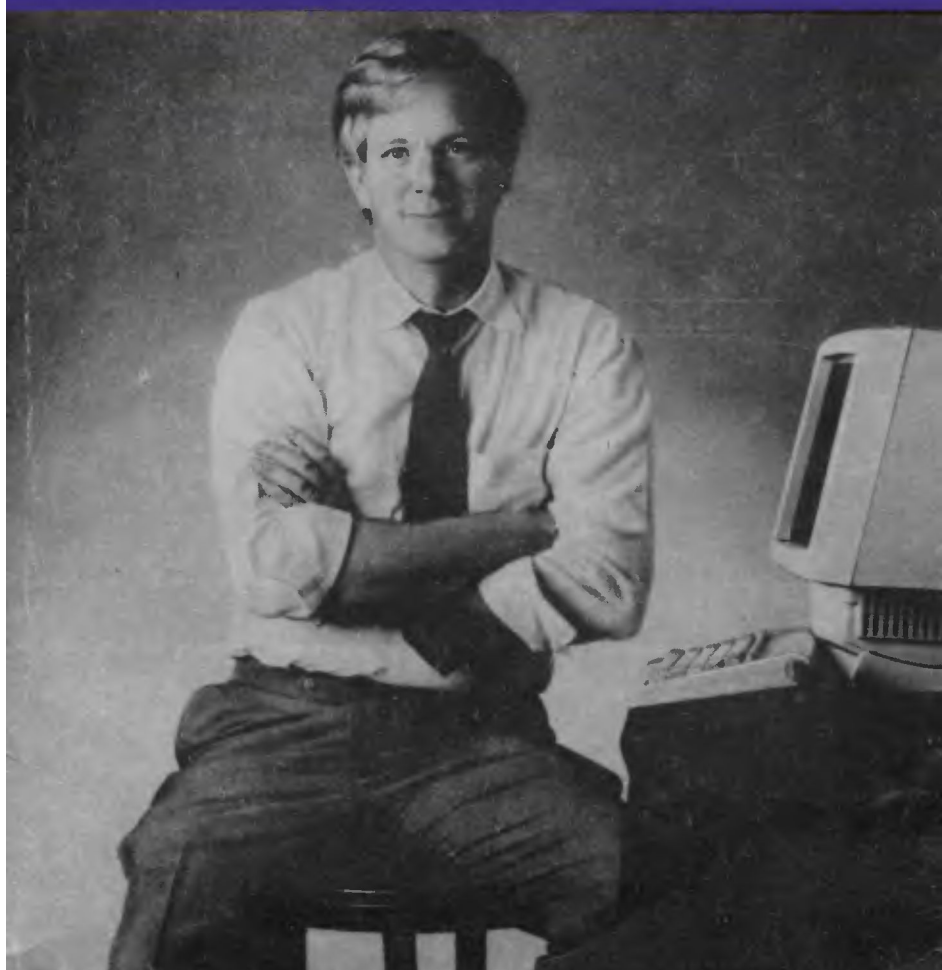


П.Нортон, Д.Соухэ

Язык ассемблера для IBM PC



П.Нортон, Д.Соухэ
Язык ассемблера для IBM PC

Peter Norton's Assembly Language Book for the IBM PC
Peter Norton John Socha

BRADY, New York

Москва, Издательство "Компьютер", 1992

ББК 32.973-01

Н 83

УДК 681.3.06

Н83

Нортон П., Соухэ Д.

Язык ассемблера для IBM PC: Пер. с англ., - М.: Издательство "Компьютер";
Финансы и статистика, 1992. - 352 с: ил.

ISBN 5-279-00936-9.

Книга предназначена для читателей-программистов,
желающих изучить язык ассемблер и возможности
микропроцессоров 8088 с целью написания более
мощных, быстрых и коротких программ.

"Гроссмейстер" программирования Питер Нортон
делится своим богатым опытом с читателями. Книга
существенно расширяет кругозор пользователей IBM
PC.

ББК 32.973-01

240409000-039

Н-----КБ 27-016-91

010(01)-92

(c) 1986 by Brady Books, a
division of Simon & Schuster, Inc

(c) Издательство "Компьютер",
перевод, 1992

ISBN 0-13-661901-0 (США)

ISBN 5-279-00936-9 (РФ)

(c) Финансы и статистика,
1992

Введение

К тому времени, как вы закончите чтение этой книги, вы будете знать, как создавать полноценные программы на языке ассемблера: редакторы текста, утилиты, и т. д. Вы познакомитесь с техническими приемами профессиональных программистов, облегчающих процедуру написания программ. Будут рассмотрены модульное конструирование программ ("modular design") и их пошаговое усовершенствование ("stepwise refinement")/пошаговое уточнение, которые наверняка удвоят вашу скорость программирования, а также помогут вам писать легко читаемые и более надежные программы.

Техника пошагового усовершенствования, в частности, значительно облегчает написание сложных программ. Если вам уже приходилось испытывать ощущение, что вы тонете в собственной программе, то вы найдёте, что пошаговое усовершенствование открывает вам простой и естественный путь написания программ. Кроме всего прочего, оно ещё и довольно увлекательно.

Однако теория не является единственным содержанием этой книги. Мы также создадим программу, которая называется Dskpatch (может применяться для "починки" диска), и вы найдете её полезной по нескольким причинам. Прежде всего, вы увидите пошаговое усовершенствование и модульное конструирование в действии в реальной программе, таким образом, у вас появится возможность узнать, почему эти методы так полезны. Кроме того, Dskpatch - это полноэкранный редактор общего назначения для работы с секторами диска, который может использоваться как самостоятельная программа и после прочтения книги.

Почему именно Ассемблер?

Мы предполагаем, что вы взялись за эту книгу, потому что заинтересованы в изучении Ассемблера. Но, возможно, вы не совсем точно уверены, почему именно его вы хотите изучать.

Основная причина заключается в том, что программы на ассемблере понятны для любого IBM PC или совместимого с ним компьютера. По отношению ко всем остальным языкам программирования, ассемблер - язык, наиболее близкий к машинному. Он позволит вам ближе познакомиться с машиной (в отличие от языков программирования высокого уровня), и поэтому изучение ассемблера означает также изучение самого микропроцессора 8088. Мы не только обучим вас инструкциям микропроцессора, чем иногда ограничиваются авторы других книг, но пойдём значительно дальше и дадим дополнительный материал, важность которого вы оцените, когда начнёте писать свои собственные программы.

После того, как вы познакомитесь с микропроцессором 8088, многие вещи в различных программах, включая и написанные на языках высокого уровня, будут иметь для вас большее значение. Например, вы возможно замечали, что максимальное целое положительное число, применяемое в Бейсике, равно 32767. Откуда взялось это число? Это довольно странное число для верхнего предела. Но, как вы увидите позже, число 32767 непосредственно связано с тем, каким образом ваш компьютер хранит числа.

Кроме того, возможно вас интересует быстроедействие и длина создаваемых программ. Как правило, программа, написанная на ассемблере, выполняется гораздо быстрее программы, написанной на любом другом языке. Обычно ассемблерные программы выполняются в два, три раза быстрее, чем эквивалентные программы Си или Паскаля, и в 15 и более раз быстрее, чем программы, пошагово интерпретируемые Бейсиком. Ассемблерные программы также значительно меньше по размеру. Так например, длина программы Dskpatch, которую мы вместе с вами напишем в этой книге, составит около одного килобайта. По сравнению с абсолютным большинством программ это немного. Похожие программы, написанные на Си или Паскале, будут длиннее раз в десять. Именно поэтому корпорация Lotus Development написала свой продукт Lotus 1-2-3 полностью на ассемблере.

Программы на ассемблере также открывают вам полный доступ к возможностям компьютера. Некоторые программы, например SideKick, ProKey и SuperKey, остаются в памяти компьютера после того, как вы их запустите. Такие программы изменяют стиль работы вашей машины, они используют возможности,

доступные только для программ, написанных на ассемблере.

Dskpatch

В нашей работе с ассемблером мы подробно рассмотрим дисковые сектора, вывод на экран символов и чисел, хранимых DOS в шестнадцатеричном виде. Dskpatch - полноэкранный редактор дисков, позволяющий изменять символы и числа, хранящиеся на секторе диска. Используя Dskpatch, вы можете, например, просматривать сектор, где DOS хранит директорию диска и изменять имена файлов или иную информацию. Делая это, можно изучить то, как DOS хранит информацию на диске.

Так как Dskpatch содержит около 50 подпрограмм, а многие из них являются подпрограммами общего назначения, то они могут пригодиться вам при создании собственных программ. Фактически книга представляет собой не только введение в изучение микропроцессора 8088 и программирование на Ассемблере, но и источник полезных подпрограмм.

Кроме того, для создания любого экранного редактора используются возможности, характерные для всего семейства компьютеров IBM PC. С помощью примеров, приведённых в этой книге, вы также научитесь писать полезные программы для IBM PC, AT и совместимых компьютеров, таких как COMPAQ.

Требования к конфигурации компьютера

Какое оборудование необходимо, чтобы вы могли запускать программы, приведенные в этой книге? Вам нужен IBM PC или совместимый компьютер с, по крайней мере, 128К памяти и одним дисководом. Вам также понадобится MS DOS (или PC DOS) версии 2.00 или выше. И, начиная с Части III, вам потребуется IBM или Microsoft Macro Assembler.

Организация этой книги

Эта книга разделена на три части, каждая из которых имеет свои особенности. Вне зависимости от того, знаете вы или нет что-либо о микропроцессорах или ассемблере, вы найдете в ней главы, наверняка заинтересующие вас.

В Части I рассматривается микропроцессор 8088. Здесь вы узнаете тайны битов, байтов и язык машины. Каждая из семи глав содержит изобилие реальных примеров, в которых используется программа Debug, входящая в пакет программ DOS. Debug позволит нам заглянуть внутрь микропроцессора 8088. Часть I подразумевает наличие у читателя, по крайней мере, начальных знаний Бейсика и того, как работать с компьютером.

Часть II, главы с 8 по 16 посвящены ассемблеру и написанию на нём программы. Подход к этому будет постепенным, и вместо того, чтобы заниматься описанием деталей самого ассемблера, мы сконцентрируем внимание на наборе команд ассемблера, необходимых для создания полезных программ.

Мы используем ассемблер также для того, чтобы переписать некоторые программы из части I и затем приступим к созданию Dskpatch. Мы будем создавать эту программу медленно, так что вы научитесь использовать пошаговое усовершенствование в построении больших программ. Заодно мы рассмотрим и чисто технические приёмы (например такие, как модульное конструирование), которые помогут создать понятные программы. Как отмечалось выше, эти методы упростят программирование, устраняя некоторые сложности, обычно ассоциирующиеся с написанием программ на ассемблере.

В части III, которая включает главы 17-29, мы сконцентрируем внимание на использовании дополнительных возможностей IBM PC - перемещении курсора и очищении экрана.

В части III мы также обсудим методы отладки больших ассемблерных программ. Программы, написанные на ассемблере, растут очень быстро и легко могут достигать длины двух и более страниц, не делая может быть при этом ничего полезного (Dskpatch будет гораздо длиннее). После того как мы используем эти отладочные методы на программах, размер которых превышает несколько страниц, вы найдёте их полезными и для небольших программ.

Ну а сейчас перейдем к микропроцессору 8088 и посмотрим, каким образом и как он хранит числа.

Часть I. Язык машины

Глава 1. DEBUG и арифметика

Итак, начнём наше обучение на ассемблере с представления того, как компьютер считает.

В повседневной деятельности мы привыкли к счёту с применением десяти цифр 1,2,3 и т. д. Вследствие чисто технических особенностей компьютер применяет другой метод счёта, в котором задействованы только две цифры 0 и 1. Например, до пяти он считает следующим образом: 1, 10, 11, 100, 101. Числа 10, 11, 100 и т. д. являются двоичными, то есть базирующимися на системе счисления, состоящей всего из двух цифр - единицы и нуля, в отличие от десяти цифр, соответствующих более привычной для нас десятичной системе. Таким образом, двоичное число 10 соответствует десятичному 2.

Однако двоичные числа обладают существенным недостатком - выглядят они длинно и громоздко. Шестнадцатеричные числа - гораздо более компактный способ записи двоичных чисел. В этой главе вы познакомитесь с двумя способами записи чисел: шестнадцатеричным и двоичным. Это поможет понять процесс счёта в компьютере и способ хранения чисел - в битах, байтах и словах. Если вы уже имеете представление о двоичных и шестнадцатеричных числах, битах, байтах и словах, то вам достаточно ознакомиться лишь с итогом этой главы.

Шестнадцатеричные числа

Так как с шестнадцатеричными числами легче обращаться, чем с двоичными числами (по крайней мере, из-за длины), то мы начнём со знакомства с шестнадцатеричными числами, и будем использовать для этого DEBUG.COM, специальную программу, которую вы найдёте на дополнительном диске DOS.

Мы будем использовать Debug и в этой, и в следующих главах для ввода и пошагового выполнения программ, написанных на машинном языке. Как и

Бейсик, Debug обеспечивает удобную диалоговую среду. Но в отличие от Бейсика, он не распознаёт десятичных чисел. Для Debug число 10 является шестнадцатеричным, а не "десяткой". И так как Debug говорит только на шестнадцатеричном языке, вам понадобится узнать кое-что о шестнадцатеричных числах. Но сначала разберёмся с программой Debug.

Debug

Почему программа называется Debug? "Bugs" (дословно "насекомые") в переводе со слэнга программистов означает "ошибки в программе". В работающей программе этих ошибок нет, в то время как неработающая ("limping") программа имеет по крайней мере одну ошибку ("bug"). Используя Debug для пошагового запуска программы и наблюдая, как программа работает на каждом этапе, мы можем найти ошибки и исправить их. Этот процесс называется отладка ("debugging"), отсюда и произошло название программы Debug.

