сложения в регистре 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.

DAS Десятичная коррекция в регистре AL после вычитания

Команда DAS корректирует результат вычитания в регистре 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.

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.

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 через вектор 0.

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.

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.

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.

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.

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). Команда INT сбрасывает флаг IF.

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 4 двухсловный адрес своего обработчика прерывания по переполнению.

IRET Возврат из программы обработки прерывания

Команда IRET возвращает управление прерванному 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 условных переходов осуществляют переход по указанному адресу при выполнении условия, заданного мнемоникой команды. Если заданное условие не выполняется, переход не осуществляется, а выполняется команда, следующая за конкретной командой условного перехода. Переход может осуществляться как вперед, так и назад в диапазоне +127...-128 байтов.

Команды условных переходов перечислены в табл. П1.1.

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 сегмента команд);
- переход прямой дальний (в другой сегмент команд);
- 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.

Таблица П1.1. Команды условных переходов и их действие

Команда	Перейти, если	Условие 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
JCXZ	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
JNG	не больше	ZF=1 или SF не равно OF
JNGE	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 нуль	ZF=0
JO	переполнение	OF=1
JP	есть четность	PF=1
JPE	сумма битов четная	PF=1
JPO	сумма битов нечетная	PF=0
JS	знаковый бит равен 1	SF=1
JZ	нуль	ZF=1

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.

LDS Загрузка указателя с использованием регистра DS.

Команда LDS считывает из памяти по указанному адресу двойное слово, 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

```
lds    reg,mem
```

эквивалентна по результату 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.

LEA Загрузка исполнительного адреса

Команда LEA загружает в регистр, указанный в команде в качестве первого операнда, относительный адрес второго операнда. В качестве первого операнда следует указывать регистр общего назначения, в качестве второго - ячейку памяти. Команда

```
lea   reg,mem
```

по своему результату эквивалентна команде

```
mov   reg,offset mem
```

LES Загрузка указателя с использованием регистра ES

Команда LES считывает из памяти по указанному адресу двойное слово, 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 по результату следующей паре 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.

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 содержимым). 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

```
lods   строка
```

(что не избавляет от необходимости инициализировать регистры DS:SI адресом строки; операнд лишь позволяет ассемблеру определить по описанию поля данных *строка* размерность загружаемых данных - байт или слово). В этом формате возможна замена сегмента строки *строка*:

```
lods   ES:строка
```

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 сегменте команд 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).

LOOPE Цикл, пока равно

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.

LOOPNE Цикл, пока не равно

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 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.

LOOPNZ Цикл, пока не нуль

Команда выполняет те же действия, что и LOOPNE.

LOOPZ Цикл, пока нуль

Команда выполняет те же действия, что и LOOPE.

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 атрибутных операторов byte ptr или word ptr. В зависимости от использованных режимов адресации команда MOV осуществляет 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;
- 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. Нельзя также непосредственно переслать содержимое одного сегментного регистра в другой.

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 содержимым). 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*; операнды лишь позволяют ассемблеру определить по описанию полей данных *строка_1* и *строка_2* размерность пересылаемых данных - байт или слово). 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.

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.

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. Нельзя использовать в качестве операнда непосредственное значение.

NOP Холостая 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 байт или слово. Нельзя использовать в качестве операнда непосредственное значение.

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 операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами.

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.

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.

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.

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.

PUSHF Занесение в стек содержимого регистра флагов

Команда PUSH уменьшает на 2 содержимое указателя стека SP и заносит на эту новую вершину содержимое регистра флагов.

RCL Циклический сдвиг влево через бит переноса

Команда RCL осуществляет сдвиг влево всех бит операнда. Если команда записана в формате

```
RCL    операнд, 1
```

выполняется сдвиг 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. В качестве операнда нельзя использовать непосредственное значение.

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. В качестве операнда нельзя использовать непосредственное значение.

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 сегмента команд. 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 будет межсегментной.

В команду RET может быть включен необязательный операнд (кратный 2), который указывает, на сколько байтов дополнительно смещается (в сторону больших адресов) указатель стека после возврата в вызывающую программу. Прибавляя эту константу к новому значению SP, команда RET обходит аргументы, помещенные в стек вызывающей программой перед вызовом команды CALL.

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.

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. Нельзя использовать в качестве операнда непосредственное значение.

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.

SAL Арифметический сдвиг влево

Команда SAL осуществляет сдвиг влево всех битов операнда. Старший бит операнда поступает в флаг CF. Если команда имет форму

```
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. Нельзя 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.

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. Нельзя 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.

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

```
sbb    операнд_1, операнд_2
```

то ее действие можно условно изобразить следующим образом:

```
операнд_1 - операнд_2 - CF -> операнд_1
```

В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда еще и 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.

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 содержимым). 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*; операнд лишь позволяет ассемблеру определить по описанию поля данных *строка* размерность сравниваемых данных - байт или слово). Замена сегментного регистра (ES), через который адресуется строка, невозможна.

Рассматриваемые команды могут предваряться префиксами повторения REPE/REPZ (повторять, пока элементы равны, т.е. до первого неравенства) и REPNE/REPNZ (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 регистр DI указывает на ячейку памяти, находящуюся за тем (если DF=0) или перед тем (если DF=1) элементом строки, на котором закончились операции сравнения.

SHL Логический сдвиг влево

Команда SHL выполняет те же действия, что и SAL.

SHR Логический сдвиг вправо

Команда SHR осуществляет сдвиг вправо всех бит операнда. Младший бит операнда поступает в флаг CF. Если команда имеет форму

```
SHR     операнд, 1
```

сдвиг осуществляется на 1 бит. В старший бит операнда загружается 0, а младший теряется. Если команда имеет форму

```
SHR     операнд, 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. Нельзя использовать в качестве операнда непосредственное значение.

STC Установка флага переноса

Команда STC устанавливает флаг переноса CF в регистре флагов.

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).

STI Установка флага прерывания

Команда STI устанавливает флаг IF в регистре флагов, разрешая все аппаратные прерывания.

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 содержимым). 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*; операнд лишь позволяет ассемблеру определить по описанию поля данных *строка* размерность записываемых данных - байт или слово). Заменить сегментный регистр ES нельзя.

Рассматриваемые команды могут предваряться префиксом повторения REP. В этом случае они повторяются CX раз, заполняя последовательные ячейки памяти одним и тем же содержимым регистра AL или AX.

SUB Вычитание целых чисел

Команда SUB вычитает второй операнд (источник) из первого (приемника) и помещает результат на место первого операнда. Ис-

ходное значение первого операнда (уменьшаемое) теряется. Таким образом, если команду вычитания записать в общем виде

```
sub    операнд_1, операнд_2
```

то ее действие можно условно изобразить следующим образом:

```
операнд_1 - операнд_2 -> операнд_1
```

В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго операнда еще и 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.

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 операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда еще и 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.

XCHG Обмен данными между операндами

Команда XCHG пересылает значение первого операнда во второй, а второго - в первый. В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами со знаком или без знака.

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, замещая находившееся в нем смещение. Длина таблицы может достигать 256 байтов. Команда XLAT не имеет явных операндов, но требует предварительной настройки регистров BX и AL.

XOR Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ

Команда XOR выполняет операцию логического (побитового) ИСКЛЮЧАЮЩЕГО ИЛИ над 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.

Приложение 2

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> - выход из меню.

Функциональные клавиши

<F1> - вывод справочника.

<F2> - вывод на экран информационного поля с содержимым 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.

D*type 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. 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.

D*type seg:addr1 addr* - дамп памяти от адреса *seg:addr1* до адреса *seg:addr2*.

R *reg* - вывод содержимого регистра *reg* и запрос на его изменение.

R *reg*=*n* - занесение в регистр *reg* значения *n*.

E*type seg:addr n1 n2* ... - занесение в память начиная с адреса *seg:addr* значений *n1, n2* ... в формате *type*. Возможные значения *type* приведены в описании команды D. По умолчанию *seg*=DS. Между командой и типом не должно быть пробела.

N*radix* - изменение системы счисления в параметрах командной строки. Параметр *radix* 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 ее 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*.

Приложение 3

Команды сопроцессора

F2XM1 Вычисление 2x-1

Команда не требует операндов; она выполняет вычисления по формуле 2x-1. Переменная x может принимать значения от 0 до 0,5 и перед выполнением команды должна быть помещена в вершину стека. Результат помещается в вершину стека, причем исходное значение x не сохраняется.

Флаги, на которые влияет команда: PE, UE, DE.

Пример

```
;Сегмент команд
        fld     x               ;ST=0.1
        f2xm1                   ;ST=20.1-1=0.07177...
;Сегмент данных
x       dd      0.1
```

FABS Абсолютное значение числа

Команда не требует операндов; она вычисляет абсолютное значение числа, маскируя разряд знака числа из ST.

Флаги, на которые влияет команда: IE.

Пример

```
;Сегмент команд
        fld     x               ;ST=-1.525e34
        fabs                    ;ST=1.525e34
;Сегмент данных
x       dd      -1.525e34
```

FADD Сложение двух действительных чисел

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда выполняет их сложение и засылает результат в первый операнд. Операндами могут быть только регистры стека.

```
        fadd    ST,ST(i)    ;ST=ST(i)+ST
        fadd    ST(i),ST    ;ST(i)=ST(i)+ST
```

Если указан один операнд, то он суммируется с содержимым вершины стека, результат помещается в вершину стека. Операндом может быть только переменная, описанная как двойное или четверное слово.

```
        fadd    mem             ;ST=ST+mem
```

При отсутствии операндов складывается содержимое вершины стека ST и находящегося под ней регистра ST(1). Результат записывается в регистр ST(1). После этого выполняется операция выталкивания из стека.

```
        fadd                    ;ST(1)=ST(1)+ST, выталкивание
```

Флаги, на которые влияет команда: PE, UE, OE, DE, IE.