В соответствии с компьютерным фольклором, термин "debugging" (дословно "обезжучивание", "обезнасекомливание") имеет глубокие корни - он появится в тот день, когда перестал работать компьютер Гарвардского университета Марк I. После долгих поисков техники обнаружили источник своих бед - небольшую моль, попавшую между контактами реле. Они удалили моль и внесли запись в сменный журнал о процессе под названием "debugging", произведенном над Марком I.

Найдите Debug на вашем дополнительном диске DOS, и мы начнём. Пожалуй, вам даже стоит скопировать DEBUG.COM на ваш рабочий диск, так как мы будем его постоянно использовать в Части I этой книги.

Примечание: начиная с этого места, в тексте будут приводиться распечатки команд пользователя и ответов компьютера. Напечатайте текст примера, нажмите клавишу "Enter", и вы увидите ответ компьютера, соответствующий тому, который приводится в распечатке. Однако, ответы компьютера могут не всегда совпадать с приведенными в книге из-за возможных различий между вашим компьютером и компьютером, с помощью которого писалась эта книга (мы обсудим эти различия позже). Отметим, что во

всех примерах используются заглавные буквы. Это сделано только для того, чтобы избежать путаницы между буквой "I" и цифрой "1". Если вы предпочитаете, то можете набирать примеры строчными буквами.

Теперь, после упомянутых выше соглашений, запустим Debug, набрав его название после приглашения DOS (которое в этом примере выглядит как "A:\>"):

```
A:\>DEBUG
```

Дефис, который вы видите в качестве ответа на вашу команду - это приглашение программы Debug, в то время как "A:\>" - это приглашение DOS. Это означает, что Debug ждёт вашей команды.

Чтобы покинуть Debug и вернуться в DOS, напечатайте "Q" (англ. "Quit") около дефиса и нажмите "Ввод". Если хотите, попробуйте выйти и затем обратно вернуться в Debug:

```
-Q  
A:\>DEBUG
```

Теперь мы можем вернуться к изучению шестнадцатеричных чисел.

Шестнадцатеричная арифметика

Мы будем использовать команду Debug, которая называется "H". "H" - это сокращение от английского слова "Hexarithmetic" [StIV] : "Hexadecimal arithmetic's" ("шестнадцатеричная арифметика"), и, как предполагает это слово, команда "H" складывает и вычитает два шестнадцатеричных числа. Давайте посмотрим, как работает "H", и начнем с $2 + 3$. Мы знаем, что $2 + 3 = 5$ для десятичных чисел. Истинно ли это для шестнадцатеричных чисел? Убедитесь, что вы по-прежнему находитесь в программе Debug, и напечатайте после дефиса следующий текст:

```
-H 3 2  
0005 0001
```

Debug печатает как сумму (0005), так и разность (0001) 3 и 2. Команда "H" всегда подсчитывает и

сумму, и разность двух чисел, в чём вы только что убедились. Итак, результаты пока одинаковы для шестнадцатеричных и десятичных чисел: и для десятичной системы счисления 5 сумма 2 и 3, а 1 - разность. Но в дальнейшем вы можете столкнуться с некоторыми сюрпризами.

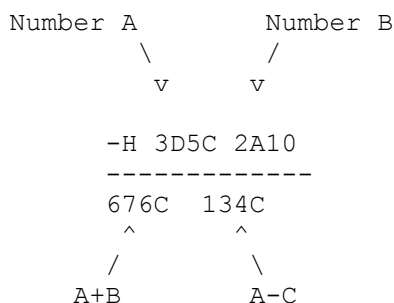


Рис.1.1. Команда шестнадцатеричной арифметики

Например, что будет, если мы напечатаем "Н 2 3", то есть сложить и вычесть два и три, а не три и два? Если мы попытаемся это сделать, то:

```
-Н 3 2
0005 FFFF
```

мы получили FFFF вместо -1 для 2 - 3. Возможно, это выглядит странно, но FFFF - шестнадцатеричное число, соответствующее единице со знаком минус "-1".

В дальнейшем мы вернёмся к этой необычной -1. Но сначала давайте возьмём другие числа, чтобы увидеть, каким образом F может появиться в качестве числа.

Попробуем сложить девять и один, что должно дать нам десятичное число 10:

```
-Н 9 1
000A 0008
```

Девять плюс один равняется A? Да, это так: A - шестнадцатеричное число, соответствующее десяти. Так, а что если мы попробуем получить ещё большее число, такое, например, как 15:

```
-Н 9 6
000F 0003
```

Если вы попробуете исследовать остальные числа между десятью и пятнадцатью, то вы обнаружите всего 16 цифр от 0 до F (от 0 до 9 и от A до F). Название "шестнадцатеричные" произошло от числа 16. Цифры от 0 до 9 одинаковы и для шестнадцатеричной, и для десятичной системы счисления; шестнадцатеричные цифры от A до F соответствуют десятичным числам от 10 до 15.

Почему Debug говорит на шестнадцатеричном языке? Скоро вы увидите, что мы можем записать 256 различных чисел с помощью двух шестнадцатеричных цифр. Как вы уже наверное подозреваете, 256 также имеет некоторое отношение к элементу, известному как байт, и байт играет главную роль в компьютерах и в этой книге. Вы найдёте значительно большую информацию о байтах ближе к концу этой главы, но сейчас мы продолжим изучение шестнадцатеричных чисел, единственной системы счисления, известной программе Debug, и шестнадцатеричной математики.

Перевод шестнадцатеричных чисел в десятичную форму

До этого мы рассматривали шестнадцатеричные числа, состоящие из одной цифры. Теперь давайте посмотрим, как выглядят шестнадцатеричные числа большей длины и как переводить эти числа в десятичную форму.

Как и с десятичными числами, мы получаем шестнадцатеричное число, состоящее из нескольких цифр, добавляя

Decimal	Hex	digit
0		0
1		1
2		2
3		3
4		4
5		5
6		6
7		7
8		8
9		9
10		A
11		B
12		C
13		D
14		E
15		F

Рис. 1.2. Шестнадцатеричные цифры

цифры слева. Предположим, например, что мы складываем число 1 и наибольшее десятичное число, состоящее из одной цифры, 9. В результате получается число, состоящее из двух цифр, 10 (десять). Что произойдет, если мы прибавим 1 к наибольшему шестнадцатеричному числу, состоящему из одной цифры, F? Мы опять получим десять.

Но подождите, десятка в шестнадцатеричной системе на самом деле 16, а не десять. Так может получиться небольшая путаница. Нам нужно как-то различать эти две десятки, поэтому, начиная с этого места, мы будем помещать букву "h" после каждого шестнадцатеричного числа. Таким образом, мы будем понимать, что 10h это шестнадцатеричное 16, а 10 это десятичное десять.

Теперь мы подошли к вопросу о том, как переводить числа из шестнадцатеричной в десятичную форму и обратно. Мы знаем, что 10h это 16, но как мы переведем большее шестнадцатеричное число, такое, например, как D3h, в десятичную форму без того, чтобы считать до D3h, начиная с 10h? Или как нам перевести десятичное число 173 в шестнадцатеричную форму?

Мы не можем положиться на помощь Debug, так как Debug не понимает десятичных чисел. В главе 10 мы напишем программу для перевода шестнадцатеричного числа в десятичную форму так, чтобы она могла выдавать нам десятичные числа. Но сейчас придется выполнять эти действия вручную, так что мы начнем с возвращения в знакомый мир десятичных чисел.

Что означает число 276? В младшей школе мы узнали, что 276 состоит из двух сотен, семи десятков и шести единиц. Графически это можно представить следующим образом:

$$\begin{array}{rcl} 2 & * & 100 & = & 200 \\ 7 & * & 10 & = & 70 \\ 6 & * & 1 & = & 6 \\ \hline & & 276 & = & 276 \end{array}$$

Такое представление числа помогает нам визуально понять значение, входящих в него цифр. Можем ли мы использовать такое представление при работе с шестнадцатеричным числом? Конечно.

Возьмём число D3h, которое мы упоминали ранее.
D - это шестнадцатеричная цифра 13, и там 16 таких цифр (по аналогии с 10 для десятичного числа).
Таким образом, D3h - это тринадцать раз по шестнадцать и три единицы. Или, в графическом представлении:

$$\begin{array}{rcl} D & \rightarrow & 13 * 16 = 208 \\ 3 & \rightarrow & 3 * 1 = 3 \\ \hline D3h & & = 211 \end{array}$$

Когда мы рассматривали десятичное число 276, мы умножали цифры на 100, 10 и 1, а цифры шестнадцатеричного числа D3h мы умножили на 16 и 1. Если бы мы имели десятичное число из четырёх цифр, мы бы умножали на 1000, 100, 10 и 1. Какие четыре числа нам использовать, имея шестнадцатеричное число из четырёх цифр?

Для десятичной системы счисления используются числа 1000, 100, 10 и 1 - степени числа 10:

$$\begin{array}{l} 10^3 = 1000 \\ 10^2 = 100 \\ 10^1 = 10 \\ 10^0 = 1 \end{array}$$

Мы можем использовать тот же метод для шестнадцатеричных цифр, но со степенями 16, а не 10, и поэтому будем применять следующие четыре числа:

$$\begin{array}{l} 16^3 = 4096 \\ 16^2 = 256 \\ 16^1 = 16 \\ 16^0 = 1 \end{array}$$

Переведём 3AC8h в десятичную форму, используя четыре числа, которые мы только что вычислили:

$$\begin{array}{rcl} 3 & \rightarrow & 3 * 256 = 768 \\ A & \rightarrow & 10 * 16 = 160 \\ C & \rightarrow & 12 * 1 = 12 \\ 8 & \rightarrow & 8 * 1 = 8 \\ \hline 3AC8h & & = 1,016 \end{array}$$

```

A --> 10 * 4,096 = 40,960
F --> 15 * 256 = 3,840
1 --> 1 * 16 = 16
C --> 12 * 1 = 12
-----
AF1Ch = 44,828

3 --> 3 * 65,536 = 196,608
B --> 11 * 4,096 = 45,056
8 --> 8 * 256 = 2,048
D --> 13 * 16 = 208
2 --> 2 * 1 = 2
-----
3B8D2h = 243,922

```

Рис.1.3. Примеры перевода

```

3 --> 3 * 4096 = 12288
A --> 10 * 256 = 2560
C --> 12 * 16 = 192
8 --> 8 * 1 = 8
-----
3AC8H = 15048

```

Теперь посмотрим, что произойдёт, если мы сложим шестнадцатеричные числа, состоящие более чем из одной цифры.

Для этого мы используем Debug и числа 3A7h и 1EDh:

```

-Н 3A7 1ED
0594 01BA

```

Таким образом, мы видим, что 3A7h + 1EDh = 594h. Вы можете проверить результаты, переведя эти числа в десятичную форму и выполнив сложение (или вычитание, если хотите) в десятичной форме; если вы отважны, выполните вычисления прямо в шестнадцатеричной форме.

```

  1      1      1
 2A7    F451    C
+ 92A    + CB03    + D
-----
CD1      1DF54     19

```

```

 1111      1 1
BCD8      BCD8
+ FAE9      + 0509
-----
1B7C1      C1E1

```

Рис.1.4. Примеры сложения шестнадцатеричных чисел

Пятизначные шестнадцатеричные числа

Итак, шестнадцатеричная математика продолжается. Что произойдёт, если мы попытаемся сложить ещё большие шестнадцатеричные числа? Сложим пятизначное и четырёхзначное числа:

```
-H 5C3F0 4BC6
  ^ Error
-
```

Да, это неожиданный ответ. Почему Debug говорит, что у нас здесь ошибка (error)? Причина этого связана с единицей хранения информации, называемой СЛОВО. Команда Debug "H" работает только со словами, а длина слов такова, что они могут содержать не более четырёх шестнадцатеричных цифр.

Более подробно о словах мы узнаем через несколько страниц, ну а сейчас запомните, что Debug может работать только с четырьмя шестнадцатеричными цифрами. Таким образом, если вы пытаетесь сложить два четырёхзначных шестнадцатеричных числа, например таких, как C000h и D000h (которые в сумме дадут 19000h), то вы получите только 9000h:

```
-H C000 D000
9000 F000
-
```

Дело в том, что Debug сохраняет только четыре правых цифры ответа.

Перевод десятичных чисел в шестнадцатеричную форму

До этого мы рассматривали только перевод из шестнадцатеричной формы представления числа в десятичную. Теперь мы научимся переводить десятичные числа в шестнадцатеричные. Как уже упоминалось, в главе 10 мы создадим программу для записи чисел микропроцессора 8088 в десятичной форме; в главе 23 мы напишем другую программу, считывающую десятичные числа в микропроцессор 8088. Однако, также как и в случае с переводом из шестнадцатеричной формы в десятичную, начнём изучение с того, как подобные действия выполняются вручную. Мы опять воспользуемся знаниями математики, полученными в младшей школе.

Когда нас обучали делению, мы делили 9 на 2 и получали остаток 1. Остаток применяется при переводе числа из десятичной формы в шестнадцатеричную. Посмотрим, что произойдёт, если мы будем последовательно делить десятичное число, на этот раз 493, на 10:

$$\begin{array}{rcl}
 493/10 & = & 49 \text{ remainder } 3 \\
 \hline
 | & & \\
 \downarrow & & \\
 49/10 & = & 4 \text{ remainder } 9 \\
 \hline
 | & & \\
 \downarrow & & \\
 4/10 & = & 0 \text{ remainder } 4
 \end{array}$$

$\begin{array}{c} \downarrow \downarrow \downarrow \\ 4 \ 9 \ 3 \end{array}$

Цифры числа 493 появляются в виде остатка, расположенного в обратном порядке - начиная с крайней правой цифры (3).

Ранее мы уже узнали, что для перевода шестнадцатеричного числа в десятичную форму потребовалось лишь заменить степени 10 степенями 16.

$$\begin{array}{rcl}
 1069/16 & = & 66 \text{ remainder } 13 \\
 \hline
 | & & \\
 \downarrow & & \\
 66/16 & = & 4 \text{ remainder } 4 \\
 \hline
 | & & \\
 \downarrow & & \\
 4/16 & = & 0 \text{ remainder } 4
 \end{array}$$

$\begin{array}{c} \downarrow \downarrow \downarrow \\ 4 \ 2 \ D \ h \end{array}$

1069 = 4 2 D h

$$\begin{array}{rcl}
 57,109/16 & = & 3,569 \text{ remainder } 5 \\
 \hline
 | & & \\
 \downarrow & & \\
 3,569/16 & = & 223 \text{ remainder } 2 \\
 \hline
 | & & \\
 \downarrow & & \\
 223/16 & = & 13 \text{ remainder } 15 \\
 \hline
 | & & \\
 \downarrow & & \\
 13/16 & = & 0 \text{ remainder } 13
 \end{array}$$

$\begin{array}{c} \downarrow \downarrow \downarrow \downarrow \\ D \ F \ 1 \ 5 \ h \end{array}$

57,109 = D F 1 5 h

Рис.1.5. Примеры перевода десятичных чисел в шестнадцатеричную форму

Можем ли мы для того, чтобы перевести десятичное число в шестнадцатеричную форму, также разделить его на 16, а не на 10? Да, конечно, в действительности перевод из десятичной системы счисления в шестнадцатеричную производится именно таким образом.

Например, найдём шестнадцатеричный эквивалент числа 493. Делим на 16, как показано ниже:

-- 16 --

Мы нашли, что 1EDh является шестнадцатеричным эквивалентом десятичного числа 493. Другими словами, делим на 16 и формируем результирующее шестнадцатеричное число из остатков. Вот и всё.

Отрицательные числа

Если вы помните, у нас есть неразрешённая загадка в виде числа FFFFh. Мы сказали, что FFFFh фактически равно -1. Однако, если мы переведём FFFFh в десятичную форму, мы получим 65535. Почему это происходит? Ведёт ли себя FFFFh действительно как отрицательное число?

Пусть так, тогда если мы сложим FFFFh (по нашей гипотезе -1) и 5, результат должен быть 4, так как $5 - 1 = 4$. Произойдёт ли это? Используя команду Debug "H" для сложения 5 и FFFFh, получим:

```
-H 5 FFFF
0004 0006
-
```

Debug, кажется, обращается с FFFFh, как с -1. Но FFFFh не всегда будет вести себя как -1 в программах, которые мы будем писать. Чтобы увидеть, почему это происходит, произведём сложение вручную.

При сложении двух десятичных чисел мы часто выполняем перенос единицы в следующий столбец, как, например, в данном случае:

```
  95
+ 58
----
 153
```

Сложение двух шестнадцатеричных чисел производится аналогично.

Например, сложение 3 и F даёт нам 2 с переносом 1 в следующий столбец:

```
  F
+ 3
----
 12h
```

А теперь посмотрим, что произошло, когда мы сложили 5h и FFFFh:

```
0005h
+FFFFh
-----
10004h
```

Так как при $Fh + 1h = 10h$, происходит последовательный перенос единицы в крайнюю левую позицию, то, если мы игнорируем эту единицу, мы получаем правильный ответ для $5 - 1$: именно, 4. Как не странно это покажется, FFFFh ведёт себя как -1, когда мы игнорируем это переполнение. Оно названо переполнением, так как число теперь пятизначное, а Debug сохраняет только первые (правые) четыре цифры.

Переполнение - ошибка, или правильный ответ? И на тот, и на другой вопрос можно ответить "да". Не противоречат ли ответы между собой? - Нет, так как эти числа могут рассматриваться двумя способами.

Допустим, мы приняли FFFFh равным 65536. Это положительное число, и оно максимальное из тех, что мы можем записать с помощью четырёх шестнадцатеричных цифр. Мы говорим, что FFFFh число без знака (англ. "unsigned"). Оно действительно беззнаковое, так как мы только что определили все четырёхзначные числа как положительные. Сложение 5 и FFFFh даёт нам 10004h; это единственный верный ответ. Следовательно, в случае чисел без знака переполнение - ошибка.

С другой стороны, мы можем рассматривать FFFFh как отрицательное число, как это делал Debug, когда мы использовали команду "H", чтобы сложить FFFFh и 5. FFFFh ведёт себя как -1, пока мы игнорируем переполнение. Фактически, все числа от 8000h до FFFFh прекрасно ведут себя как отрицательные числа. Для чисел со знаком, как здесь и показано, переполнение не является ошибкой. Микропроцессор 8088 может рассматривать числа как со знаком, так и без знака; выбирать вам. Инструкции для тех и других слегка различаются, и мы рассмотрим эти различия в следующих главах, как только начнём использовать числа в микропроцессоре 8088. Сейчас же, до того, как вы сможете записать отрицательное число, соответствующее, скажем, числу 3C8h, нам необходимо разобраться со специфическим понятием

бита и его связью с байтом, словом и шестнадцатеричным числом.

Биты, байты, слова и двоичная система счисления

Настало время углубиться в изучение вашего IBM PC - узнать об арифметических действиях, выполняемых микропроцессором 8088 над двоичными числами. Микропроцессор 8088, при всей его мощности, слегка глуп. Он знает только две цифры: 0 и 1, так что любое число, которое он использует, должно быть сформировано из длинной строки нулей и единиц. Это и есть двоичная система счисления.

Когда Debug выводит на дисплее число в шестнадцатеричной форме, то при этом он применяет небольшую программу, переводящую двоичные числа, обрабатываемые микропроцессором, в шестнадцатеричную форму представления. В главе 5 мы создадим такую программу сами, но сначала нам надо побольше узнать собственно о двоичных числах.

Давайте возьмём двоичное число 1011b ("b", сокращение от англ. "binary" - двоичный, признак двоичного числа). Это число соответствует десятичному 11 и шестнадцатеричному Bh. Чтобы увидеть, почему это так, умножим цифры числа 1011b на основу системы счисления, 2:

$$\begin{aligned}2^3 &= 8 \\2^2 &= 4 \\2^1 &= 2 \\2^0 &= 1\end{aligned}$$

$$\begin{aligned}1 * 8 &= 8 \\0 * 4 &= 0 \\1 * 2 &= 2 \\1 * 1 &= 1\end{aligned}$$

1011b = 11 или Bh

Аналогично, 1111b это FFFFh, или 15. И 1111b является наибольшим беззнаковым четырёхзначным числом, которое мы можем записать, в то время как 0000b - наименьшее. Таким образом, с помощью четырёх двоичных цифр мы можем записать 16 различных чисел. Имеется также 16 шестнадцатеричных цифр, так что мы можем писать одну шестнадцатеричную цифру вместо каждых четырёх двоичных цифр.

Двузначное шестнадцатеричное число, такое как 4Ch, может быть записано как 0100 1100b. Оно состоит из восьми цифр, которые мы разделили на группы по четыре для простоты чтения. Каждая из этих двоичных цифр называется бит, так что число 0100 1100b, или 4Ch, имеет длину восемь бит.

Часто бывает удобно нумеровать каждый из битов длинной строки, начиная с самого крайнего справа бита, который имеет номер 0. Так, например, в числе 10b бит, в котором записана 1, имеет номер 1; а крайний слева бит в числе 1011b - номер 3. Нумерация битов таким способом упрощает разговор о каждом конкретном бите.

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Рис.1.6. Двоичные, шестнадцатеричные и десятичные числа от 0 до F

Группа из восьми двоичных цифр называется байт, а группа из 16 двоичных цифр, или два байта, называется слово. Мы будем часто использовать эти термины в книге, так как биты, байты и слова являются фундаментальными понятиями для любого микропроцессора, в том числе и для 8088.

Теперь мы видим, почему шестнадцатеричная система счисления удобна; две шестнадцатеричные цифры помещаются точно в один байт (четыре бита на одну цифру), а четыре шестнадцатеричных цифры помещаются точно в одно слово. Мы не можем сказать то же самое о десятичных числах. Если мы попробуем поместить два десятичных числа в один байт, мы не сможем записать числа, большие чем 99, так что мы потеряем значения от 100 до 255 - более

чем половиной чисел, которые может содержать байт. А если мы будем использовать для этой же цели три десятичных цифры, то нам придётся игнорировать более половины трёхзначных десятичных чисел, так как числа от 256 до 999 не могут поместиться в один байт.

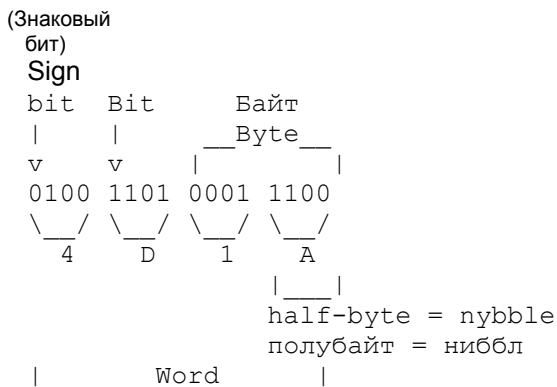


Рис.1.7.Слово, состоящее из битов и байт

Дополнительный код - особый тип отрицательного числа

Теперь мы готовы к более глубокому изучению отрицательных чисел. Мы говорили, что в том случае, если мы игнорируем переполнение, то все числа с 8000h по FFFFh ведут себя как отрицательные. Проще всего опознать отрицательные числа, записав их в двоичном виде:

Положительные числа:

0000h 0000 0000 0000 0000b

.
. .
. . .

7FFFh 0111 1111 1111 1111b

Отрицательные числа:

8000h 1000 0000 0000 0000b

.
. .
. . .

FFFFh 1111 1111 1111 1111b

В двоичной форме записи у всех положительных чисел крайний левый бит (бит номер 15) всегда равен 0, а для отрицательных 1. Эта разница и является тем признаком, по которому микропроцессор 8088 распознаёт отрицательные или положительные числа: при обработке чисел, он воспринимает пятнадцатый (знаковый) бит каждого числа. Если мы используем в нашей программе инструкции для чисел без знака, то

микропроцессор 8088 будет игнорировать знаковый бит.

Отрицательные числа в таком представлении известны как дополнительный код (или двоичное дополнение) положительных чисел. Почему дополнение? Потому, что переход от положительного числа, например такого как 3C8h, к его дополнительному коду - это процесс из двух этапов, первый из которых - переход от числа к его дополнению.

Мы почти не будем применять отрицательные числа, но произведём этот переход для того, чтобы вы увидели, как микропроцессор 8088 делает числа отрицательными. Метод перевода может показаться вам немного странным, но давайте посмотрим, как это делается.

Чтобы найти дополнительный код любого числа, надо сначала записать число в двоичной форме, игнорируя знак. Например, 4Ch становится 0000 0000 0100 1100b.

Чтобы сделать это число отрицательным, сначала реверсируем (переворачиваем) все нули и единицы. Этот процесс реверсирования называется дополнением, и взяв дополнение от 4Ch, мы найдём, что:

```
0000 0000 0100 1100
становится:
1111 1111 1011 0011
```

на втором этапе преобразования к дополнению добавляется 1b

```
1111 1111 1011 0011
+                               1
-----
1111 1111 1011 0100
-4Ch = FFB4h
```

Результат, который мы получили, соответствует вычитанию 4Ch из 0h.

При желании вы можете сложить FFB4h и 4Ch вручную, и удостовериться, что ответ равен 10000h. Из нашего более раннего обсуждения вы знаете, что эту единицу необходимо игнорировать, чтобы получить 0 ($4C + (-4C) = 0$), при выполнении сложения в дополнительном коде.

Итог

Эта глава явилась довольно энергичным вступлением в мир шестнадцатеричных и двоичных чисел. В дальнейшем в главе 3, мы медленно и осторожно

двинемся дальше, а теперь давайте вдохнём глоток свежего воздуха и ещё раз осмыслим то, что мы узнали.

Мы начали со встречи с программой Debug. В последующих главах мы с Debug станем близкими друзьями, но так как она не понимает привычные нам десятичные числа, мы начали нашу дружбу с изучения новой системы счисления - шестнадцатеричной.

Изучая шестнадцатеричные числа, вы также узнали, как переводить десятичные числа в шестнадцатеричные и обратно. После того как мы освоили основные понятия шестнадцатеричной системы, мы смогли обратить свой взор на биты, байты, слова, с важными свойствами которых вы скоро столкнетесь при исследовании структуры микропроцессора 8088 и программирования на языке ассемблера.

Затем мы занялись изучением отрицательных шестнадцатеричных чисел - дополнительного кода чисел. Они привели нас к числам со знаком и без знака, где мы столкнулись с переполнением двух типов: когда переполнение даёт правильный ответ (сложение двух чисел со знаком); и когда переполнение приводит к неправильному ответу (сложение двух чисел без знака).

Пользу от этой учебы вы в полной мере прочувствуете в следующих главах, в которых мы будем использовать наши знания о шестнадцатеричных числах для общения с Debug, и Debug будет действовать как интерпретатор между нами и микропроцессором 8088. Следующая глава будет посвящена изучению самого микропроцессора 8088. Мы опять положимся на Debug, и будем при общении с микропроцессором использовать шестнадцатеричные числа, предпочитая их двоичным. Мы также узнаем о регистрах микропроцессора - устройствах, в которых он хранит числа, и уже в главе 3 мы будем готовы к тому, чтобы написать настоящую программу, которая будет печатать символ на экране. Мы также узнаем о том, как микропроцессор 8088 выполняет свои математические операции; к тому времени, когда мы достигнем главы 10, мы сможем написать программу, которая будет переводить двоичные числа в десятичные.

Глава 2. Арифметика микропроцессора 8088

Зная кое-что о шестнадцатеричной арифметике программы Debug (отладчик) и двоичной арифметике микропроцессора 8088, мы можем начать изучение того, как 8088 выполняет свои математические операции. Он использует внешние команды, называемые инструкциями.