Пример 1

```
        fadd    ST,ST(5)        ;ST=ST+ST(5)
```

Пример 2

```
;Сегмент команд                  ST      ST(1)   ST(2)   ST(3)
        fld     x1              ;4.8     ?       ?       ?
        fld     x2              ;1.4     4.8     ?       ?
        fld     x3              ;5.2     1.4     4.8     ?
        fadd    ST(2),ST        ;5.2     1.4     10.0    ?
;Сегмент данных
x1      dd      4.8
x2      dd      1.4
x3      dd      5.2
```

Пример 3

```
        fadd    ST,ST           ;ST=ST*2
```

Пример 4

```
;Сегмент команд
        fadd    x1              ;ST=ST+x1
;Сегмент данных
x1      dd      3.2e38          ;Двойное слово
```

Пример 5

```
;Сегмент команд                  ST      ST(1)
        fld     a12             ;0.4e308  ?
        fadd    dat             ;1.7e308  ?
;Сегмент данных
a12     dq      0.4e308         ;Четверное слово
dat     dq      1.3e308         ;Четверное слово
```

Пример 6

```
·Сегмент команд                  ST      ST(1)   ST(2)
        fld     x1              ;41.8     ?       ?
        fld     x2              ;12.4    41.8     ?
        fadd                    ;54.2     ?       ?
;Сегмент данных
x1      dd      41.8
x2      dd      12.4
```

FADDP Сложение двух действительных чисел и запись результата в любой регистр стека с последующим выполнением операции выталкивания из стека

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда выполняет их сложение и засылает результат в первый операнд. Затем производится выталкивание из стека. Операндами могут быть только регистры стека. В качестве второго операнда может быть использован только регистр ST.

```
faddp  ST(i),ST   ;ST(i)=ST(i)+ST, выталкивание
```

Если указан один операнд, то он суммируется с содержимым вершины стека, результат помещается в операнд. Затем производится выталкивание из стека. Операндом может быть любой регистр стека. Эта команда имеет тот же код, что и команда fadd ST(i),ST, она просто является другой формой ее записи.

```
faddp  ST(i)      ;ST(i)=ST(i)+ST,выталкивание
```

При отсутствии операндов складывается содержимое вершины стека ST и находящейся под ней ячейки ST(1). Результат записывается в регистр ST(1). Затем производится выталкивание из стека. Данная команда полностью аналогична команде fadd без операндов.

```
faddp             ;ST(1)=ST+ST(1),
```

Флаги, на которые влияет команда: PE, UE, OE, DE, IE.

Пример 1

```
faddp  ST(4),ST   ;ST(4)=ST(4)+ST, выталкивание
```

Пример 2

```
;Сегмент команд          ST    ST(1)  ST(2)  ST(3)
        fld    x1       ;4.8    ?      ?      ?
        fld    x2       ;1.4   4.8     ?      ?
        fld    x3       ;5.2   1.4    4.8     ?
        faddp  ST(2),ST  ;1.4   10.0    ?      ?
;Сегмент данных
x1      dd     4.8
x2      dd     1.4
x3      dd     5.2
```

Пример 3

```
faddp  ST(3)      ;ST(3)=ST(3)+ST, выталкивание
```

Данная команда аналогична команде faddp ST(3),ST

Пример 4

```
faddp  ST         ;ST=ST+ST. Поскольку результат
                  ;выталкивается из стека и теряется,
                  ;команда не имеет смысла
```

FBLD Преобразование операнда из двоично-десятичного кода в формат действительных чисел и загрузка в стек

Команда использует один операнд. Она преобразует упакованный двоично-десятичный операнд в действительный, загружая результат в стек. Знак сохраняется.

Флаги, на которые влияет команда: IE.

Пример

```
;Сегмент данных
tn      dt      999999999999999999
;Сегмент команд
        fbld   tn          ;ST=999999999999999999
```

FBSTP Преобразование числа из вершины стека в упакованный десятичный формат и выталкивание из стека

Команда использует один операнд. Число, находящееся в вершине стека, округляется. Полученная целая часть преобразуется в упакованный десятичный формат. Результат запоминается в ячейке памяти, адрес которой определяется операндом программы. После этого выполняется выталкивание из стека.

Флаги, на которые влияет команда: IE.

Пример

```
;Сегмент данных
ddf     dt      ?
ff      dw      99h
;Сегмент команд
        fild   ff          ;ST=153
        fbstp  ddf         ;ST=?, ddf=153
```

FCHS Инвертирование знака числа, находящегося в вершине стека

Команда не требует операндов. Она изменяет знак содержимого вершины стека.

Флаги, на которые влияет команда: IE.

Пример

```
;Сегмент данных
m1      dd      0.015
;Сегмент команд
        fld    m1          ;ST=0.015
        fchs               ;ST=-0.015
```

FCLEX Очистка флагов исключительных ситуаций сопроцессора

Команда fclex и аналогичная ей команда fnclex не требуют операндов. Они очищают все флаги исключительных ситуаций, а также поле занятости и поле запроса прерывания в слове состояния сопроцессора. Команда fclex перед очисткой выполняет, а команда fnclex не выполняет проверку условий ошибки с плавающей точкой.

FCOM Сравнение двух действительных чисел

Имеются два варианта использования команды fcom и схожей с ней команды fcomp: без операндов и с одним операндом.

При использовании операнда выполняется его сравнение с содержимым вершины стека. Для этого он вычитается из содержимого вершины стека и по результатам устанавливаются разряды C0, C2 и C3 слова состояния сопроцессора (табл. П3.1).

Таблица П3.1. Значения разрядов C0, C2 и C3 слова состояния сопроцессора после выполнения команды сравнения.

Результат операции	Разряды слова состояния C3	C2	C0
ST не сравнимо с операндом	1	1	1
ST равно операнду	1	0	0
ST меньше операнда	0	0	1
ST больше операнда	0	0	0

Операндом может быть любой регистр стека сопроцессора или переменная, описанная как двойное или четверное слово.

```
fcom   mem          ;ST-mem
fcom   ST(i)        ;ST-ST(i)
```

Если операнды отсутствуют, то сравнивается содержимое регистров ST и ST(1).

```
fcom                ;ST-ST(1)
```

При использовании команды fcomp после сравнения выполняется операция выталкивания из стека.

```
fcomp  mem          ;ST-mem, выталкивание
fcomp  ST(i)        ;ST-ST(i), выталкивание
fcomp               ;ST-ST(1), выталкивание
```

Флаги, на которые влияют команды: DE, IE.

Пример 1

```
;Сегмент данных
dlt     dd     0.5e-8
;Сегмент команд
        fcom   dlt          ;Выполняется операция ST-dlt. По ее
                            ;результатам устанавливаются C0, C2
                            ;и C3.Содержимое стека не изменяется
```

Пример 2

```
;Сегмент данных
fl      dq     0.5e-100
xx      dd     0.06
;Сегмент команд
        fld    xx           ;ST=0.06
        fcom   fl           ;Выполняется операция ST-fl
                            ;C3=0, C2=0, C0=0; ST=0.06
```

Пример 3

```
;Сегмент данных
xe      dd     100.0
xt      dd     19.8
;Сегмент команд
        fld    xt           ;ST=19.8, ST(1)=?
        fcomp  xe           ;Выполняется операция ST-xe
                            ;C3=0, C2=0, C0=1; ST=?
```

Пример 4

```
;Сегмент данных
xe      dd      100.0
xt      dd      19.8
;Сегмент команд          ST      ST(1)   ST(2)
        fld     xe      ;100.0    ?       ?
        fld     xt      ;19.8   100.0     ?
        fcom            ;19.8   100.0     ? C3=0, C2=0, C0=1
```

Пример 5

```
;Сегмент данных
xe      dd      100.0
xt      dd      100.0
;Сегмент команд          ST      ST(1)   ST(2)
        fld     xe      ;100.0    ?       ?
        fld     xt      ;100.0  100.0     ?
        fcomp           ;100.0    ?       ? C3=1, C2=0, C0=0;
```

FCOMP Сравнение двух действительных чисел и выполнение операции выталкивания из стека

См. описание команды fcom.

FCOMPP Сравнение двух действительных чисел и выталкивание их из стека

Команда не требует операндов. Она сравнивает содержимое вершины стека ST с содержимым следующего регистра ST(1). Для этого выполняется операция ST-ST(1) и по результатам вычитания устанавливаются разряды C0, C2 и C3 слова состояния сопроцессора (табл. П3.2). После сравнения два раза выполняется операция чтения из стека, то есть из стека выталкиваются оба сравниваемых числа.

```
        fcompp          ;Выполняется операция ST-ST(1),
                        ;выталкивание, выталкивание
```

Флаги, на которые влияет команда: DE, IE.

Таблица П3.2. Значения разрядов C0, C2 и C3 слова состояния сопроцессора после выполнения команды fcompp.

Результат операции	Разряды слова состояния C3	C2	C0
ST не сравнимо с ST(1)	1	1	1
ST равно ST(1)	1	0	0
ST меньше ST(1)	0	0	1
ST больше ST(1)	0	0	0

Пример 1

```
;Сегмент данных
xe      dd      100.0
xt      dd      100.0
```

```
;Сегмент команд          ST      ST(1)    ST(2)
        fld    xe       ;100.0    ?        ?
        fld    xt       ;100.0   100.0     ?
        fcompp          ;  ?      ?        ? C3=1, C2=0, C0=0;
```

FCOS Вычисление косинуса (только начиная с процессора 80387).

Команда не требует операндов. Она вычисляет косинус действительного числа, расположенного в вершине стека, при этом предполагается, что аргумент задан в радианах. Результат выполнения команды помещается в стек на место операнда. Следует иметь в виду, что модуль величины угла не должен превышать значения 2^{63}, причем следить за этим должен сам программист. Если значение аргумента выходит из заданного диапазона, то после выполнения команды ST не изменится и установится флаг C2 слова состояния.

Флаги, на которые влияет команда: PE, DE, IE.

Пример

```
                        ;ST        ST(1)
        fldpi           ;3.1416...   ?
        fcos            ;-1.0        ?
```

FDECSTP Уменьшение указателя стека

Команда не требует операндов. После ее выполнения указатель стека уменьшается на 1, то есть все регистры проталкиваются в стек. При этом в вершину стека заносится содержимое регистра ST(7).