Регистры как переменные

Debug, наш гид и интерпретатор, много знает о микропроцессоре 8088, расположенном внутри IBM PC. Мы будем использовать его, чтобы исследовать внутренние процессы, происходящие в 8088; и начнём с запроса к Debug, чтобы тот высветил всё, что он может сообщить о маленьких кусочках памяти, называемых регистрами, в которых могут храниться числа. Регистры похожи на переменные в Бейсике, но они не совсем то же самое. В отличие от Бейсика микропроцессор 8088 содержит ограниченное число регистров, и эти регистры не являются частью памяти вашего IBM PC.

Мы просим Debug показать содержимое регистров микропроцессора 8088 с помощью команды "R" (сокращение от англ. "Register"):

```
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 E485 IN AL,85
```

(Возможно, на своем дисплее вы увидите другие числа во второй и третьей строках, эти числа показывают количество памяти компьютера. Эти различия будут сохраняться и дальше, позже мы их обсудим.)

Сейчас Debug выдал нам достаточно много информации. Обратим внимание на первые четыре регистра, AX, BX, CX и DX, о значениях которых Debug сообщил, что они все равны 0000. Это регистры общего назначения. Остальные регистры SP, BP, SI, DI, DS, ES, SS, CS и IP являются регистрами специального назначения, с которыми мы будем иметь дело в следующих главах.

четырёхзначное число, показанное сразу вслед за именем регистра, является шестнадцатеричным. В

главе 1 мы узнали, что одно слово описывается ровно четырьмя шестнадцатеричными цифрами. Так что, как вы видите, каждый из 13 регистров микропроцессора 8088 является словом и имеет длину 16 бит. Поэтому компьютеры, созданные на основе микропроцессора 8088, называются шестнадцатеразрядными.

Мы уже отмечали, что регистры похожи на переменные Бейсика. Это означает, что мы можем изменять их состояние, и мы будем это делать. Команда Debug "R" не только высвечивает регистры. Если указать в команде имя регистра, то Debug поймет, что мы хотим взглянуть на содержимое именно этого регистра и может быть изменить его. Например, мы можем изменить регистр AX так, как это показано ниже:

```
-R AX
AX = 0000
:3A7
```

Давайте опять просмотрим содержимое регистров, чтобы убедиться в том, что в регистре AX теперь содержится 3A7h:

```
-R
AX=03A7 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 E485 IN AL,85
```

Так и есть. Итак, мы можем помещать шестнадцатеричное число в регистр с помощью команды "R", указывая имя регистра и вводя его новое значение после двоеточия, как мы только что сделали. В дальнейшем мы будем использовать эту команду для ввода чисел в регистры микропроцессора 8088.

Вы наверно помните, что уже видели число 3A7h в главе 1, когда мы применили команду Debug "H" для сложения 3A7h и 1EDh. Тогда, Debug делал работу за нас. Теперь же, мы будем использовать Debug больше как интерпретатор, чтобы работать непосредственно с микропроцессором 8088. Мы будем задавать 8088 инструкции на сложение чисел из двух регистров: сначала поместим число в регистр BX, затем дадим инструкцию 8088 прибавить число, хранящееся в BX, к числу, хранящемуся в AX, и поместить ответ обратно в AX. Для начала нам нужно ввести число в

регистр BX. Давайте сложим 3A7h и 92Ah. Используйте команду "R", чтобы поместить 92Ah в BX.

Память и микропроцессор 8088

Итак, регистры AX и BX должны, наверное, содержать 3A7h и 92Ah, в чем мы можем убедиться с помощью команды "R":

```
-R BX
BX 0000
:92A
```

```
-R
AX=03A7 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 E485 IN AL,85
```

Теперь, в регистрах AX и BX имеются нужные нам числа, каким образом мы сообщим микропроцессору 8088 о том, что их, то есть содержимое регистров BX и AX, нужно сложить? Для этого мы введём некоторые числа в память компьютера.

IBM PC имеет по меньшей мере 128Кбайт памяти - значительно больше, чем нам здесь потребуется. Мы поместим два байта машинного кода в одном из уголков этого огромного количества памяти. В этом случае машинный код будет представлен двумя двоичными числами, с помощью которых мы сообщим микропроцессору 8088 о том, что надо прибавить регистр BX к AX. Затем для того чтобы мы могли увидеть результат, мы выполним эту инструкцию с помощью программы Debug.

В какой именно части памяти мы должны поместить эту двухбайтную инструкцию, и как мы сообщим микропроцессору 8088 о том, где её искать? При включении микропроцессор 8088 разбивает память на части по 64Кбайт каждая, называемые сегментами. Чаще всего мы будем иметь дело с одним из этих сегментов, не зная реально, где именно в памяти он начинается. Мы можем поступать таким образом из-за способа, которым микропроцессор 8088 размечает память.

Все байты памяти помечаются числами, начиная с 0h и выше. Но помните четырёхзначное ограничение шестнадцатеричных чисел? Это означает, что наибольшее число, которое может использовать микропроцессор 8088 - это шестнадцатеричный эквивалент числа 65535. Это, в свою очередь, показывает, что максимальный объем памяти, размечаемый 8088, составляет 64Кбайт. Однако, как мы знаем из практики, микропроцессор 8088 может адресовать больше

64Кбайт памяти. Как он это делает? - С помощью небольшого трюка: он использует два числа, одно для номера 64Кбайт сегмента, второе для каждого байта, или смещения, внутри сегмента. При такой адресации 8088 может использовать до одного миллиона байт памяти.

Все адреса (метки), которые мы будем использовать в дальнейших примерах - это смещения от начала сегмента. Мы будем записывать адреса как номер сегмента и вслед за ним, через двоеточие, смещение внутри сегмента. Например, 3756:0100 будет означать, что мы находимся на смещении 100h внутри сегмента 3756h.

Позже в главе 11, мы изучим сегменты более подробно и посмотрим, почему мы имеем такой большой номер сегмента. Но сейчас мы просто доверим программе Debug следить за сегментами и не будем обращать внимание на их номера. Каждый адрес соответствует одному байту в сегменте, и адреса расположены в возрастающем порядке, так что 101h -байт следующий в памяти за 100h.

В записи наша двухбайтовая инструкция о сложении BX и AX будет выглядеть так: "ADD AX, BX". Мы поместим эту инструкцию по адресу 100H и 101H в любом сегменте, который Debug начнёт использовать. В соответствии с соглашением об адресации, оговоренном выше, мы будем говорить, что инструкция размещена по адресу 100h, так как это место размещения первого байта инструкции.

Команда Debug для исследования и изменения памяти называется "E" (от англ. "Enter"). Используйте эту команду для ввода двух байт инструкции ADD, как показано ниже:

```
Start of --->
segment 3756   .
               .
               .
3756:0100-->  01h   \  _  ADD      AX, BX
3756:0101-->  D8h   /
```

Рис.2. 1. Наша инструкция, начинающаяся с байта 100h от начала сегмента.

```
-E 100
3756:0100 E4.01
-E 101
3756:0101 85.08
```

Числа 01h и D8h расположены по адресам 3756:0100 и 3756:0101. Номер сегмента, который вы увидите, возможно, будет другим, но это различие не будет влиять на нашу программу. Как и в приведённом примере, Debug высветит два различных двухзначных числа в ответ на команды. Эти числа (E4h и 85h в нашем примере) - старые значения ячеек памяти на смещении 100h и 101h от начала сегмента, выбранного программой Debug, то есть эти числа - данные, оставшиеся в памяти от предыдущей программы после запуска Debug. (Если вы только что включили свой компьютер, то числа должны быть 00.)

Сложение, метод микропроцессора 8088

Теперь регистры должны выглядеть так:

```
AX=03A7 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 01D8 ADD AX,BX
```

Инструкция ADD помещена в память именно там, где мы хотели её разместить. Это видно из третьей строки сообщения. Первые два числа, 3756:0100, дают нам адрес (100h) первого числа нашей инструкции. За ними мы видим два байта, означающие ADD:01D8. Байт, равный 01 h, расположен по адресу 100h, а D8h - по адресу 101h. В конце строки располагается сообщение на машинном языке. Эту запись микропроцессор будет интерпретировать как инструкцию о сложении. После того, как инструкция размещена в памяти, необходимо сообщить микропроцессору 8088 о том, где она расположена.

Микропроцессор находит номер сегмента и адрес смещения в двух специальных регистрах, CS и IP, [!] CS:IP которые вы можете видеть распечатанными на предыдущем листинге. Номер сегмента хранится в CS, или сегменте кода (англ. "Code Segment"), регистре, который мы вскоре рассмотрим. Если вы посмотрите на

на распечатку регистров, то увидите, что Debug уже установил для нас CS (в нашем примере CS=3756). Таким образом, полный адрес начала нашей инструкции 3756:0100.

Следующая часть этого адреса (смещение внутри сегмента 3756) хранится в регистре IP - указателе инструкции (англ. "Instruction Pointer"). Микропроцессор 8088 использует смещение, взятое из регистра IP, чтобы найти нашу первую инструкцию. Мы можем сообщить микропроцессору, где её искать, записав в регистр IP адрес нашей первой инструкции IP = 0100.

Но регистр IP уже установлен в 100h. Мы немного [!] словчили: Debug устанавливает IP в 100h всякий раз, когда его запускают. Зная это, мы специально выбрали 100h адресом первой инструкции и таким образом освободились от необходимости выполнять установку регистра IP отдельно. Этот приём стоит запомнить.

После ввода инструкции и правильной установки регистров мы попросим Debug её выполнить. Для этого мы применим команду Debug "Т" (от англ. "Trace"), которая выполняет одну инструкцию за шаг и затем показывает содержимое регистров. После каждого запуска IP будет указывать на следующую инструкцию, в нашем случае будет указывать на 102h. Мы не помещали никакой инструкции в 102h, поэтому в последней строке распечатки мы увидим инструкцию, оставшуюся от предыдущей программы.

Давайте с помощью команды "Т" попросим Debug выполнить инструкцию:

```
-T
AX=0CD1 BX=092A CX=00000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0102 NV UP DI PL NZ NA PO NC
3756:0102 AC LODSB
```

Вот и всё. Регистр AX теперь содержит число CD1h, которое является суммой 3A7h и 92Ah. А регистр IP указывает на адрес 102h, так что в последней строке распечатки регистров мы видим инструкцию, расположенную в памяти по адресу 102h, а не по адресу 100h.

Как отмечалось ранее, указатель инструкции вместе с регистром CS всегда указывают на следующую инструкцию, которую нужно выполнить микропроцессору 8088. Если мы опять напечатаем "Т", то

выполнится следующая инструкция. Но не делайте этого сейчас - ваш микропроцессор 8088 может "зависнуть".

А что, если мы захотим выполнить введённую инструкцию ещё раз, то есть сложить 92Ah и CD1h и сохранить новый ответ в AX? Что нам надо сделать для того, чтобы объяснить микропроцессору 8088 где найти следующую инструкцию, и чтобы этой следующей инструкцией оказалась та же "ADD AX, BX", расположенная по адресу 100h. Можем ли мы изменить значение регистра IP на 100? Давайте попробуем. Используйте команду "R", чтобы установить IP в 100, и посмотрите распечатку регистров:

```
AX=0CD1 BX=092A CX=00000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 ADD AX,BX
```

Попробуйте ещё раз ввести команду "T" и посмотрите, содержит ли регистр AX число 15FBh. Действительно содержит.

Как видите, перед тем, как использовать команду "T", вам необходимо проверить регистр IP и соответствующую его значению инструкцию, располагаемую в нижней части листинга, показываемого командой "R". Таким образом, вы будете уверены, что микропроцессор 8088 выполняет нужную инструкцию.

А сейчас, установите регистр IP в 100h, убедитесь, что в регистрах содержится AX=15FBh, BX=092Ah, и снова попробуйте произвести вычитание.

```
AX:03A7    BX:092A
      \
[ IP:100 ] > ADD    AX,BX
----- /  LODSB
```

Рис.2.2. Перед выполнением инструкции сложения

```
AX:03A7    BX:092A
(?)  _____ \
[pause]
[ IP:100 ] > ADD    AX,BX
----- /  LODSB
```

Рис.2.3. После выполнения инструкции сложения

Вычитание, метод микропроцессора 8088

Мы собираемся написать инструкцию для вычитания BX из AX, так что после двух вычитаний в регистре AX появится результат 3A7h. Тогда мы вернёмся к той точке, с которой начали. Кроме того, вы увидите, каким образом можно немного сэкономить усилия при вводе двух байтов в память.

Когда мы вводили два байта инструкции ADD, то печатали команду "E" дважды: один раз с 0100h для первого адреса и второй раз с 0101h для второго адреса. Однако мы можем ввести второй байт и без использования еще одной команды "E", если мы отделим второй байт от введенного пробелом. После окончания ввода нажмите клавишу "Enter". Попробуйте этот метод на нашей инструкции вычитания:

```
-E 100  
3756:0100 01.29 D8.D8
```

Листинг регистров (помните о необходимости установки регистра IP в 100h) должен теперь показать инструкцию "SUB AX,BX", которая вычитает содержимое регистра BX из регистра AX и размещает результат в AX. Порядок записи AX и BX может быть разным, но инструкция, как и выражение $AX = AX - BX$, написанное на Бейсике, предполагает, что микропроцессор 8088, в отличие от Бейсика, всегда помещает ответ в первую переменную (в первый регистр).

Выполните эту инструкцию с помощью команды "T". AX должен содержать CD1. Измените IP так, чтобы он указывал на эту инструкцию, и выполните её опять (не забывайте сначала проверять инструкцию внизу листинга регистров). AX теперь должен содержать 03A7h.