Команда не изменяет флаги.

Пример

```
        fdecstp
```

Результат выполнения команды проиллюстрирован на рис. П3.1.

Рис. П3.1. Результат выполнения команды fdecstp.

FDIV Деление двух действительных чисел

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда делит содержимое первого операнда на содержимое второго и засылает результат в первый операнд. Операндами могут быть только регистры стека.

```
fdiv   ST,ST(i)   ;ST=ST/ST(i)
fdiv   ST(i),ST   ;ST(i)=ST(i)/ST
```

Если указан один операнд, то на него делится содержимое вершины стека и результат помещается в вершину стека. Операндом может быть только переменная, описанная как двойное либо четвертное слово.

```
fdiv   mem        ;ST=ST/mem
```

При отсутствии операндов содержимое регистра ST(1) делится на содержимое вершины стека ST. Результат записывается в регистр ST(1) и затем выполняется операция выталкивания из стека.

```
fdiv              ;ST(1)=ST(1)/ST, выталкивание
```

Флаги, на которые влияет команда: PE, UE, OE, ZE, DE, IE.

Пример 1

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld    x1           ; 4.0    ?       ?       ?
        fld    x2           ; 6.3   4.0      ?       ?
        fld    x3           ;10.0   6.3     4.0      ?
        fdiv   ST,ST(2)     ; 2.5   6.3     4.0      ?
;Сегмент данных
x1      dd     4.0
x2      dd     6.3
x3      dd     10.0
```

Пример 2

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld    x1           ; 4.0    ?       ?       ?
        fld    x2           ; 6.3   4.0      ?       ?
        fld    x3           ;10.0   6.3     4.0      ?
        fdiv   ST(2),ST     ;10.0   6.3     0.4      ?
;Сегмент данных
x1      dd     4.0
x2      dd     6.3
x3      dd     10.0
```

Пример 3

```
;Сегмент команд              ST     ST(1)   ST(2)
        fld    x1           ;11.5    ?       ?
        fld    x2           ; 6.3   11.5     ?
        fdiv   ST,ST        ; 1.0   11.5     ?
;Сегмент данных
x1      dd     11.5
x2      dd     6.3
```

Пример 4

```
;Сегмент команд              ST      ST(1)
        fld    a1           ;4.0e307  ?
        fdiv   dlm          ;20.0     ?
;Сегмент данных
a1      dq     0.4e308      ;Четверное слово
dlm     dq     2.0e306      ;Четверное слово
```

Пример 5

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld    x1           ;6.0     ?       ?       ?
        fld    x2           ;2.0    6.0      ?       ?
        fld    x3           ;4.0    2.0     6.0      ?
        fdiv                ;0.5    6.0      ?       ?
;Сегмент данных
x1      dd     6.0
x2      dd     2.0
x3      dd     4.0
```

FDIVP Деление двух действительных чисел с последующим выталкиванием из стека

Существуют три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда делит содержимое первого операнда на содержимое второго и засылает результат в первый операнд. Затем производится выталкивание из стека. Операндами могут быть только регистры стека. В качестве первого операнда запрещается использовать регистр ST.

```
fdivp  ST(i),ST   ;ST(i)=ST(i)/ST, выталкивание
```

Если указан один операнд, то он делится на содержимое вершины стека, результат помещается в этот операнд. Затем производится выталкивание из стека. Операндом может быть только регистр стека. Допускается использовать регистр ST.

```
fdivp  ST(i)      ;ST(i)=ST(i)/ST, выталкивание
```

При отсутствии операндов содержимое регистра ST(1) делится на содержимое вершины стека ST. Результат записывается в регистр ST(1). Затем производится выталкивание из стека. Данная команда аналогична команде fdiv.

```
fdivp             ;ST(1)=ST(1)/ST, выталкивание
```

Флаги, на которые влияет команда: PE, UE, OE, ZE, DE, IE.

Пример 1

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld    x1           ; 4.0    ?       ?       ?
        fld    x2           ; 1.5   4.0      ?       ?
        fld    x3           ;10.0   1.5     4.0      ?
        fdivp  ST(2),ST     ; 1.5   0.4      ?       ?
```

```
;Сегмент данных
x1      dd      4.0
x2      dd      1.5
x3      dd      10.0
```

Пример 2

```
;Сегмент команд              ST     ST(1)   ST(2)
        fld     f           ; 4.0    ?       ?
        fld     s           ;10.0   4.0      ?
        fdivp   ST(1)       ; 0.4            ?
;Сегмент данных
f       dd      4.0
s       dd      10.0
```

Пример 3

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ;6.0     ?       ?       ?
        fld     x2          ;2.0    6.0      ?       ?
        fld     x3          ;4.0    2.0     6.0      ?
        fdivp               ;0.5    6.0      ?       ?
;Сегмент данных
x1      dd      6.0
x2      dd      2.0
x3      dd      4.0
```

FDIVR Деление двух действительных чисел в обратном порядке

Данная команда идентична команде fdiv, но делимое и делитель меняются местами. Варианты использования операндов показаны ниже.

```
        fdivr   ST,ST(i)    ;ST=ST(i)/ST
        fdivr   ST(i),ST    ;ST(i)=ST/ST(i)
        fdivr   mem         ;ST=mem/ST
        fdivr               ;ST(1)=ST/ST(1), выталкивание
```

Флаги, на которые влияет команда: PE, UE, OE, ZE, DE, IE.

Пример 1

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ; 4.0    ?       ?       ?
        fld     x2          ; 6.3   4.0      ?       ?
        fld     x3          ;10.0   6.3     4.0      ?
        fdivr   ST,ST(2)    ; 0.4   6.3     4.0      ?
;Сегмент данных
x1      dd      4.0
x2      dd      6.3
x3      dd      10.0
```

Пример 2

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ; 4.0    ?       ?       ?
        fld     x2          ; 6.3   4.0      ?       ?
        fld     x3          ;10.0   6.3     4.0      ?
        fdivr   ST(2),ST    ;10.0   6.3     2.5      ?
;Сегмент данных
x1      dd      4.0
x2      dd      6.3
x3      dd      10.0
```

Пример 3

```
;Сегмент команд              ST        ST(1)
        fld     eps          ;2.6e-38     ?
        fdivr   del          ;19999.9...  ?
;Сегмент данных
eps     dq      2.6e-38      ;Двойное слово
del     dq      5.2e-34      ;Двойное слово
```

Пример 4

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1           ;6.0     ?       ?       ?
        fld     x2           ;2.0     6.0     ?       ?
        fld     x3           ;4.0     2.0     6.0     ?
        fdivr                ;2.0     6.0     ?       ?
;Сегмент данных
x1      dd      6.0
x2      dd      2.0
x3      dd      4.0
```

FDIVRP Деление двух действительных чисел в обратном порядке с последующим выталкиванием из стека

Команда идентична fdivp, но операнды (делимое и делитель) меняются местами. Варианты использования операндов показаны ниже.

```
        fdivrp ST(i),ST   ;ST(i)=ST/ST(i), выталкивание
        fdivrp ST(i)      ;ST(i)=ST/ST(i), выталкивание
        fdivrp            ;ST(1)=ST/ST(1), выталкивание
                          ;Команда аналогична команде fdivr
```

Флаги, на которые влияет команда: PE, UE, OE, ZE, DE, IE.

Пример 1

```
;Сегмент команд              ST     ST(1)    ST(2)   ST(3)
        fld     x1           ;6.0     ?        ?       ?
        fld     x2           ;2.0     6.0      ?       ?
        fld     x3           ;4.0     2.0      6.0     ?
        fdivrp ST(2)         ;2.0     0.666... ?       ?
;Сегмент данных
x1      dd      6.0
x2      dd      2.0
x3      dd      4.0
```

Пример 2

```
;Сегмент команд              ST      ST(1)    ST(2)   ST(3)
        fld     x1           ;6.0      ?        ?       ?
        fld     x2           ;3.7      6.0      ?       ?
        fld     x3           ;4.0      3.7      6.0     ?
        fdivrp               ;1.08...  6.0...   ?       ?
;Сегмент данных
x1      dd      6.0
x2      dd      3.7
x3      dd      4.0
```

FFREE Освобождение регистра стека

Данная команда помещает в тег регистра, который является ее операндом, число 11b. После этого данный регистр рассматривается как

пустой и в него можно снова записывать информацию (попытка записи в непустой регистр вызывает исключительную ситуацию).

```
ffree ST(i)        ;Регистр ST(i) очищен
```

Команда не влияет на флаги.

FIADD Сложение целого числа с действительным

При использовании этой команды необходимо указывать один целочисленный операнд. Он суммируется с содержимым вершины стека. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как слово или двойное слово.

```
fiadd mem          ;ST=ST+mem
```

Флаги, на которые влияет команда: PE, OE, DE, IE.

Пример 1

```
;Сегмент команд              ST          ST(1)
        fld    op           ;0.6785e6     ?
        fadd   dat          ;200000.0     ?
;Сегмент данных
op      dd     0.16785e6    ;Двойное слово
dat     dw     32150        ;Слово
```

FICOM Сравнение заданного целого числа с вещественным числом из вершины стека

При использовании данной команды необходимо указывать один целочисленный операнд. Он преобразуется в действительное представление и сравнивается с содержимым вершины стека. Операндом может быть только переменная, описанная как слово или двойное слово.

```
ficom mem          ;ST-mem
```

Данная команда идентична команде fcom.

Флаги, на которые влияет команда: DE, IE.

Пример

```
;Сегмент данных
nmb     dd     60000        ;Двойное слово
lim     dd     60000        ;Двойное слово
;Сегмент команд
        fild   lim          ;ST=60000.0, ST(1)=?
        ficom  nmb          ;Выполняется операция ST-nmb
                            ;C3=1, C2=0, C0=0; ST=60000.0,ST(1)=?
```

FICOMP Сравнение заданного целого числа с вещественным числом из вершины стека с последующим выталкиванием из стека

При использовании данной команды необходимо указывать один целочисленный операнд. Он преобразуется в действительное представление и сравнивается с содержимым вершины стека. После этого выполняется операция выталкивания из стека. Операндом может быть только переменная, описанная как слово или двойное слово.

```
        ficomp mem          ;ST-mem, выталкивание
```

Данная команда идентична команде fcomp.

Флаги, на которые влияет команда: DE, IE.

Пример

```
;Сегмент данных
nmb     dw      1200        ;Слово
lim     dd      60000       ;Двойное слово
;Сегмент команд
        fild    lim         ;ST=60000.0, ST(1)=?
        ficomp nmb          ;Выполняется операция ST-nmb
                            ;C3=0, C2=0, C0=0; ST=?, ST(1)=?
```

FIDIV Деление действительного числа на заданный целочисленный операнд

При использовании этой команды необходимо указывать один целочисленный операнд. Команда fidiv делит содержимое вершины стека на этот операнд. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как слово или двойное слово.

```
        fidiv  mem          ;ST=ST/mem
```

Флаги, на которые влияет команда: PE, UE, OE, ZE, DE, IE.

Пример

```
;Сегмент команд                 ST        ST(1)
        fldpi               ;3.141...     ?
        fidiv  del          ;0.785...     ?
;Сегмент данных
del     dw      4           ;Слово
```

FIDIVR Деление заданного целочисленного операнда на действительное число

При использовании этой команды необходимо указывать один целочисленный операнд. Команда fidivr делит операнд на действительное число, записанное в вершине стека. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как слово или двойное слово.

```
        fidivr mem          ;ST=mem/ST
```

Флаги, на которые влияют команды: PE, UE, OE, ZE, DE, IE.

Пример

```
;Сегмент команд                 ST        ST(1)
        fldpi               ;3.141...     ?
        fidivr grad         ;57.29...     ?
;Сегмент данных
grad    dd      180         ;Двойное слово
```

FILD Преобразование заданного целочисленного операнда в действительное представление и загрузка его в стек

При использовании этой команды необходимо указывать один целочисленный операнд. Данная команда загружает в стек заданное целое число, предварительно преобразуя его в действительный формат. Операндом может быть только переменная, описанная как слово, двойное слово или четверное слово.

```
        fild   mem          ;Загрузка стека, ST=mem
```

Флаги, на которые влияет команда: IE.

Пример

```
;Сегмент команд                 ST         ST(1)    ST(3)
        fild   d1           ;  -126          ?        ?
        fild   d2           ;  56665        -126      ?
        fild   d3           ;6576575611   56665    -126
;Сегмент данных
d1      dw     -126         ;Слово
d2      dd     56665        ;Двойное слово
d3      dq     6576575611   ;Четверное слово
```

FIMUL Умножение заданного целого числа на действительное

Команда требует один целочисленный операнд, который умножается на содержимое вершины стека. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как слово или двойное слово.

```
        fimul  mem          ;ST=ST*mem
```

Флаги, на которые влияет команда: PE, OE, DE, IE.

Пример

```
;Сегмент команд                   ST        ST(1)
        fldpi               ;3.141...     ?
        fimul  kf           ;6.283...     ?
;Сегмент данных
kf      dw     2            ;Слово
```

FINCSTP

Увеличение указателя стека

Команда не требует операндов. После ее выполнения указатель стека увеличивается на 1, то есть выполняется операция выталкивания из стека. Таким образом в вершине стека будет находится содержимое регистра ST(1). Содержимое старой вершины стека заносится в регистр ST(7).

Команда не изменяет флаги. Результат выполнения команды проиллюстрирован на рис. П3.2.

	Содержимое стека до выполнения команды	после выполнения команды
ST	0	1
ST(1)	1	2
ST(2)	2	3
ST(3)	3	4
ST(4)	4	5
ST(5)	5	6
ST(6)	6	7
ST(7)	7	0

Рис. П3.2. Результат выполнения команды fincstp.

FINIT Инициализация сопроцессора

Команда finit и аналогичная ей команда fninit не требуют операндов. Их действие подобно аппаратному сбросу сопроцессора. После выполнения этих команд слово управления устанавливается в 037Fh (округлять к ближайшему, все особые ситуации замаскированы, точность 64 бита), слово состояния очищается, а в слове признаков устанавливается 0FFFFh (все регистры пустые). Команда finit перед выполнением инициализации проверяет наличие немаскируемых условий ошибки для операций с плавающей точкой, а команда fninit этого не делает.

Команды не изменяют флаги.

FIST Преобразование числа из вершины стека в целое число и запись его в поле памяти

При использовании этой команды необходимо указывать один операнд. Данная команда округляет число из вершины стека до целого значения и записывает его в указанный операнд, которым может быть только переменная, описанная как слово, двойное слово или длинное целое, занимающее четыре слова. Способ округления определяется содержимым поля RC регистра управления сопроцессора. Если RC=00b, то выполняется округление до ближайшего целого. Если RC=01b или 10b, то число округляется в сторону уменьшения или увеличения, соответственно. При RC=11b происходит отбрасывание дробной части.

```
fist   mem          ;mem=ST
```

Флаги, на которые влияют команды: PE, IE.

Пример 1

```
;Содержимое поля RC регистра управления равно 00b
;Сегмент команд                ST        ST(1)     xx
```

```
                        ;12.459...     ?       ?
        fist   xx       ;12.459...     ?       12
;Сегмент данных
xx      dw     ?        ;Слово
```

Пример 2

```
;Содержимое поля RC регистра управления равно 10b
;Сегмент команд            ST        ST(1)     xx
                        ;12.459...     ?       ?
        fist   xx       ;12.459...     ?       13
;Сегмент данных
xx      dw     ?        ;Слово
```

FISTP Преобразование числа из вершины стека в целое число, запись его в поле памяти и выполнение операции выталкивания из стека

При использовании этой команды необходимо указывать один операнд. Данная команда округляет число из вершины стека до целого значения и записывает его в указанный операнд, которым может быть только переменная, описанная как слово, двойное слово или длинное целое, занимающее четыре слова. Способ округления определяется содержимым поля RC регистра управления сопроцессора. Если RC=00b, то выполняется округление до ближайшего целого. Если RC=01b или 10b, то число округляется в сторону уменьшения или увеличения, соответственно. При RC=11b происходит отбрасывание дробной части числа. После записи результата в память выполняется операция выталкивания из стека.

```
        fistp  mem       ;mem=ST, выталкивание
```

Флаги, на которые влияют команды: PE, IE.

Пример 1

```
;Содержимое поля RC регистра управления равно 00b
;Сегмент команд            ST        ST(1)     xx
                        ;2.59...       ?       ?
        fistp  xx       ;   ?          ?       2
;Сегмент данных
xx      dw     ?        ;Слово
```

Пример 2

```
;Содержимое поля RC регистра управления равно 11b
;Сегмент команд            ST        ST(1)     xx
                        ;2.59...       ?       ?
        fistp  xx       ;   ?          ?       2
;Сегмент данных
xx      dw     ?        ;Слово
```

FISUB Вычитание заданного целого числа из действительного

При использовании этой команды необходимо указывать один целочисленный операнд. Он вычитается из содержимого вершины стека. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как слово или двойное слово.

```
        fisub  mem          ;ST=ST-mem
```

Флаги, на которые влияют команды: PE, OE, DE, IE.

Пример

```
;Сегмент команд              ST     ST(1)
                            ;125.14   ?
        fisub  g3           ;  5.14   ?
;Сегмент данных
g3      dw     120
```

FISUBR Вычитание действительного числа из заданного целого

При использовании этой команды необходимо указывать один целочисленный операнд. Из него вычитается содержимое вершины стека. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как слово или двойное слово.

```
        fisubr mem          ;ST=mem-ST
```

Флаги, на которые влияют команды: PE, OE, DE, IE.

Пример

```
;Сегмент команд              ST     ST(1)
                            ;125.14   ?
        fisubr g3           ; -5.14   ?
;Сегмент данных
g3      dw     120
```

FLD Загрузка в стек действительного числа

При применении этой команды необходимо указывать один операнд. Команда загружает в стек действительное число, преобразуя его в формат, используемый в сопроцессоре. Операндом может быть регистр стека или переменная, занимающая двойное слово, четверное слово или десятерное слово.

```
        fld    ST(i)        ;Содержимое ST(i) загружается в ST
        fld    mem          ;Содержимое mem загружается в ST
```

Попытка выполнения команды fld при полностью загруженном стеке приводит к тому, что выполняется операция записи, однако содержимое вершины стека становится неопределенным (рис. П3.3).

Флаги, на которые влияет команда: IE.

Пример

```
;Сегмент команд                          ST          ST(1)       ST(2)
ST(3)
        fld    e34          ;3.2e38        ?           ?           ?
        fld    m2           ;1.8e-108   3.2e38         ?           ?
        fld    mt           ;1.05e4932  1.8e-108    3.2e38         ?
        fld    ST(2)        ;3.2e38     1.05e4932   1.8e-108    3.2e38
;Сегмент данных
m2      dq     1.8e-108     ;Четверное слово
e34     dd     3.2e38       ;Двойное слово
mt      dt     1.05e4932    ;Десятерное слово
```

	Содержимое стека до выполнения команды	Содержимое стека после выполнения команды
ST	0	NAN
ST(1)	1	0
ST(2)	2	1
ST(3)	3	2
ST(4)	4	3
ST(5)	5	4
ST(6)	6	5
ST(7)	7	6

Рис. П3.3. Результат выполнения команды fld при полностью загруженном стеке (NAN - значение регистра стека не определено).

FLDCW Загрузка слова управления сопроцессора

Команда использует один операнд, который загружается в регистр управления. Предварительно слово управления должно быть сохранено с помощью команды fstcw или fnstcw. Если разряды исключительных ситуаций установлены до выполнения команды fldcw и новое слово не маскирует исключительные ситуации, то удовлетворяется немедленный запрос следующей команды.