Отрицательные числа в микропроцессоре 8088

В последней главе мы узнали, как микропроцессор 8088 использует форму двоичного дополнения для отрицательных чисел. Сейчас мы поработаем непосредственно с инструкцией SUB, чтобы проводить вычисления с отрицательными числами. Давайте дадим микропроцессору 8088 небольшой тест, чтобы

посмотреть, получим ли мы FFFFh в качестве -1. Мы вычтем единицу из нуля, и, если мы были правы, то в результате вычитания в регистре AX должно оказаться FFFFh (-1). Установите значение AX равным нулю и BX равным единице, затем запустите инструкцию по адресу 100h. Мы получили то, что ожидали: AX = FFFFh.

Байты в микропроцессоре 8088

До этого вся наша арифметика совершалась над словами, то есть четырьмя шестнадцатеричными цифрами. Знает ли микропроцессор 8088, как выполнять математические операции над байтами? Да, знает.

Так как одно слово состоит из двух байт, каждый регистр общего назначения может быть разделён на два байта, известных как старший байт (первые две шестнадцатеричные цифры) и младший байт (следующие две шестнадцатеричные цифры). Название каждого из полученных регистров складывается из первой буквы названия регистра (от "A" до "D"), стоящей перед "X" в слове, и буквы "H" для старшего байта или буквы "L" для младшего. Например, DL и DH - регистры длиной в байт, а DX - длиной в слово. (Применяемая здесь терминология не всегда удобна, так как в словах, хранящихся в памяти, младший байт идет первым, а старший вторым.)

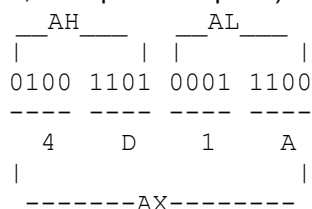


Рис.2-4. Разбиение регистра AX на два байтовых регистра (AH и AL)

Проверим байтовую арифметику на инструкции ADD. Введите два байта 00h и C4h, начиная с адреса 0100h. Внизу листинга регистров вы увидите инструкцию "ADD AH,AL", которая суммирует два байта регистра AX и поместит результат в старший байт AH.

Затем загрузите в регистр AX число 0102H. Таким образом вы поместите 01h в регистр AH и 02h в регистр AL. Установите регистр IP в 100h, выполните команду "T", и вы увидите, что регистр AX теперь

содержит 0302. Результат сложения 01h + 02h будет 03h, и именно это значение находится в регистре AH.

Но предположим, что вы не собирались складывать 01h и 02h. Допустим, на самом деле вы хотели сложить 01h и 03h. Если регистр AX уже содержит 0102, можем ли мы изменить значение регистра AL на 03h? Нет. Вам придётся изменять значение регистра AX на 0103h. Почему? Потому что Debug позволяет нам изменять только шестнадцатиразрядные регистры. Невозможно изменить только младшую или только старшую часть регистра с помощью Debug. Но, как вы видели в предыдущей главе, это не проблема. Имея дело с шестнадцатеричными числами, мы можем разделить слово на два байта, разбив четырёхзначное число пополам. Так что слово 0103h становится двумя байтами 01h и 03h.

Чтобы испытать в действии эту инструкцию сложения, загрузите в регистр AX число 0103h. Инструкция "ADD AH,AL" по-прежнему находится в памяти по адресу 0100h, так что установите регистр IP в 100h и, имея в регистрах AH и AL значения 01h и 03h, запустите выполнение инструкции. После этого AX будет содержать 0403h: 04h, это сумма 01h + 03h, находится в регистре AH.

Умножение и деление, метод микропроцессора 8088

Мы видели, как микропроцессор 8088 складывает и вычитает два числа. Теперь мы увидим, что он может также умножать и делить. Инструкция умножения называется "MUL", а машинный код для умножения AX на BX - F7h E3h. Мы введём его в память, но сначала несколько слов об инструкции MUL.

Где инструкция MUL сохраняет ответ? В регистре AX? Не совсем; здесь надо быть аккуратными. Как вы скоро увидите, умножение двух 16-битных чисел может дать 32-разрядный ответ, так что инструкция MUL сохраняет результат в двух регистрах, DX и AX. Старшие 16 бит помещаются в регистр DX, а младшие - в AX. Мы можем также записать эту комбинацию регистров как DX:AX.

Давайте вернёмся к Debug и к микропроцессору 8088. Введите инструкцию умножения F7h E3h по адресу 0100h, как вы это делали для инструкций сложения и вычитания, и установите AX = 7C4Bh и BX=100h. Вы увидите инструкцию в листинге регистров как "MUL BX", без всяких ссылок на регистр AX. При умножении

слов, микропроцессор 8088 всегда умножает регистр, имя которого вы указываете в инструкции, на регистр AX, и сохраняет ответ в паре регистров DX:AX.

Перед тем, как мы запустим эту инструкцию умножения, давайте произведём умножение вручную. Как мы можем подсчитать $100h * 7C4Bh$? Три цифры 100 имеют в шестнадцатеричной системе такой же эффект, как и в десятичной, так что умножение на 100h просто добавит два нуля справа от шестнадцатеричного числа. Таким образом, $100h * 7C4Bh = 7C4B00h$. Этот результат слишком длинен для того, чтобы поместиться в одном слове, поэтому мы разбиваем его на два слова 007Ch и 4B00h.

Используйте Debug для запуска инструкции. Вы увидите, что DX содержит слово 007Ch, и AX содержит слово 4B00h. Другими словами, микропроцессор 8088 возвращает результат инструкции умножения слов в паре регистров DX:AX. Так как результат умножения двух слов не может быть длиннее двух слов, но часто бывает длиннее одного слова (как мы только что видели), инструкция умножения слов всегда возвращает ответ в паре регистров DX:AX.

А как насчёт деления? Когда мы делим числа, микропроцессор 8088 сохраняет как результат, так и остаток от деления. Посмотрим на выполнение деления в 8088.

```
DX      AX      BX
0000  7C4B      0100

-----\
[IP 100 > MUL BX
-----/  LODSB
```

Рис.2.5.Перед выполнением инструкции умножения

```
DX      AX      BX
007C  4B00      0100

-----\  MUL BX
[IP 102 > LODSB
-----/
```

Рис.2.6. После исполнения инструкции умножения

DX	AX	BX
007C	4B12	0100

```

-----\
[IP 100 > DIV BX
-----/  LODSB

```

Рис.2.7. Перед выполнением инструкции деления

DX	AX	BX
0012	7C4B	0100

```

-----\  MUL BX
[IP 102 > LODSB
-----/

```

Рис.2.8. После исполнения инструкции деления

Поместим инструкцию F7h F3h по адресу 0100h (и 101h). Как и инструкция MUL, DIV использует пару регистров DX:AX, не сообщая об этом, так что всё, что мы видим - это "DIV BX". Загрузим в регистры значения: DX = 007Ch и AX = 4B12h; регистр BX по-прежнему должен содержать 0100h.

Подсчитаем результат вручную:
 $7C4B12h / 100h = 7C4Bh$ с остатком 12h. После выполнения инструкции деления по адресу 0100h мы получим для AX = 7C4Bh результат нашего деления и для DX = 0012h, остаток. (Мы найдём этому остатку очень хорошее применение в главе 10, когда будем писать программу перевода десятичных чисел в шестнадцатеричные).

Итог

Почти настало время написания нами реальной программы, которая будет печатать символ на экране. Время изучения основ уже прошло. Оглянемся назад, оценим то что мы уже изучили и затем поспешим вперёд.

Мы начали эту главу с изучения регистров и их сходства с переменными в Бейсике. Однако мы увидели, что в отличие от Бейсика, микропроцессор 8088 имеет ограниченное число регистров. Мы сконцентрировали наше внимание на четырёх регистрах общего назначения после быстрого обзора регистров

2*

CS и IP, используемых для размещения номера сегмента и адреса смещения.

Узнав, как изменять и считывать регистры, мы занялись созданием небольших программ, состоящих из одной инструкции, вводя машинные коды для сложения, вычитания, умножения и деления двух чисел с помощью регистров AX и BX. В последующих главах мы будем использовать многое из того, чему мы здесь научились, но вам не нужно запоминать машинные коды каждой инструкции.

Мы также узнали о том, как сообщить Debug о необходимости выполнить (протрассировать) одну инструкцию. Конечно, по мере того как программы будут расти, их трассировка будет становиться как более полезной, так и более утомительной процедурой. Позже мы научимся тому, каким образом можно выполнить несколько инструкций одной командой Debug.

Сейчас мы вернемся к программам и узнаем, как делать программу, которая может выдавать сообщения.

Глава 3. Вывод символов на экран

Теперь мы узнали достаточно для того, чтобы сделать что-либо более основательное. Начнём с того, что заставим DOS вывести символ на экран, а затем займёмся ещё более интересной работой - создадим программу, состоящую из нескольких инструкций, и с её помощью научимся ещё одному способу записи данных в регистры. Посмотрим, сумеем ли мы достигнуть того, чтобы DOS заговорил.

INT - мощное прерывание

К четырём математическим инструкциям ADD, SUB, MUL и DIV мы добавим новую инструкцию, называемую "INT" (от англ. "Interrupt" - прерывание). INT немного похожа на оператор Бейсика GOSUB. Мы будем использовать инструкцию INT для того, чтобы заставить DOS напечатать символ на экране.

Перед тем, как мы узнаем о том, как работает INT, рассмотрим один пример. Запустите Debug и поместите 200h в AX и 41h в DX. Инструкция INT для функций DOS имеет вид "INT 21h", в машинном коде

CDh 21h. Как и инструкция DIV из последней главы, это двухбайтная инструкция. Поместите "INT 21h" в память, начиная с адреса 100h, и используйте команду "R", чтобы убедиться, что инструкция читается как "INT 21h" (не забудьте установить IP в 100h, если вы этого ещё не сделали).

Теперь мы готовы выполнить эту инструкцию, но не сможем, как в предыдущей главе, использовать команду трассировки, потому что она выполняет одну инструкцию за шаг, а инструкция INT вызывает большую программу из DOS, выполняющую некоторые действия. При этом INT работает почти так же, как программы в Бейсике, которые могут вызывать подпрограммы с помощью оператора GOSUB.

Мы не собираемся выполнять каждую из инструкций целой "подпрограммы" DOS с помощью трассировки. Напротив, мы хотим запустить нашу программу, состоящую из одной строки, но остановиться перед выполнением инструкции, размещенной по адресу 102h. Мы можем это сделать с помощью команды Debug "G" (сокр. от англ. "Go"), после которой пишется адрес, на котором мы хотим остановиться:

```
-G 102
AX=0241 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0102 NV UP DI PL NZ NA PO NC
3970:0102 8BE5 MOV SP,BP
```

DOS напечатал букву "A" и затем возвратил управление в нашу программу (учтите, что инструкция, размещенная по адресу 102h, является данными, оставшимися от другой программы, так что последняя строка вашего листинга может выглядеть по другому.)

Программа в некотором смысле состоит из двух инструкций, вторая из которых расположена по адресу 102h, то есть имеется что-то вроде этого:

```
INT 21
MOV SB,BP
```

(либо что-то похожее на вашем компьютере)

В дальнейшем мы заменим эту случайную вторую инструкцию на одну из уже известных, но сейчас мы предложили Debug запустить нашу

программу, остановить выполнение, когда она дойдет до этой второй инструкции, и показать содержимое регистров, что и было сделано.

А как DOS узнал, что нужно печатать "A"? Значение 02h в регистре AH говорит DOS, что надо напечатать некоторый символ. Иное значение этого регистра сообщает DOS о том, что нужно выполнить другую функцию. (Часть функций мы рассмотрим позже, но если вам любопытно, вы можете найти список этих функций в руководстве по DOS.)

Для самого же символа DOS использует число в регистре DL, рассматриваемое как ASCII код символа, выводимого на экран. Для вывода на экран заглавной буквы "A" в регистр DL заносится значение 41h, ASCII код заглавной буквы "A".

В Приложении E вы найдете таблицу кодов символов ASCII для всех символов, которые может вывести на экран ваш IBM PC. Значения кодов представлены как в десятичном, так и в шестнадцатеричном виде. В связи с тем, что Debug понимает только шестнадцатеричные числа, работа с выводом символов на экран является хорошей практикой перевода десятичных чисел в шестнадцатеричные. Возьмите символ из таблицы и переведите самостоятельно его код в шестнадцатеричную форму. Теперь проверьте свой перевод, введя полученное шестнадцатеричное число в регистр DL и снова запустив инструкцию INT (помните о необходимости установки IP в 100h).

Результат применения команды трассировки к инструкции INT может вызвать удивление своей непредсказуемостью. Допустим, мы не исполняли команды "G 102", а произвели трассировку для того, чтобы увидеть, что же произойдет. Если вы решили попробовать выполнить такую процедуру самостоятельно, то не заходите слишком далеко, так как можете увидеть, что ваш компьютер начнет вытворять нечто странное. После того как вы протрассируете несколько шагов, выйдете из Debug с помощью команды "Q". Это ликвидирует беспорядок, который вы за собой оставите.

```
-R
AX=0200 BX=092A CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100 NV UP DI PL NZ NA PO NC
```

```
3970:0100 CD21 INT 21
-T
AX=0200 BX=092A CX=0000 DX=0041 SP=FFE8 BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3372 IP=0180 NV UP DI PL NZ NA PO NC
3372:0180 80FC4B CMP AH,4B
-T
AX=0200 BX=092A CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3372 IP=0183 NV UP DI PL NZ NA PO NC
3372:0183 7405 JZ 018A
-T
AX=0200 BX=092A CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3372 IP=0185 NV UP DI PL NZ NA PO NC
3372:0185 2E CS:
3372:0186 FF2EAB0B JMP FAR [0BAB] CS:0BAB = 0BFF
```

Обратите внимание на то, что первое число, являющееся составляющей адреса, изменилось с 3970 на 3372. Три последние инструкции являются частью DOS, а программа для DOS находится в другом сегменте. Фактически существует очень много инструкций, которые DOS выполняет перед тем, как напечатать символ; поэтому даже такая, кажущаяся простой задача не так проста. Теперь стало ясно, почему мы использовали команду "G" для выполнения программы только до адреса 102h. В противном случае мы бы увидели лавину инструкций от DOS. (Если у вас иная версия DOS, то инструкции, которые вы увидите при трассировке, могут быть другими.)