Команда не изменяет флаги.

Пример

```
;Сегмент данных
singl   dw    ?          ;Поле памяти для хранения слова
                         ;управления
;Сегмент команд
        fldcw singl
```

FLDENV Загрузка рабочей среды сопроцессора

Команда использует один операнд. В результате ее выполнения загружается рабочая среда сопроцессора из 14 или 28-байтного буфера памяти, заданного операндом. Эта информация должна быть предварительно записана в буфер с помощью команды fstenv. Размер операнда определяется тем, какой атрибут use установлен для текущего сегмента кодов. Если установлен атрибут use16, то используется 14-байтный буфер, в случае use32 - 28-байтный буфер. Если разряды исключительных ситуаций установлены до выполнения команды и новое слово не маскирует исключительные ситуации, то удовлетворяется немедленный запрос следующей команды.

Команда не изменяет флаги.

Примеры

```
;Сегмент данных
```

```
        buf    db          14 dup (?)
;Сегмент команд
        code segment use16
        ...
        fstenv buf
        ...
        fldenv buf
```

FLD1, FLDL2T, FLDL2E, FLDPI, FLDLG2, FLDLN2, FLDZ Загрузка в стек предопределенных констант

Каждая из этих команд предназначена для загрузки в стек определенной константы. Точность всех констант, за исключением 0 и 1 составляет 19 десятичных разрядов.

```
        fld1            ;Загрузка в стек единицы
        fldl2t          ;Загрузка в стек логарифма по
                        ;основанию 2 числа 10
        fldl2e          ;Загрузка в стек логарифма по
                        ;основанию 2 числа e
        fldpi           ;Загрузка в стек числа pi
        fldlg2          ;Загрузка в стек логарифма по
                        ;основанию 10 числа 2
        fldln2          ;Загрузка в стек логарифма по
                        ;основанию e числа 2
        fldz            ;Загрузка в стек нуля
```

Флаги, на которые влияют команды: IE.

Пример

```
                        ;ST      ST(1)   ST(2)   ST(3)   ST(4)
        fldz            ;0.0      ?       ?       ?       ?
        fld1            ;1.0     0.0      ?       ?       ?
        fldpi           ;3.14... 1.0     0.0      ?       ?
        fldl2t          ;3.32... 3.14... 1.0     0.0      ?
        fldlg2          ;0.30... 3.32... 3.14... 1.0     0.0
```

FMUL Умножение двух действительных чисел

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда умножает содержимое второго операнда на содержимое первого и засылает результат в первый операнд. Операндами могут быть только регистры стека.

```
        fmul    ST,ST(i)    ;ST=ST*ST(i)
        fmul    ST(i),ST    ;ST(i)=ST(i)*ST
```

Если указан один операнд, то он умножается на содержимое вершины стека. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как двойное или четверное слово.

```
        fmul    mem         ;ST=ST*mem
```

При отсутствии операндов содержимое вершины стека ST умно-

жается на содержимое находящейся под ней ячейки ST(1). После этого выполняется выталкивание из стека и результат записывается в вершину стека.

```
        fmul                ;ST(1)=ST(1)*ST, выталкивание
```

Флаги, на которые влияет команда: PE, UE, OE ,DE, IE.

Пример 1

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ;4.0    ?       ?       ?
        fld     x2          ;6.3    4.0     ?       ?
        fld     x3          ;0.4    6.3     4.0     ?
        fmul    ST,ST(2)    ;1.6    6.3     4.0     ?
;Сегмент данных
x1      dd      4.0
x2      dd      6.3
x3      dd      0.4
```

Пример 2

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ;4.0    ?       ?       ?
        fld     x2          ;6.3    4.0     ?       ?
        fld     x3          ;0.4    6.3     4.0     ?
        fmul    ST(2),ST    ;0.4    6.3     1.6     ?
;Сегмент данных
x1      dd      4.0
x2      dd      6.3
x3      dd      0.4
```

Пример 3

```
;Сегмент команд              ST      ST(1)
        fld     a1          ;4.0e26  ?
        fmul    a3          ;0.08    ?
;Сегмент данных
a1      dq      0.4e26      ;Четверное слово
a3      dq      2.0e-28     ;Четверное слово
```

Пример 4

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ;6.0    ?       ?       ?
        fld     x2          ;2.5    6.0     ?       ?
        fld     x3          ;2.0    2.5     6.0     ?
        fmul                ;5.0    6.0     ?       ?
;Сегмент данных
x1      dd      6.0
x2      dd      2.5
x3      dd      2.0
```

FMULP Умножение двух действительных чисел с последующим выталкиванием из стека

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда умножает содержимое второго операнда на содержимое первого и засылает результат в первый опе-

ранд. Затем производится выталкивание из стека. Операндами могут быть только регистры стека. В качестве первого операнда запрещается использовать регистр ST.

```
fmulp  ST(i),ST   ;ST(i)=ST(i)*ST, выталкивание
```

Если указан один операнд, то он умножается на содержимое вершины стека, результат помещается в этот операнд. Затем производится выталкивание из стека. Операндом может быть только регистр стека. Допускается использовать регистр ST.

```
fmulp  ST(i)      ;ST(i)=ST(i)*ST, выталкивание
```

При отсутствии операндов содержимое вершины стека ST умножается на содержимое находящейся под ней ячейки ST(1). Результат записывается в регистр ST(1). Затем производится выталкивание из стека. Эта команда аналогична команде fmul.

```
fmulp             ;ST(1)=ST(1)*ST, выталкивание
```

Флаги, на которые влияет команда: PE, UE, OE, ZE, DE, IE.

Пример 1

```
;Сегмент команд          ST     ST(1)  ST(2)  ST(3)
        fld    x1       ;4.0    ?      ?      ?
        fld    x2       ;0.7    4.0    ?      ?
        fld    x3       ;0.4    0.7    4.0    ?
        fmulp  ST(2)    ;0.7    1.6    ?      ?
;Сегмент данных
x1      dd     4.0
x2      dd     0.7
x3      dd     0.4
```

Пример 2

```
;Сегмент команд          ST     ST(1)  ST(2)
        fld    x1       ;4.0    ?      ?
        fld    x2       ;0.7    4.0    ?
        fmulp  ST       ;4.0    ?      ?
;Сегмент данных
x1      dd     4.0
x2      dd     0.7
```

Пример 3

```
;Сегмент команд          ST     ST(1)  ST(2)  ST(3)
        fld    x1       ;6.0    ?      ?      ?
        fld    x2       ;2.5    6.0    ?      ?
        fld    x3       ;2.0    2.5    6.0    ?
        fmulp           ;5.0    6.0    ?      ?
;Сегмент данных
x1      dd     6.0
x2      dd     2.5
x3      dd     2.0
```

FNCLEX Очистка флагов исключительных ситуаций сопроцессора

См. описание команды fclex.

FNINIT Инициализация сопроцессора

См. описание команды finit.

FNOP Нет операции

Команда nop не выполняет никакой операции. Она влияет только на значение указателя команд.

Команда не изменяет флаги.

FNSAVE Запись состояния сопроцессора

См. описание команды fsave.

FNSTCW Запись слова управления сопроцессора

См. описание команды fstcw.

FNSTENV Запись рабочей среды сопроцессора

См. описание команды fstenv.

FNSTSW Запись слова состояния сопроцессора

См. описание команды fstsv.

FPATAN Вычисление арктангенса угла

Команда не требует операндов. Она вычисляет арктангенс величины, равной ST(1)/ST. При выполнении этой команды производится выталкивание из стека и, таким образом, полученное значение угла замещает оба операнда. Значение угла вычисляется в радианах.

```
        fpatan          ;ST(1)=arctg[ST(1)/ST],
                        ;выталкивание
```

Флаги, на которые влияет команда: PE, UE, OE, DE.

Пример

```
;Сегмент команд          ST      ST(1)     ST(2)
        fld1            ;1.0      ?         ?
        fld1            ;1.0      1.0       ?
        fpatan          ;0.78...  ?         ?
        fldpi           ;3.14...  0.78...   ?
        fdiv            ;0.25...  ?         ?
        fimul  c        ;45.0      ?        ?
;Сегмент данных
c       dw     180
```

FPREM Вычисление частичного остатка деления

Команда не требует операндов. Она вычисляет остаток, полученный от деления ST на ST(1). Знак остатка тот же, что и знак исходного делимого в ST.

Флаги, на которые влияют команды: UE, DE, IE.

Пример

```
;Сегмент команд          ST      ST(1)     ST(2)
        fld    op1      ;10.0     ?         ?
        fld    op2      ;17.0     10.0      ?
        fprem           ;7.0...   10.0      ?
```

```
;Сегмент данных
op1     dd      10.0
op2     dd      17.0
```

FPREM1 Вычисление частичного остатка деления по стандарту IEEE-754 (только начиная с процессора 387)

Команда не требует операндов. Она вычисляет остаток, полученный от деления ST на ST(1), и оставляет остаток в ST. По этой команде выполняется операция ST=ST-ST(1), до тех пор, пока не выполнится условие ST<ST(1)/2.

Флаги, на которые влияют команды: UE, DE, IE.

Пример 1

```
;Сегмент команд          ST      ST(1)     ST(2)
        fld     op1     ;10.0     ?         ?
        fld     op2     ;17.0    10.0       ?
        fprem           ;-3.0...  10.0       ?
;Сегмент данных
op1     dd      10.0
op2     dd      17.0
```

Пример 2

```
;Сегмент команд          ST        ST(1)     ST(2)
        fld     op1     ;10.0      ?         ?
        fld     op2     ;14.9999   10.0      ?
        fprem           ; 4.9999   10.0      ?
;Сегмент данных
op1     dd      10.0
op2     dd      14.9999
```

FPTAN Вычисление тангенса угла

Команда не требует операндов. Она вычисляет тангенс угла, заданного в вершине стека. Угол должен быть задан в радианах.

Выполнение этой команды зависит от типа сопроцессора. При использовании сопроцессоров 8087 и 80287 значение угла должно находиться в диапазоне от 0 до pi/4. Результат в виде двух величин X и Y записывается в ST(1) и ST. Для завершения вычисления надо разделить ST на ST(1).

При использовании сопроцессоров 80387+ значение модуля угла не должно превышать 2 в степени 63. После выполнения команды угол заменяется его тангенсом, вслед за чем в стек помещается число 1.0.

Флаги, на которые влияет команда: PE, DE, IE.

Пример 1 (для 8087, 80287)