Грациозный выход - INT 20h

Помните, что наша инструкция INT была 21h? Если мы изменим 21h на 20h, мы получим "INT 20h", другую инструкцию прерывания, которая сообщает DOS о том, что мы хотим выйти из нашей программы и чтобы управление опять вернулось к DOS. В нашем случае "INT 20h" вернёт управление к Debug, так как мы выполняем нашу программу из Debug, а не из DOS.

Введите инструкцию CDh 20h, начиная с адреса 100h, а затем попробуйте проделать следующее (только не забудьте проверить инструкцию "INT 20h" с помощью команды "R"):

```
-G 102
Program terminated normally

-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100 NV UP D1 PL NZ NA PO NC
3970:0100 CD20 INT 20

-G
Program terminated normally
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100 NV UP D1 PL NZ NA PO NC
3970:0100 CD20 INT 20
```

Команда "G" выполняет всю программу (которая сейчас состоит всего из одной инструкции, так как "INT 20" - это инструкция выхода), и затем возвращается к началу. Когда мы начали выполнение, IP был установлен в 100h. Значения регистров в этом примере равны 0 только потому, что мы заново запустили Debug.

Мы можем ввести инструкцию INT 20h в конец программы так, чтобы грациозно передать управление DOS (или Debug) и поэтому сейчас введём эту инструкцию вместе с "INT 21h" в программу, состоящую из двух строк.

Программа из двух строк - соединение частей вместе

Начиная с адреса 100h, введите две инструкции "INT 21h", "INT 20h" (CDh 21h CDh 20h) одну за другой (мы теперь всегда будем начинать программы с адреса 100h).

Когда у нас была только одна инструкция, то мы могли "пролистать" эту инструкцию командой "R", но теперь у нас две инструкции. Чтобы увидеть их, у нас есть команда "U" (от англ. "Unassemble"), которая работает аналогично команде Бейсика "List":

```
-U 100
3970:0100 CD21          INT  21
3970:0102 CD20          INT  20
3970:0104 D98D460250B8 ESC  09, [DI+0246] [DI+B850]
3970:010A 8D00          LEA  AX, [BX+SI]
3970:010C 50            PUSH AX
3970:010D E82A23        CALL 243A
3970:0110 8BE5          MOV  SP, BP
3970:0112 8EC41A        ADD  SP, + 1A
3970:0115 5D            POP  BP
3970:0116 C3            RET
3970:0117 55            PUSH BP
3970:0118 83EC02        SUB  SP, + 02
3970:011B 8BEC          MOV  BP, SP
3970:011D 823E0E0000    CMP  BYTE PTR (000E), 00
```

Первые две инструкции, которые представлены в списке, являются инструкциями, которые мы только что ввели. Остальные инструкции остались в памяти от предыдущих программ. По мере того как наша программа будет расти, количество выводимых на экран кодов также будет увеличиваться.

Поместите в регистр АН значение 02h, а в регистр DL код любого символа (как вы это делали раньше, когда изменяли регистры АХ и DX), и затем напечатайте команду "G", чтобы увидеть ваш символ на экране. Например, если вы поместили в DL число 41 h, то вы увидите:

```
-G
A
Program terminated normally
-
```

Попробуйте вывести на экран еще несколько символов, пока мы не перейдем к изучению другого способа установки этих регистров.

Ввод программ

Начиная с этой страницы, большая часть программ будет иметь в длину более чем одну инструкцию, и чтобы просмотреть эти программы, мы будем использовать инструкцию разассемблирования ("U"), поэтому наша последняя программа появится в виде:

```
3970:0100 CD21 INT  21
3970:0102 CD20 INT  20
```

До этого мы вводили инструкции программ в виде чисел, например CDh, 21h. Но это слишком тяжёлая работа, и как оказывается, имеется более простой способ ввода инструкций.

В дополнение к команде разасемблирования в программе Debug имеется команда, позволяющая вводить мнемонические, или человекочитаемые, инструкции. Так что вместо того, чтобы вводить непонятные числа программы, мы можем применить команду ассемблирования для следующего ввода:

```
-A 100
3970:0100 INT 21
3970:0102 INT 20
3970:0104
-
```

После ввода инструкций необходимо нажать клавишу "Enter", и вновь появится приглашение Debug.

Команда "A" сообщает Debug о том, что мы хотим ввести инструкции в мнемонической форме, а число 100 в команде означает, что ввод инструкций начнётся с ячейки 100H. Команда ассемблирования значительно упрощает ввод программ.

Использование команды MOV для пересылки данных между регистрами

Хотя раньше мы во всем полагались на Debug, мы не всегда будем запускать программы с её помощью. Обычно программа сама устанавливает регистры AH и DL перед инструкцией "INT 21h". Чтобы уметь это делать, мы изучим ещё одну инструкцию, MOV. Применение инструкции MOV позволит создавать программы, способные запускаться непосредственно из DOS. В дальнейшем мы будем использовать инструкцию MOV для того, чтобы загружать числа в регистры AH и DL. Начнём изучение MOV с осуществления пересылки чисел между регистрами. Поместите 1234h в AX (12h в регистр AH и 34h в AL) и ABCDh в DX (ABh в DH и CDh в DL). С помощью команды "A" введите инструкцию:

```
396F:0100 88D4 MOV AH,DL
```

Эта инструкция пересылает число из DL в AH, копируя его в AH, AL при этом не используется. Если

Использование команды MOV_
для пересылки данных между регистрами_

вы протрассируете эту строку, то увидите, что AX = CD34h и DX = ABCDh. Изменился только AH. Теперь он содержит копию числа из DL.

Как и оператор Бейсика "LET AH=DL", инструкция MOV пересылает число из второго регистра в первый, и по этой причине мы пишем AH перед DL. Несмотря на то, что имеются некоторые ограничения, о которых мы поговорим позже, мы можем применить иные формы инструкции MOV, для копирования чисел между парами регистров. Например, переустановите IP и попробуйте ввести следующее:

```
396F:0100      89C3      MOV     BX,AX
```

Вы только что загрузили из регистра в регистр слово, а не байт. Инструкция MOV всегда копирует или слова или байты, но никогда слова в байты. Это вполне понятно: как вы загрузите слово в байт?

В начале мы установили загрузку числа из регистра AH в регистр DL. Проделаем то же самое с другой формой инструкции MOV:

```
396F:0100      B402      MOV     AH,02
```

Эта инструкция загружает число 02h в регистр AH без применения регистра AL. Второй байт инструкции, 02h является числом, которое мы хотим загрузить. Попробуйте загрузить в AH другое число: с помощью команды "E 101" измените второй байт, чтобы он был равен, например, C1h.

Сложим все части вместе и построим длинную программу. Она будет печатать звездочку, "*", выполняя все операции сама, не требуя от нас установки регистров (AH и DL). Программа использует инструкции MOV для того, чтобы установить регистры AH и DL перед вызовом INT 21 h из DOS:

```
396F:0100  B402      MOV     AH,02
396F:0102  B22A      MOV     DL,2A
396F:0104  CD21      INT     21
396F:0106  C020      INT     20
```

Введите программу и проверьте её командой "U" ("U100"). Убедитесь, что IP указывает на ячейку

100h, затем попробуйте командой "G" запустить её. В итоге на вашем экране должен появиться символ "**".

```
-G
*
Program terminated normally
*
```

У нас есть законченная, содержательная программа, запишем её на диск в виде .COM файла для того, чтобы мы могли запускать её прямо из DOS. Это можно сделать просто набрав её имя. Так как у программы нет имени, то мы должны его присвоить.

Команда Debug "N" (сокр. от англ. "Name") присваивает файлу имя перед записью на диск. Напечатайте:

```
-N WRITESTR.COM
```

Эта команда не запишет файл на диск - она только назовёт его WRITESTR.COM.

Далее мы должны сообщить Debug о том, сколько байт занимает программа для того, чтобы он знал размер файла. Если вы посмотрите на разассемблированный листинг программы, то вы увидите, что каждая инструкция в нём занимает два байта (что в общем не всегда верно). У нас четыре инструкции, следовательно программа имеет длину $4 \times 2 = 8$ байт. (Мы могли бы также задействовать команду Debug "H". Напечатав "H 108 100", где 108 -адрес инструкции после INT 20, мы получим 8.)

Так как у нас теперь есть число байт, то нам надо куда-нибудь его записать. Debug для этого использует пару регистров BX:CX, и поэтому, поместив 8h в CX, мы сообщим Debug о том, что программа имеет длину в восемь байт. BX должен быть предварительно установлен в ноль.

После того как мы установили имя и длину программы, мы можем записать её на диск с помощью команды Debug "W" (от англ. "Write"):

```
-W
Writing 0008 bytes
-
```

Теперь на диске есть программа WRITESTR.COM, и мы сейчас с помощью "Q" покинем Debug и посмотрим на неё. Используйте команду DOS DIR, чтобы увидеть файл:

```
A:\>DIR WRITESTR.COM

Volume in drive A has no label Directory of A:\

WRITESTR.COM      8      6-30-83      10:05a
      1 File(s) 18432 bytes free
A:\>
```

Листинг директории сообщает, что WRITESTR.COM находится на диске и его длина составляет восемь байт, как и должно быть. Чтобы запустить программу, напечатайте "Writestr" в ответ на приглашение DOS и нажмите "Enter". Вы увидите "**", появившуюся на дисплее. Вот и всё.

Вывод на экран строки символов

В качестве последнего примера в этой главе мы используем "INT 21 h" с другим номером функции в регистре AH для того, чтобы вывести на экран целую строку символов. Мы сохраним эту строку в памяти и затем сообщим DOS, где её искать, так что в процессе работы над такой программой мы познакомимся более близко с понятиями адреса и памяти. Как мы уже отметили, функция номер 02h для прерывания "INT 21h" печатает один символ на экране. Другая функция, номер 09h, печатает целую строку и прекращает печать, когда находит символ "\$". Давайте поместим строку в память. Мы начнём с ячейки 200h, так что запись строки не перепутается с кодом самой программы. Введите следующие числа, используя инструкцию "E 200":

```
48  65  6C  6C
6F  2C  20  44
4F  53  20  68
65  72  65  2E
24
```

Последнее число 24h является ASCII-кодом для знака "\$", и оно сообщает DOS, что это конец строки символов. Через минуту вы увидите, что сообщает эта строка, запустив программу, которую сейчас введете:

```
396F:0100      B409      MOV AH,09
396F:0102      BA0002    MOV DX,0200
```

```
396F:0105 CD21 INT 21
396F:0107 CD20 INT 20
```

200h это адрес строки, которую мы ввели, а загрузка 200h в регистр DX сообщает DOS о том, где её искать. Проверьте программу командой "U" и затем запустите её командой "G":

```
-G
Hello, Dos here
Program terminated normally
```

Мы сохранили в памяти несколько введённых символов, и теперь самое время познакомиться с другой командой Debug, "D" (от англ. "Dump"). То, как эта команда дампирует (выводит содержимое) памяти на экран, похоже на действия, совершаемые командой "U" при распечатке инструкций. Также, как и при использовании команды "U", поместите адрес после "D", чтобы сообщить Debug, откуда начинать дамп. Например, команда "D 200" выводит содержимое участка памяти, в котором хранится только что введённая строка.

```
-D 200
396F:0200 48 65 6C 6C 6F 2C 20 44-4F 53 20 68 65 72 65 2E
Hello, Dos here
396F:0210 24 5D C3 55 83 EC 30 8B-EC C7 06 10 00 00 00 E8
$JCU.10.IG.....h
.
.
.
```

После каждого числа, обозначающего адрес (как 396F:0200 в нашем примере), мы видим 16 шестнадцатеричных байт, вслед за которыми записаны 16 ASCII-символов для этих байт. Так например, в первой строке вы видите почти все ASCII-коды и символы, которые вы ввели. Символ "\$" является первым символом в следующей строке, остальная часть строки представляет собой беспорядочный набор символов.

Когда вы видите точку (".") в окне ASCII, знайте, что это может быть как точка, так и специальный символ, например, греческая буква "pi". Команда Debug "D" выдаёт только 96 из 256 символов символического набора IBM PC, поэтому точка используется для обозначения остальных 160 символов.

В дальнейшем мы будем использовать команду "D" для проверки чисел, введенных в качестве данных, независимо от того, являются ли эти данные символами или обычными числами. (Если вам необходима более глубокая информация на эту тему, смотрите ту часть руководства по DOS, которая посвящена Debug.)

Программа, выводящая строку на экран, закончена, так что мы можем записать её на диск. Процедура записи та же, которую мы использовали для записи на диск WRITESTR.COM, за исключением того, что на этот раз нам нужно установить значение длины программы, достаточное для того, чтобы включить строку по адресу 200h. Программа начинается со строки 100h, и из только что выполненного дампа памяти можно видеть, что символ, следующий за знаком "\$", заканчивающим нашу строку, расположен по адресу 211h. Как и раньше, мы можем использовать команду "H", чтобы определить разность между этими двумя числами. Найдите 211h-100h и сохраните это значение в регистре CX, опять установив BX в ноль. Используйте команду "N", чтобы дать имя программе (добавьте расширение .COM, чтобы запускать программу прямо из DOS), и затем командой "W" запишите программу и данные в дисковый файл. Вот и все о выводе символов на экран, кроме одного последнего замечания: вы должны помнить, что DOS не печатает символа "\$". Это происходит потому, что DOS использует этот знак для пометки конца строки символов. Следовательно, мы не можем прямо использовать DOS, чтобы напечатать строку, содержащую "\$", однако в последующих главах мы увидим, как обойти это препятствие.