```
;Сегмент команд          ST        ST(1)
        fldpi           ;3.14...    ?
        fidiv c         ;1.047...   ?
        fptan           ;1.093...   1.8938...
        fdiv            ;1.732...
;Сегмент данных
c       dw      3
```

Пример 2 (для 80387+)

```
;Сегмент команд              ST        ST(1)
        fldpi               ;3.14...    ?
        fidiv  c            ;0.785...   ?
        fptan               ;1          1.732...
;Сегмент данных
c       dw     3
```

FRNDINT Округление действительного числа из вершины стека до целого значения

Команда не требует операндов. Она округляет действительное число из вершины стека до целого в соответствии со значением поля RC слова управления сопроцессора. Результат заменяет исходное действительное число в вершине стека. Если RC=00b, то выполняется округление до ближайшего целого. Если RC=01b или 10b, то число округляется в сторону уменьшения или увеличения, соответственно. При RC=11b происходит отбрасывание дробной части числа. Для того, чтобы изменить режим округления, можно записать слово управления командой fstcw и, изменив содержимое поля RC, загрузить его снова с помощью команды fldcw.

Флаги, на которые влияет команда: PE, IE.

Пример

```
;Сегмент данных
nmb     dd     23.6666667
;Сегмент команд
        fld    nmb          ;ST<-nmb
        frndint             ;ST=24, при RC=00b или 10b
                            ;ST=23, при RC=01b или 11b
```

FRSTOR Восстановление состояния сопроцессора

Команда использует один операнд. Она восстанавливает полное состояние сопроцессора по содержимому буфера памяти (размером 94 или 108 байтов), заданному этим операндом. Размер буфера определяется тем, какой атрибут use установлен для текущего сегмента кодов. Если установлен атрибут use16, то используется 94-байтный буфер, а если установлен атрибут use32, то 108-байтный буфер. Для того, чтобы сформировать содержимое буфера, предварительно используется команда fsave.

Команда не изменяет флаги.

Пример

```
;Сегмент данных
buf     db     94 dup (?)
;Сегмент команд
code    segment use16
        ...
        fsave  buf
        ...
        frstor buf
```

FSAVE Запись состояния сопроцессора

Команда fsave и аналогичная ей команда fnsave используют один операнд. Они записывают полное состояние сопроцессора в буфер памяти размером 94 или 108 байтов, заданный этим операндом. Размер буфера определяется тем, какой атрибут use установлен для текущего сегмента кодов. Если установлен атрибут use16, то используется 94-байтный буфер, а если установлен атрибут use32, то 108-байтный буфер. После выполнения этих команд производится инициализация сопроцессора. При выполнении команды fsave перед записью состояния производится проверка условия немаскируемой ошибки с плавающей точкой. При выполнении команды fnsave такая проверка не производится.

Команды не изменяют флаги.

Примеры

```
;Сегмент данных
buf     db      94 dup (?)
;Сегмент команд
code    segment use16
        ...
        fnsave buf
```

FSCALE Масштабирование действительного числа

Команда не требует операндов. Она умножает содержимое вершины стека ST на два в степени ST(1). Если содержимое регистра ST(1) оказывается нецелым, команда использует ближайшее меньшее по величине целое число. Таким образом, с помощью этой команды можно выполнять умножение или деление на целочисленные степени 2.

Флаги, на которые влияет команда: UE, OE, IE.

Пример 1

```
;Сегмент команд          ST          ST(1)
        fild   pow      ;5.0          ?
        fild   p1       ;8.0          5.0
        fscale          ;256.0        5.0
;Сегмент данных
p1      dw     8
pow     dw     5
```

Пример 2

```
;Сегмент команд          ST          ST(1)
        fild   pow      ;2.0          ?
        fild   p1       ;73.87        2.0
        fscale          ;295.48       2.0
;Сегмент данных
p1      dd     73.87
pow     dw     2
```

FSIN Вычисление синуса (только начиная с процессора 80387)

Команда не требует операндов. Она вычисляет синус действительного числа, расположенного в вершине стека, при этом предполагается,

что аргумент задан в радианах. Результат выполнения команды помещается в стек на место операнда. Следует иметь в виду, что модуль величины угла не должен превышать 2 в степени 63, причем следить за этим должен сам программист. Если значение аргумента выходит из заданного диапазона, то, после выполнения команды, ST не изменится и будет установлен флаг C2 слова состояния.

```
fsin                ;ST=sin(ST)
```

Флаги, на которые влияет команда: PE, DE, IE.

Пример

```
                    ;ST         ST(1)
fldpi               ;3.1416...    ?
fsin                ;0.0          ?
```

FSINCOS Одновременное вычисление синуса и косинуса (только начиная с процессора 80387)

Команда не требует операндов. Она вычисляет синус и косинус действительного числа, расположенного в вершине стека, при этом предполагается, что аргумент задан в радианах. Результат выполнения команды помещается в стек на место операнда. Следует иметь в виду, что модуль величины угла не должен превышать 2 в степени 63, причем следить за этим должен программист. Если значение аргумента выходит из заданного диапазона, то, после выполнения команды, ST не изменится и будет установлен флаг C2 слова состояния. После вычисления значение косинуса помещается в регистр ST, а значение синуса в ST(1).

```
fsincos             ;ST=cos(ST), ST(1)=sin(ST)
```

Флаги, на которые влияет команда: PE, DE, IE.

Пример

```
                    ;ST         ST(1)
fldpi               ;3.1416...    ?
fsincos             ;-1.0        0.0
```

FSQRT Вычисление квадратного корня

Команда не требует операндов. Она вычисляет квадратный корень положительного действительного числа, расположенного в вершине стека. Результат помещается на место исходного числа.

```
fsqrt               ;ST=sqrt(ST)
```

Флаги, на которые влияет команда: PE, DE, IE.

Пример

```
;Сегмент команд          ST          ST(1)
        fild   p1       ;2.0          ?
        fsqrt           ;1.414...     ?
;Сегмент данных
p1      dw     2
```

FST Запись действительного числа

При использовании этой команды необходимо указывать один операнд, в который копируется содержимое вершины стека. Операндом может быть регистр стека или переменная, занимающая двойное или четверное слово. Перед записью выполняется округление в соответствии с значением поля RC управляющего слова до размеров операнда. Если RC=00b, то выполняется округление до ближайшего числа. Если RC=01b или 10b, то число округляется в сторону уменьшения или увеличения, соответственно. При RC=11b лишние цифры дробной части числа отбрасываются.

```
fst    mem        ;mem=ST
fst    ST(i)      ;ST(i)=ST
```

Флаги, на которые влияет команда:PE, UE, OE, IE.

Пример 1

```
               ;   ST       ST(1)     ST(2)
                            ;1.2459e1976    ?        ?
        fst    ST(2)        ;1.2459e1976    ?   1.2459e1976
```

Пример 2

```
;Сегмент команд              ;   ST          ST(1)    zo
                             ;1.34e-189...    ?       ?
        fst    zo           ;1.34e-189...    ?       0
;Сегмент данных:
zo      dd     ?            ;Двойное слово
```

Пример 3

```
;Сегмент команд                 ST          ST(1)    zz
                             ;1.34e-189...    ?       ?
        fst    zz           ;1.34e-189...    ?  1.34e-189
;Сегмент данных:
zz      dq     ?            ;Четверное слово
```

FSTCW Запись слова управления сопроцессора

Команда fstcw и аналогичная ей команда fnstcw используют один операнд. В результате выполнения в этот операнд записывается текущее слово управления сопроцессора. При выполнении команды fstcw перед записью состояния производится проверка условия немаскируемой ошибки с плавающей точкой. При выполнении команды fnstcw такая проверка не производится.

Команда не изменяет флаги.

Пример

```
;Сегмент данных
cword   dw     ?
;Сегмент команд
        fstcw cword
        fnstcw cword
```

FSTENV Запись рабочей среды сопроцессора

Команда fstenv и аналогичная ей команда fnstenv используют один операнд. В результате выполнения текущая рабочая среда сопроцессора записывается в 14- или 28-байтный буфер памяти, заданный операндом. Размер операнда определяется тем, какой атрибут use установлен для текущего сегмента кодов. Если установлен атрибут use16, то используется 14-байтный буфер, в случае use32 - 28-байтный. Рабочая среда сопроцессора включает в себя слово управления, слово состояния, слово признаков и указатели исключительных ситуаций. При выполнении команды fstenv перед записью состояния производится проверка условия немаскируемой ошибки с плавающей точкой. При выполнении команды fnstenv такая проверка не производится.

Команды не изменяют флаги.

Пример

```
;Сегмент данных
buf     db      14 dup (?)
;Сегмент команд
code    segment use16
        ...
        fnstenv         buf
```

FSTP Запись действительного числа и выталкивание из стека

При использовании этой команды необходимо указывать один операнд; команда копирует в него содержимое вершины стека. Операндом может быть регистр стека, а также переменная, занимающая двойное или четверное слово. Перед записью выполняется округление в соответствии со значением поля RC управляющего слова до размеров операнда. Если RC=00b, то выполняется округление до ближайшего числа. Если RC=01b или 10b, то число округляется в сторону уменьшения или увеличения, соответственно. При RC=11b лишние цифры дробной части числа отбрасываются. После записи производится выталкивание из стека.