Итог

Сведения, полученные в первых двух главах, позволили нам работать над настоящей программой. В этой главе мы использовали знание шестнадцатеричных чисел, программы Debug, инструкций микропроцессора 8088 и памяти для того, чтобы создать программы, способные выводить на экран символ и их строку. Во время этого процесса мы заодно узнали ряд новых вещей.

Прежде всего, мы узнали об инструкциях INT, не очень много, но достаточно для того, чтобы написать две небольшие программы. В последующих главах,

одновременно с расширением представления о микропроцессоре, мы углубим также знания об инструкциях прерываний. Debug, как и раньше, был преданным и полезным гидом. Мы возложили на него обязанности распечатки содержимого регистров и памяти, и кроме этого Debug запускал созданные программы с помощью команды "G".

Мы также узнали об инструкции выхода "INT 20" и инструкции копирования чисел между регистрами MOV. Инструкция выхода ("INT 20") позволила нам создать законченную программу, которую мы затем записали на диск с помощью Debug и запустили из DOS. Инструкция MOV дала возможность установить значения регистров перед инструкцией печати "INT 21" и мы смогли написать самостоятельную программу, выводящую на экран один символ.

Глава заканчивается применением функции "INT 21h", для печати целой строки символов. Отметим удобное дополнение работы с Debug - возможность ввода инструкций в виде мнемонических кодов, облегчающих восприятие текстов программ. Теперь у нас достаточно знаний, чтобы приступить к выводу на экран двоичных чисел. В следующей главе мы напишем программу, которая будет брать один байт памяти и печатать его на экране в виде строки двоичных цифр (единиц и нулей).

Глава 4. Вывод на экран двоичных чисел

В этой главе мы создадим программу, которая будет выводить на экран двоичные числа в виде единиц и нулей. У нас есть почти все знания, которые нам нужны, и работа, которой мы сейчас займёмся, поможет подтвердить ранее выдвинутые идеи. Мы также добавим ряд инструкций к тем, с которыми уже познакомились, включая новую версию ADD и инструкции, которые должны нам помочь в повторении частей нашей программы. Мы познакомимся также с некоторыми принципиально новыми понятиями.

Циклический сдвиг и флаг переноса

В главе 2, когда мы впервые столкнулись с шестнадцатеричной арифметикой, мы узнали, что сложение 1 и FFFFh должно дать 10000h, но этого не происходит. Только четыре шестнадцатеричных цифры справа помещаются в одно слово. Мы также узнали, что 1 (для числа 10000h) это переполнение, и что она не теряется. Куда она попадает? Она помещается в ячейку, называемую флагом - в данном случае, флагом переноса, в сокращении CF (от англ. "Carry Flag"). Флаги содержат числа, состоящие из одного бита, так что они могут содержать либо единицу, либо ноль. Если в пятом разряде шестнадцатеричного числа появляется единица (происходит переполнение), то она попадает во флаг переноса.

Вернёмся к инструкции ADD из главы 3 ("ADD AX,BX"). Поместите FFFFh в AX и 1 в BX, затем протрассируйте инструкцию ADD. В конце второй строки распечатки, полученной с помощью команды Debug "R", вы увидите восемь пар чисел. Последняя из них, которая может выглядеть как NC или CY, и является флагом переноса. Из-за того, что инструкция сложения дала переполнение в виде 1, состояние флага выглядит как CY (от англ. "Carry"). Бит переноса теперь равен 1, или, как мы будем говорить, флаг установлен.

Чтобы убедиться в том, что мы записываем в нём семнадцатый бит (при сложении байт это будет девятый бит), прибавьте единицу к нулю в AX, установив IP в 100h и повторив трассировку инструкции сложения. Флаг переноса переустанавливается в каждой операции сложения, и так как на этот раз переполнения не будет, то флаг будет сброшен. Действительно, перенос равен нулю, что видно из состояния флага, которое мы можем наблюдать с помощью все той же команды "R": NC (от англ. "No Carry" - нет переноса).

(Об остальных флагах состояния мы узнаем позже, но если вам любопытно, вы можете найти информацию о них прямо сейчас в разделе руководства по DOS, посвященному команде Debug "R".)

Рассмотрим задачу печати двоичного числа, чтобы увидеть, как нам может пригодиться информация о переносе. За шаг мы печатаем только один символ, и нам надо произвести выборку всех битов нашего числа, одного за другим, слева направо. Например, первое число, которое мы будем печатать, это 1000 0000h. Если мы сдвинем весь этот байт влево на

одну позицию, помещая единицу во флаг переноса и добавляя ноль справа, и затем повторим этот процесс для каждой последующей цифры, во флаге переноса будут по очереди содержаться все цифры нашего двоичного числа. И мы можем это сделать с помощью новой инструкции, называемой RCL (англ. "Rotate Carry Left" - циклический сдвиг влево с переносом).

(StIV) : "Rotate Left Through Carry"

Чтобы увидеть, как она работает, введите программу:

```
3985:0100 D0D3 RCL BL,1
```

Эта инструкция циклически сдвигает байт в BL влево на один бит (то есть 1), и делает это через флаг переноса. Инструкция названа циклическим сдвигом, так как RCL сдвигает крайний левый бит во флаг переноса, в то время как бит, находившийся до этого во флаге переноса, сдвигается в крайне правую позицию (т.е. в нулевой бит). В процессе сдвига все остальные биты сдвигаются влево. После определённого количества циклических сдвигов (17 для слова, 9 для байта) биты возвращаются на их начальные позиции, и вы получаете исходное число.

Поместите B7h в регистр BX и протрассируйте инструкцию циклического сдвига несколько раз. Если перевести полученные вами результаты в двоичную форму, то вы увидите следующее:

Carry BL register

0	1 0 1 1 0 1 1 1	B7h	Исходное состояние
1	0 1 1 0 1 1 1 0	6Eh	
0	1 1 0 1 1 1 0 1	DDh	
1	1 0 1 1 1 0 1 0	BAh	
1	0 1 1 1 0 1 0 1	75h	
0	1 1 1 0 0 0 1 1	ebh	
1	1 1 0 1 0 1 1 0	d6h	
1	0 0 1 0 0 1 0 1	adh	
1	0 1 0 1 0 0 1 1	5bh	
0	1 0 1 1 0 1 1 1	B7h	После 9 циклов

Во время первого циклического сдвига бит 7 регистра BL сдвигается во флаг переноса, а остальные биты - на одну позицию влево. Дальнейшее повторение этой операции будет сдвигать все биты влево до тех пор, пока после девяти сдвигов в регистре BL не появится исходное число.

Мы уже скоро приступим к написанию программы распечатки (вывода на экран) двоичных чисел, но сначала нам надо узнать ещё кое-что. Давайте

посмотрим, как мы можем превратить бит во флаге переноса в символ "1" или "0".

Сложение с использованием флага переноса

Обычная инструкция сложения, например, "ADD AX,BX", просто сложит два числа. Другая инструкция ADC (англ. "Add with Carry" - сложение с переносом), складывает три числа: два числа, как и раньше, ПЛЮС один бит из флага переноса. Если вы посмотрите на таблицу кодов ASCII, вы увидите, что 30h это символ "0", а 31h - символ "1". Таким образом, сложение флага переноса и 30h даст нам символ "0", когда этот флаг сброшен, и "1", когда он установлен. Поэтому если DL = 0 и флаг переноса установлен (т.е. равен 1), то выполнится:

ADC DL,30

CF		BL
---		-----
1 <--	0110110 <--	
---	-----	

Рис.4.1. Инструкция "RCL BL,1" - (StIv) : наверно "RCL (?) [pause]

сложение DL (0) и 30h ("0") и 1h (перенос), дающее нам 31h ("1"). И теперь, имея в своем распоряжении инструкцию для перевода переноса в символ, мы можем приступить к печати.

Однако, вместо того, чтобы сразу запускать пример выполнения инструкции ADC, давайтеждемся законченной программы. Когда мы напишем эту программу, то выполним её инструкции по одной за шаг специальной процедурой, называемой пошаговым режимом, и с её помощью увидим, и как работает инструкция ADC, и как она прекрасно пригодится в нашей программе. Но сначала нам понадобится ещё одна инструкция, которую мы будем использовать для того, чтобы повторить RCL, ADC и INT 21h (печать) восемь раз: для каждого бита в байте.

Организация циклов

Как мы уже отмечали, возможности инструкции RCL не ограничиваются циклическим сдвигом байт, она может также двигать целые слова. Мы используем эту

возможность для демонстрации инструкции LOOP. Действие этой инструкции немного сходно с циклом FOR-NEXT в Бейсике, хотя имеются некоторые различия. Как и для цикла FOR-NEXT в Бейсике, мы должны сообщить LOOP о том, сколько раз запускать цикл. Мы делаем это, помещая счётчик повторений в регистр CX. В каждом цикле микропроцессор 8088 вычитает единицу из CX, и когда CX становится равным нулю, LOOP заканчивает цикл.

Почему именно регистр CX? Буква "C" в названии регистра CX означает "Счётчик" (англ. "Count"). Мы можем использовать этот регистр и в качестве регистра общего назначения, но, как вы увидите в следующей главе, регистр CX используется при необходимости повторения операций.

Ниже приводится простая программа, циклически сдвигающая содержимое регистра BX влево восемь раз, сдвигая BL в BH (но не наоборот, так как мы производим сдвиг через флаг переноса):

```
396F:0100  BBC5A3  MOV  BX,A3C5
396F:0103  B90800  MOV  CX,0008
396F:0106  D1D3    RCL  BX,1
396F:0108  E2FC    LOOP 0106
396F:010A  CD20    INT  20
```

Цикл начинается по адресу 106h (RCL BX,1) и заканчивается инструкцией LOOP. Число, записанное вслед за LOOP (106h), это адрес инструкции RCL. Когда мы запускаем программу, LOOP вычитает единицу из CX, и затем переходит к адресу 106h, если CX не содержит ноль. Инструкция "RCL BX,1" (циклический перенос влево на одну позицию) выполняется восемь раз, так как перед началом цикла в CX было загружено число 8.

```
0106:      <----  Decrement
              | CX
      LOOP 0106 --
              | Continue
              |  when CX=0
              v
      INT 20
```

Рис.4.2. Инструкция LOOP

Вы могли заметить, что, в отличие от цикла FOR-NEXT из Бейсика, инструкция LOOP пишется в конце нашего цикла (там где мы помещаем оператор NEXT в

Бейсике). А в начале цикла по адресу 106h, там, где записана инструкция RCL, отсутствует специальная инструкция, как например FOR в Бейсике. Если вы знаете такой язык программирования, как Паскаль, то вы видите, что инструкция LOOP в некоторой степени соответствует паре инструкций REPEAT-UNTIL, где инструкция REPEAT обозначает начало блока инструкций цикла.

Вы можете выполнить эту программу несколькими способами. Если вы просто наберёте "G", то не увидите каких либо изменений в распечатке регистров, так как Debug сохраняет значения всех регистров перед выполнением команды "G". И затем, если встретит инструкцию INT 20 (как это будет в программе), то он восстановит все регистры. Попробуйте ввести "G". Вы увидите, что IP был установлен в 100h (откуда вы начали), и что остальные регистры тоже выглядят не особо изменившимися.

Вы можете протрассировать эту программу, наблюдая за тем, как изменяется на каждом шаге содержимое регистров микропроцессора:

```
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0100 NV UP DI PL NZ NA PO NC
0CDE:0100 BBC5A3 MOV BX,A3C5
-T
AX=0000 BX=A3C5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0103 NV UP DI PL NZ NA PO NC
0CDE:0103 B90800 MOV CX,0008
-T
AX=0000 BX=A3C5 CX=0008 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0106 NV UP DI PL NZ NA PO NC
0CDE:0106 DID3 RCL BX,1
-T
AX=0000 BX=478A CX=0008 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
```

```
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0108 OV UP DI PL NZ NA PO NC
0CDE:0108 E2FC LOOP 0106
-T
AX=0000 BX=478A CX=0007 CX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0106 OV UP DI PL NZ NA PO NC
0CDE:0106 D1D3 RCL BX,1
-T
.
.
.
AX=0000 BX=C551 CX=0001 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0108 NV UP DI PL NZ
NA PO NC
0CDE:0108 E2FC LOOP0106
-T
AX=0000 BX=C551 CX=0008 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=010A NV UP DI PL NZ NA PO NC
0CDE:010A CD20 INT 20
-T
```

Существует ещё один способ запуска программы: вы можете набрать "G 10A", чтобы выполнить всю программу, кроме инструкции по адресу 10Ah; тогда регистры покажут результат программы.

Если вы поступите таким образом, то увидите, что CX=0, и что BX=C551 или BX=C5D1, в зависимости от значения флага переноса перед запуском программы. Число C5, которое инструкция MOV поместила в BL вначале, теперь находится в регистре BH, но BL не содержит A3, так как мы циклически сдвигали BX через флаг переноса. Позже мы рассмотрим другие способы циклического сдвига без использования переноса. Давайте вернёмся к нашей цели - печати числа в двоичном представлении.

Вывод на экран двоичного числа

Мы увидели, как можно разделить двоичное число на цифры и как перевести их в ASCII-символы. Если мы добавим инструкцию INT 21h для печати цифр,

то наша программа будет закончена. Ниже приводится программа, в которой первая инструкция устанавливает AH в 02 для вызова функции INT 21h (02 сообщает DOS о необходимости печатать символ из регистра DL):

```
3985:0100 B402 MOV AH,02
3985:0102 B90800 MOV CX,0008
3985:0105 B200 MOV DL,00
3985:0107 D0D3 RCL BL,1
3985:0109 80D230 ADC DL,30
3985:010C CD21 INT 21
3985:010E E2F5 LOOP 0105
3985:0110 CD20 INT 20
```

Мы уже видели, как работают части этой программы по отдельности, а теперь сложим их вместе. Используйте циклический сдвиг BL (с помощью команды "RCL BL,1") для того, чтобы произвести выборку битов числа, выберите число, которое необходимо напечатать в двоичном виде, загрузите его в регистр BL, а затем запустите программу командой "G". После выполнения инструкции INT 20h команда "G" восстановит в регистрах те значения, которые в них были прежде, поэтому BL будет содержать число, напечатанное в двоичном виде.