```
        fstp    mem         ;mem=ST, выталкивание
        fstp    ST(i)       ;ST(i)=ST, выталкивание
```

Флаги, на которые влияет команда: PE, UE, OE, IE.

Пример 1

```
                            ;   ST              ST(1)     ST(2)
                            ;1.2459e1976         12.1        ?
        fstp    ST(2)       ;   12.1          1.2459e1976   ?
```

Пример 2

```
;Сегмент команд                 ST          ST(1)    ST(2)    zo
                            ;1.34e-189...    1.9        ?       ?
        fstp    zo          ;   1.9          ?          ?       0
;Сегмент данных
zo      dd      ?           ;Двойное слово
```

Пример 3

```
;Сегмент команд                ST       ST(1)   ST(2)   zz
                              ;1.34e-189  1.9      ?       ?
        fstp    zo            ;  1.9       ?      ?   1.34e-189
;Сегмент данных
zz      dq      ?             ;Четверное слово
```

FSTSW Запись слова состояния сопроцессора

Команда fstsw и схожая с ней команда fnstsw используют один операнд. В операнд, в качестве которого может быть использовано не только поле данных, определенное как dw, но и регистр AX, записывается текущее слово состояния сопроцессора. При выполнении команды fstsw перед записью слова состояния производится проверка условия немаскируемой ошибки с плавающей точкой. При выполнении команды fnstsw такая проверка не производится. Команды не изменяют флаги.

Пример

```
;Сегмент данных
sword   dw      ?
;Сегмент команд
        fstsw   sword
        fnstsw AX
```

FSUB Вычитание двух действительных чисел

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда вычитает второй операнд из содержимого первого и засылает результат в первый операнд. Операндами могут быть только регистры стека.

```
        fsub    ST,ST(i)    ;ST=ST-ST(i)
        fsub    ST(i),ST    ;ST(i)=ST(i)-ST
```

Если указан один операнд, то он вычитается из содержимого вершины стека, результат помещается в вершину стека. Операндом может быть только переменная, описанная как двойное или четверное слово.

```
        fsub    mem         ;ST=ST-mem
```

При отсутствии операндов вычитается содержимое вершины стека ST из находящейся под ней ячейки ST(1). Результат записывается в ST(1) и после этого выполняется выталкивание из стека.

```
        fsub                ;ST(1)=ST(1)-ST, выталкивание
```

Флаги, на которые влияет команда: PE, UE, OE, DE, IE.

Пример 1

```
;Сегмент команд              ST      ST(1)   ST(2)   ST(3)
        fld     x1          ; 4.2     ?       ?       ?
        fld     x2          ; 6.3    4.2      ?       ?
        fld     x3          ;10.0    6.3     4.2      ?
        fsub    ST,ST(2)    ; 5.8    6.3     4.2      ?
```

```
;Сегмент данных
x1      dd      4.2
x2      dd      6.3
x3      dd      10.0
```

Пример 2

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ; 4.2    ?       ?       ?
        fld     x2          ; 6.3   4.2      ?       ?
        fld     x3          ;10.0   6.3     4.2      ?
        fsub    ST(2),ST    ;10.0   6.3    -5.8      ?
;Сегмент данных
x1      dd      4.2
x2      dd      6.3
x3      dd      10.0
```

Пример 3

```
;Сегмент команд              ST     ST(1)   ST(2)
        fld     x1          ;11.5    ?       ?
        fld     x2          ; 6.3   11.5     ?
        fsub    ST,ST       ; 0.0   11.5     ?
;Сегмент данных
x1      dd      11.5
x2      dd      6.3
```

Пример 4

```
;Сегмент команд              ST      ST(1)
        fld     um          ;4.0e307  ?
        fsub    wch         ;3.8e307  ?
;Сегмент данных
um      dq      0.4e308     ;Четверное слово
wch     dq      2.0e306     ;Четверное слово
```

Пример 5

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ; 6.0    ?       ?       ?
        fld     x2          ; 2.0   6.0      ?       ?
        fld     x3          ; 4.0   2.0     6.0      ?
        fsub                ;-2.0   6.0      ?       ?
;Сегмент данных
x1      dd      6.0
x2      dd      2.0
x3      dd      4.0
```

FSUBP Вычитание двух действительных чисел и запись результата в любой регистр стека с последующим выталкиванием из стека

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда вычитает второй операнд из первого и засылает результат в первый операнд. Затем производится выталкивание из стека. Операндами могут быть только регистры стека. В качестве первого операнда запрещается использовать регистр ST.

```
fsubp  ST(i),ST   ;ST(i)=ST(i)-ST, выталкивание
```

Если указан один операнд, то он вычитается из содержимого вершины стека, результат помещается в операнд. Затем производится выталкивание из стека. Операндом может быть только регистр стека. Допускается использовать регистр ST.

```
fsubp  ST(i)       ;ST(i)=ST(i)-ST, выталкивание
```

При отсутствии операндов вычитается содержимое вершины стека ST из находящейся под ней ячейки ST(1). Результат записывается в регистр ST(1). Затем производится выталкивание из стека.

```
fsubp              ;ST(1)=ST(1)-ST, выталкивание
```

Флаги, на которые влияет команда: PE, UE, OE, DE, IE.

Пример 1

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld    x1           ; 4.0    ?       ?       ?
        fld    x2           ; 1.5   4.0      ?       ?
        fld    x3           ;10.0   1.5     4.0      ?
        fsubp  ST(2),ST     ; 1.5  -6.0      ?       ?
;Сегмент данных
x1      dd     4.0
x2      dd     1.5
x3      dd     10.0
```

Пример 2

```
;Сегмент команд              ST     ST(1)   ST(2)
        fld    f            ; 4.0    ?       ?
        fld    s            ;10.0   4.0      ?
        fsubp  ST(1)        ;-6.0            ?
;Сегмент данных
f       dd     4.0
s       dd     10.0
```

Пример 3

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld    x1           ;6.0     ?       ?       ?
        fld    x2           ;2.0    6.0      ?       ?
        fld    x3           ;1.5    2.0     6.0      ?
        fsubp               ;0.5    6.0      ?       ?
;Сегмент данных
x1      dd     6.0
x2      dd     2.0
x3      dd     1.5
```

FSUBR Вычитание двух действительных чисел в обратном порядке

Данная команда схожа с командой fsub, но отличается от нее тем, что вычитаемое и уменьшаемое меняются местами.

```
        fsubr  ST,ST(i)    ;ST=ST(i)-ST
        fsubr  ST(i),ST    ;ST(i)=ST-ST(i)
        fsubr  mem         ;ST=mem-ST
        fsubr              ;ST(1)=ST-ST(1), выталкивание
```

Флаги, на которые влияет команда: PE, UE, OE, DE, IE.

Пример 1

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ; 4.2    ?       ?       ?
        fld     x2          ; 6.3    4.2     ?       ?
        fld     x3          ;10.0    6.3     4.2     ?
        fsubr   ST,ST(2)    ;-5.8    6.3     4.2     ?
;Сегмент данных
x1      dd      4.2
x2      dd      6.3
x3      dd      10.0
```

Пример 2

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ; 4.2    ?       ?       ?
        fld     x2          ; 6.3    4.2     ?       ?
        fld     x3          ;10.0    6.3     4.2     ?
        fsubr   ST(2),ST    ;10.0    6.3     5.8     ?
;Сегмент данных
x1      dd      4.0
x2      dd      6.3
x3      dd      10.0
```

Пример 3

```
;Сегмент команд              ST        ST(1)
        fld     wch         ;2.6e-35     ?
        fsubr   wch+8       ;4.94e-34    ?
;Сегмент данных
wch     dq      2.6e-35, 5.2e-34
```

Пример 4

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ;6.0      ?       ?       ?
        fld     x2          ;2.0     6.0      ?       ?
        fld     x3          ;4.0     2.0     6.0      ?
        fsubr               ;2.0     6.0      ?       ?
;Сегмент данных
x1      dd      6.0
x2      dd      2.0
x3      dd      4.0
```

FSUBRP Вычитание двух действительных чисел в обратном порядке с последующим выталкиванием из стека

Данная команда схожа с командой fsubp, но отличается от нее тем, что вычитаемое и уменьшаемое меняются местами.

```
        fsubrp ST,ST(i)     ;ST=ST(i)-ST, выталкивание
        fsubrp ST(i)        ;ST(i)=ST(i)-ST, выталкивание
        fsubrp              ;ST(1)=ST-ST(1), выталкивание
```

Флаги, на которые влияет команда: PE, UE, OE, DE, IE.

Пример 1

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ;6.0      ?       ?       ?
        fld     x2          ;2.0     6.0      ?       ?
        fld     x3          ;4.5     2.0     6.0      ?
        fsubrp ST(2)        ;2.0    -1.5      ?       ?
```

```
;Сегмент данных
x1      dd      6.0
x2      dd      2.0
x3      dd      4.5
```

Пример 2

```
;Сегмент команд          ST      ST(1)   ST(2)   ST(3)
        fld     x1      ;6.0      ?       ?       ?
        fld     x2      ;3.7     6.0      ?       ?
        fld     x3      ;4.2     3.7     6.0      ?
        fsubrp          ;0.5     6.0      ?       ?
;Сегмент данных
x1      dd      6.0
x2      dd      3.7
x3      dd      4.2
```

FTST Сравнение вершины стека с нулем

Команда не требует операндов, выполняя сравнение числа из вершины стека с нулем. По результатам сравнения устанавливаются разряды C3, C2 и C0 слова состояния (табл.П3.3).

Флаги, на которые влияет команда: DE, IE.

FUCOM Неупорядоченное сравнение вещественных чисел (только начиная с процессора 80387)

Существует два варианта использования команды: с одним операндом и без операндов. В том и в другом случае эта команда производят неупорядоченное сравнение содержимого вершины стека с содержимым одного из регистров стека. Сравниваемый регистр может быть явно указан в виде операнда.

```
        fucom  ST(i)        ;Неупорядоченное сравнение
                            ;ST с ST(i)
```

Таблица П3.3. Значения разрядов C0, C2 и C3 слова состояния сопроцессора после выполнения команды fst.

Результат сравнения	C3	C2	C0
ST>0.0	0	0	0
ST<0.0	0	0	1
ST=0.0	1	0	0
ST=?	1	1	1

Если операнд не указан, то содержимое вершины стека сравнивается с содержимым регистра ST(1).

```
        fucom               ;Неупорядоченное сравнение
                            ;ST с ST(1)
```

По результатам сравнения устанавливаются разряды C3, C2 и C0 слова состояния (табл. П3.4).

В отличие от команды fucom, команда неупорядоченного сравнения не вызывает исключения "недействительная операция" в случае NAN операнда. NAN операнд появляется при выполнении операций занесения данных в занятый регистр стека или чтения из свободного регистра.

Флаги, на которые влияет команда: DE, IE.

Таблица П3.4. Значения разрядов C0, C2 и C3 слова состояния сопроцессора после выполнения команд неупорядоченного сравнения.

Результат сравнения	C3	C2	C0
ST>ST(i)	0	0	0
ST<ST(i)	0	0	1
ST=ST(i)	1	0	0
ST не сравнимо с ST(i)	1	1	1

Пример

```
                    ;ST      ST(1)      ST(2)
                    ;45.7    34.8       127.9
        fucom ST(2) ;C3=0, C2=0, C0=1
        fucom       ;C3=0, C2=0, C0=0
```

FUCOMP Неупорядоченное сравнение вещественных чисел с выполнением операции выталкивания числа, находящегося в вершине стека (только начиная с процессора 80387)

Существует два варианта использования команды: с одним операндом и без операндов. В том и в другом случае эта команда производят неупорядоченное сравнение содержимого вершины стека с содержимым одного из регистров стека. Затем выполняется операция выталкивания из стека. Сравниваемый регистр может быть явно указан в виде операнда

```
        fucomp ST(i)    ;Неупорядоченное сравнение
                        ;ST с ST(i), выталкивание
```

Если операнд не указан, то содержимое вершины стека сравнивается с содержимым регистра ST(1), после чего выполняется операция выталкивания из стека

```
        fucomp          ;Неупорядоченное сравнение
                        ;ST с ST(1), выталкивание
```

По результатам сравнения устанавливаются разряды C3, C2 и C0 слова состояния (см. табл. П3.4).

В отличие от команды fucom, команда неупорядоченного сравнения не вызывает исключения "недействительная операция" в случае NAN операнда. NAN операнд появляется при выполнении операций занесения данных в занятый регистр стека или чтения из свободного регистра.

Флаги, на которые влияет команда: DE, IE.

Пример

```
                    ;ST     ST(1)    ST(2)
                    ;45.3   42.7     42.7
fucomp ST(2)        ;42.7   42.7      ? C3=0, C2=0, C0=0
fucomp              ;42.7    ?        ? C3=1, C2=0, C0=0
```

FUCOMPP Неупорядоченное сравнение вещественных чисел и выталкивание сравниваемых чисел из стека (только начиная с процессора 80387)

Команда fucompp выполняет неупорядоченное сравнение содержимого вершины стека с содержимым регистра ST(1). По результатам сравнения устанавливаются разряды C3, C2 и C0 слова состояния (см. табл.П3.4). После выполнения операции сравниваемые числа выталкиваются из стека.

```
fucompp             ;Неупорядоченное сравнение ST с
                    ;ST(1), выталкивание, выталкивание
```

В отличие от команды fucom, команда неупорядоченного сравнения не вызывает исключения "недействительная операция" в случае NAN операнда. NAN операнд появляется при выполнении операций занесения данных в занятый регистр стека или чтения из свободного регистра.

Флаги, на которые влияет команда: DE, IE.

Пример

```
                    ;ST     ST(1)    ST(2)
                    ;45.3   12.7     42.3
fucompp             ;42.3    ?        ? C3=0, C2=0, C0=0
```

FWAIT Ожидание окончания работы сопроцессора

Команда не требует операндов. Она синхронизирует работу основного процессора и сопроцессора, приостанавливая действия процессора до завершения сопроцессором текущей операции. В процессорах Pentium, 80486DX и сопроцессоре 80387 команды сопроцессора автоматически синхронизируются, то есть процессор ожидает окончания выполнения предыдущей команды сопроцессора перед запуском следующей. При использовании сопроцессора 8087 необходимо наличие команды fwait для гарантии синхронизации.

Команда не изменяет флаги.

FXAM Проверка содержимого вершины стека

Команда не требует операндов. Она возвращает информацию о содержимом вершины стека, устанавливая разряды C3, C2, C1 и C0 слова состояния сопроцессора, в соответствии с таблицей П3.5.

Команда не изменяет флаги.

Таблица П3.5. Значения разрядов C0, C1, C2 и C3 слова состояния сопроцессора после выполнения команды fxat.

Содержимое вершины стека	C3	C2	C1	C0
Положительное ненормализованное число	0	0	0	0
Нечисло (знак +)	0	0	0	1
Отрицательное ненормализованное число	0	0	1	0
Нечисло (знак -)	0	0	1	1
Положительное нормализованное число	0	1	0	0
+ бесконечность	0	1	0	1
Отрицательное нормализованное число	0	1	1	0
- бесконечность	0	1	1	1
+ 0.0	1	0	0	0
Пусто	1	0	0	1
- 0.0	1	0	1	0
Пусто	1	0	1	1
Положительное денормализованное число	1	1	0	0
Пусто	1	1	0	1
Отрицательное денормализованное число	1	1	1	0
Пусто	1	1	1	1

FXCH Обмен содержимым двух регистров сопроцессора

Существует два варианта использования данной команды: без операндов и с одним операндом.

Если задан один операнд, то производится обмен содержимым этого операнда и вершины стека. В качестве операнда можно использовать только регистр стека.

```
fxch   ST(i)        ;Обмен содержимым между ST и ST(i)
```

При отсутствии операндов выполняется обмен содержимым регистров ST и ST(1).

```
fxch                ;Обмен содержимым между ST и ST(1)
```

Флаги, на которые влияет команда: IE.

Пример

```
                    ;ST          ST(1)        ST(2)
                    ;8.1e4931    12.7         1.0e-1465
fxch   ST(2)        ;1.0e-1465   12.7         8.1e4931
fxch                ;12.7        1.0e-1465    8.1e4931
```

FXTRACT Выделение показателя степени и мантиссы

Команда не требует операндов. Она разделяет число, расположенное в вершине стека, на мантиссу и показатель степени. Вместо числа записывается показатель степени. После этого в стек записывается мантисса.

Флаги, на которые влияет команда: IE.

Пример 1

```
;Сегмент данных
numb    dw    8.0           ;Два в третьей степени
```

```
;Сегмент команд          ST      ST(1)     ST(2)
        fild   mmb      ;8.0      ?         ?
        fxtract          ;1       3.0       ?
```

Пример 2

```
;Сегмент данных
mmb     dd     14.54
;Сегмент команд          ST        ST(1)     ST(2)
        fild   mmb      ;14.54      ?         ?
        fxtract          ;1.817...  3.0       ?
```

FYL2X Вычисление Y*Log2X

Команда не требует операндов. Она вычисляет произведение числа, записанного в ST(1), на логарифм по основанию 2 числа, записанного в ST. После этого выполняется выталкивание из стека и результат записывается на место сомножителя. Таким образом, после выполнения операции исходные числа теряются. ST не может быть отрицательным.

Флаги, на которые влияет команда: PE, UE, OE, DE.

Пример 1

```
;Сегмент данных
mmb     dw     8          ;Два в третьей степени
;Сегмент команд          ST      ST(1)     ST(2)
        fld1             ;1.0      ?         ?
        fild   mmb       ;8.0     1.0        ?
        fyl2x            ;3.0      ?         ?
```

Пример 2

```
;Сегмент данных
mmb     dw     1024       ;Два в десятой степени
;Сегмент команд          ST          ST(1)       ST(2)
        fldpi            ;3.14...      ?          ?
        fild   mmb       ;1024        3.14...     ?
        fyl2x            ;31.4...      ?          ?
```

Пример 3

```
;Сегмент данных
h1      dt     2.0e-128
a12     dd     1269.56
;Сегмент команд          ST                ST(1)       ST(2)
        fld    h1        ;2.0e-128          ?          ?
        fld    a12       ;1269.56        2.0e-128      ?
        fyl2x            ;2.06...e-127      ?          ?
```

Пример 4

```
;Сегмент данных
mmb     dd     1.24
;Сегмент команд          ST              ST(1)     ST(2)
        fld1             ;1.0              ?         ?
        fld    mmb       ;0.24            1.0        ?
        fyl2xp1          ;0.3103401317     ?         ?
```

FYL2XP1 Вычисление Y*Log2(X+1)

Команда не требует операндов. Она вычисляет произведение числа, записанного в регистр ST(1), на логарифм по основанию 2 от числа, за-

писанного в ST, сложенного с единицей. После этого выполняется выталкивание из стека и результат записывается на место сомножителя. Таким образом после выполнения операции исходные числа теряются. Операнд ST должен находиться в диапазоне sqrt(2)/2-1<= ST<=sqrt(2)-1

Флаги, на которые влияет команда: PE, UE, OE, DE, IE.

Пример

```
;Сегмент данных
nmb     dd      0.24
;Сегмент команд          ST              ST(1)     ST(2)
        fld1            ;1.0              ?         ?
        fld     nmb     ;0.24             1.0       ?
        fyl2xp1         ;0.3103401143     ?         ?
```

Книга содержит свыше 100 законченных программ, которые позволят вам за короткое время освоить базовые понятия языка ассемблера и архитектуры машин типа IBM PC:

адресное пространство и способы адресации в реальном и защищенном режимах

система команд микропроцессоров Intel

арифметические и логические операции и преобразование данных

организация подпрограмм

макросредства ассемблера

программирование аппаратуры через пространство памяти и порты

использование прерываний BIOS и функций DOS работа в защищенном режиме

программирование арифметического сопроцессора.

Книга написана в нетрадиционной манере, строгому изложению основ языка авторы предпочли рассмотрение множества простых и наглядных примеров, каждый из которых позволяет достаточно глубоко разобраться в том или ином принципиальном вопросе.