Инструкция "ADC DL,30" преобразует значение флага переноса в символ "0" или "1". Сначала инструкция "MOV DL,0" устанавливает DL в ноль, затем инструкция ADC добавляет 30h к DL, и, наконец, добавляется перенос. Так как 30h это ASCII код нуля, то результатом "ADC DL,30" будет код символа "0" при сброшенном флаге переноса (состояние NC) или код "1" при установленном флаге (CY).

Протрассируйте программу для того, чтобы увидеть результат её действия. Помните, что вам нужно быть осторожными при пошаговом режиме работы с командой "T". Программа содержит инструкцию "INT 21h", и, как вы уже видели раньше, при первой встрече с "INT 21h", DOS выполняет большой объём работы для этой инструкции. Вот почему вам не стоит использовать "T" для "INT 21".

Вы можете, однако, протрассировать все остальные инструкции, за исключением последней "INT 20", которая вас не будет касаться до самого конца программы. Во время трассировки, каждый раз, когда вы пройдёте цикл и достигнете инструкции "INT 21h",

печатайте "G 10E". Эта команда сообщит Debug о том, что надо продолжать выполнение программы до того момента, когда IP достигнет значения введённого адреса. Таким образом, Debug будет выполнять инструкцию "INT 21h" без трассировки и останавливаться при достижении инструкции LOOP по адресу 10E, что позволит вернуться к трассировке программы. (Число, которое вы напечатали после "G", в руководстве по DOS называется точка останова, эти точки очень полезны при анализе работы программ.)

Наконец, прервите программу при достижении инструкции "INT 20h", введя команду "G".

Команда обхода

Пытались вы или нет оттрассировать программу вышеуказанным методом, вы должны понять, что команда "G 10E" позволяет провести трассировку, минуя инструкцию INT, которая начинается, скажем, с 10Ch. Но это означает, что каждый раз, когда мы хотим провести трассировку, минуя инструкцию INT, необходимо знать адрес инструкции, следующей непосредственно за INT.

Оказывается, имеется специальная команда Debug, значительно облегчающая трассировку через инструкции INT. Команда "P" (от англ. "Proceed" -переходить, продолжать) делает всю работу за нас. Чтобы увидеть, как это происходит, протрассируйте программу, но на этот раз, дойдя до "INT 21h", наберите "P", а не "G 10E", как описывалось ранее.

В дальнейшем мы будем часто использовать команду "P", так как это очень удобный способ обхода команд, вызывающих из DOS большие подпрограммы, например INT. Перед тем, как идти дальше, мы должны отметить одно свойство команды "P" - она не документирована в руководствах по DOS до версии 3.00. Этот недостаток документации объясняется, видимо, тем, что фирма Microsoft не успела достаточно её протестировать перед выпуском версии 2.00 DOS. По этой причине, если у вас версия DOS до 3.00, вы должны знать, что команда "P" может не всегда срабатывать, однако у нас не было проблем с её использованием.

Вот, кажется, и всё, что нужно было сделать для того, чтобы вывести на экран двоичное число в виде строки единиц и нулей. Попробуйте для практики

выполнить простое упражнение: модифицируйте программу таким образом, чтобы она печатала "b" в конце двоичного числа (Подсказка: ASCII-код для "b" 62h).

Итог

В этой главе мы немного перевели дыхание после тяжелой работы над новыми понятиями в главах 1.3. Итак, где мы были и с чем познакомились?

Мы впервые встретились с флагами, и особое внимание обратили на флаг переноса, потому что он значительно упростил вывод двоичного числа на экран. Мы также узнали об инструкции циклического сдвига RCL, сдвигающей байт или слово влево на один бит за шаг.

После того, как мы узнали о флаге переноса и циклическом сдвиге байт и слов, мы рассмотрели новую версию инструкции сложения ADC, и стали почти подготовленными для написания программы печати двоичного числа.

Затем на сцену вышла инструкция LOOP. Загружая в регистр CX счётчик цикла, мы можем заставить микропроцессор 8088 выполнить цикл инструкций несколько раз. Мы установили CX в 8 для выполнения цикла восемь раз подряд.

Этого оказалось достаточным для написания программы. В дальнейшем мы также будем применять эти инструкции, а в следующей главе напечатаем двоичное число в шестнадцатеричном представлении, примерно так же, как это делает Debug. К тому времени, когда мы закончим главу 5, у нас будет хорошее понимание того, каким образом Debug переводит числа из двоичной формы в шестнадцатеричную. После этого мы начнём разбираться с другой способностью Debug - чтением чисел, написанных в шестнадцатеричной форме, и переводом их в двоичное представление микропроцессора 8088.

Глава 5. Вывод на экран чисел в шестнадцатеричной форме

Наша программа в главе 4 была достаточно простой для понимания. Мы были счастливы от того,

что наличие флага переноса значительно упростило вывод на экран двоичного числа в виде строки символов "0" и "1". Теперь мы займемся печатью чисел в шестнадцатеричном представлении. Работа усложнится для понимания: придётся записывать одни и те же последовательности инструкций несколько раз, но в главе 7 мы узнаем о процедурах, или подпрограммах, освобождающих нас от необходимости повторения. Предварительно мы познакомимся с некоторыми полезными инструкциями и посмотрим, как печатать числа в шестнадцатеричном представлении.

Операция сравнения и биты состояния

В последней главе мы узнали кое-что о флагах состояния и разобрались с флагом переноса, который в распечатке состояний регистров представлен значениями CY или NC. Остальные флаги, которые также не менее полезны, сохраняют состояние, или статус последней арифметической операции. Существует всего восемь флагов, и мы будем изучать их по мере необходимости.

Напоминаем, что CY означает, что флаг переноса равен 1, или, другими словами, установлен, в то время как NC означает, что флаг переноса равен 0 или сброшен. Для всех флагов 1 означает истинно, а 0 означает ложно. Например, если вы выполнили операцию вычитания с результатом 0, то флаг, называемый флагом нуля (англ. "Zero Flag"), будет установлен в 1 -истинно -, и вы его увидите на распечатке регистров как ZR (от англ. "Zero" - ноль). В противном случае, флаг нуля будет установлен в 0 - NZ (от англ. "Not Zero" - не ноль).

Давайте рассмотрим пример, в котором проверяется флаг нуля. Для этого инструкцией SUB произведём вычитание одного числа из другого. Если числа равны, то результат равен нулю, и флаг нуля появится на дисплее в виде ZR. Введите следующую инструкцию вычитания:

```
396F:0100 29D8 SUB AX,BX
```

Протрассируйте её с несколькими разными числами, наблюдая за состоянием флага нуля (ZR или NZ). Если вы поместите одинаковые числа (F5h в приведённом ниже примере) в регистры AX и BX, то

увидите, что флаг нуля будет установлен после одной операции вычитания и сброшен после другой:

```
-R
AX=00F5 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0100 NV UP DI PL NZ NA PO NC
0CDE:0100 29D8 SUB AX,BX
-T
AX=0000 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0102 NV UP DI PL ZR NA PO NC
0CDE:0102 3F AAS
-R IP
IP 0102
:100
-R
AX=0000 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0100 NV UP DI PL ZR NA PO NC
0CDE:0100 2908 SUB AX,BX
-T
AX=FF0B BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0102 OV UP DI PL NZ NA PO NC
0CDE:0102 3F AAS
```

Если мы вычитаем единицу из нуля, то результат будет FFFFh, который, как мы помним из главы 1, является дополнительным кодом. Можем ли мы узнать из распечатки регистров, является полученное число положительным или отрицательным? Да, так как существует другой флаг, называющийся флаг знака (англ. "Sign Flag"), изменяющийся от NG (сокр. от англ. "Negative") до PL ("Plus") и устанавливающимся в 1, если число является отрицательным с дополнительным кодом.

Ещё один флаг, который нас интересует, это флаг переполнения, который изменяется между OV ("Overflow" переполнение), и становится равным 1, и NV ("No Overflow"), устанавливается в 0. Флаг переполнения устанавливается в том случае, если знаковый бит изменился в той ситуации, когда этого не должно

было произойти. Например, если мы сложим два положительных числа, таких как 7000h и 6000h, то получим отрицательное число, D000h или -12288. Это ошибка, так как результат переполняет слово. Результат должен быть положительным, но таковым не является, поэтому микропроцессор 8088 устанавливает флаг переполнения. (Помните, что если мы имеем дело с числами без знака, то это не будет ошибкой, в этом случае мы должны игнорировать флаг переполнения.)

Попробуйте, проводя операции с разными числами, устанавливать и сбрасывать каждый из этих флагов; сделайте это несколько раз, пока не привыкните к обращению с ними. Для получения переполнения вычитите большое отрицательное число из большого положительного числа - например, 7000h-8000h, так как 8000h-это отрицательное число равное -32768 в виде дополнительного кода.

Теперь мы готовы рассмотреть набор инструкций, называемый инструкциями условного перехода. Они позволяют проверять флаги состояния более удобным образом, чем тот, который мы могли использовать до этого. Инструкция JZ (от англ. "Jump if Zero" - перейти, если ноль) переходит на новый адрес, если результат последней арифметической операции был ноль. Таким образом, если мы вслед за инструкцией SUB напомним, скажем, "JZ 15A", то нулевой результат вычитания будет означать, что микропроцессор 8088 начнёт выполнять инструкции, находящиеся по адресу 15A, а не следующую по порядку инструкцию.

Инструкция JZ проверяет флаг нуля, и если он установлен (ZR), то выполняется переход, аналогичный переходу в операторе Бейсика

```
IF A = 0 THEN 100.
```

Противоположной по отношению к JZ является инструкция JNZ (англ. "Jump if Not Zero" - перейти, если не ноль). Давайте рассмотрим простой пример с применением JNZ. В этом примере из числа вычитается единица до тех пор, пока в результате не получится ноль:

```
396F:0100 2C01 SUB AL,01
396F:0102 75FC JNZ 0100
396F:0104 CD20 INT 20
```

Поместите небольшое число в AL, чтобы побыстрее закончить цикл, затем протрассируйте программу, по одной инструкции за шаг, чтобы уви-

деть, как работает условное ветвление. Мы поместили "INT 20h" в конце программы, так что напечатайте "G", чтобы случайно не выйти из программы: это неплохая защитная практика.

Вы наверно обратили внимание на то, что применение SUB для сравнения двух чисел, имеет нежелательный посторонний эффект изменения первого числа. Другая инструкция, CMP (от англ. "Compare" -сравнение), позволяет производить сравнение без записи результата и без изменения первого числа. Результат используется только для установки флагов, поэтому после сравнения можно записать одну из многочисленных инструкций условного перехода. Для того чтобы увидеть, что именно происходит при сравнении, загрузите в регистры AX и BX одинаковые числа, например F5h, и протрассируйте инструкцию:

```
-A 100
0CDE:0100 CMP AX,BX
0CDE:0102
-T
AX=00F5 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0102 NV UP DI PL ZR NA PO NC
0CDE:0102 3F AAS
```

Теперь установлен флаг нуля (ZR), но оба регистра сохранили свое значение - F5h.

Давайте попробуем использовать CMP для печати одной шестнадцатеричной цифры. Мы создадим набор инструкций, в котором будут применены флаги, изменяющие выполнение программы, аналогично тому, как это выполняет оператор Бейсика IF-THEN. Флаги будут применяться для проверки таких условий, как "меньше чем", "больше чем" и так далее. Нам нет необходимости контролировать процесс установки или сброса флагов, при выполнении инструкции это происходит автоматически.

Вывод на экран одной шестнадцатеричной цифры

начнём с записи небольшого числа (между 0 и Fh) в регистр BL. Так как любое число между 0 и Fh соответствует одной шестнадцатеричной цифре, можно

перевести выбранное число в ASCII-символ и затем напечатать его. Рассмотрим шаги, которые надо предпринять для того, чтобы произвести этот перевод.

ASCII-символы от 0 до 9 имеют значения от 30h до 39h символы от A до F, однако, имеют значения от 41h до 46h. В этом и заключена проблема: ASCII символы разделены на две группы по семь символов в каждой. В результате переход в ASCII будет затруднён из-за наличия двух групп чисел (от 0 до 9 и от Ah до Fh), так что мы должны обрабатывать отдельно каждую группу. Программа на Бейсике выглядит так:

```
100 IF BL<&H0A                                [pause] (?)
    THEN BL=BL+&H30
    ELSE BL=BL+&H37
```

(Заметим, что мы написали число A в виде 0Ah, а не AH, чтобы не спутать число Ah с регистром AH. Мы будем часто помещать ноль перед шестнадцатеричными числами в ситуациях, как эта, чтобы не ошибиться. Так как наличие нуля перед шестнадцатеричным числом не изменяет его значения, то есть хорошая идея - помещать ноль перед всеми шестнадцатеричными числами.)

Программа перевода, написанная на Бейсике, довольно проста. К сожалению, машинный язык микропроцессора 8088 не включает оператора ELSE, он гораздо примитивнее Бейсика, и поэтому приходится немного изощряться. Следующая программа на Бейсике, показывает метод, который будет использоваться в программе, написанной на машинном языке:

```
100 BL=BL+&H30
110 IF BL>=&H3A
    THEN BL=BL+&H7
```

Вы можете убедиться, что программа работает, опробовав её на нескольких случайных примерах. Числа 0, 9, Ah и Fh особенно хороши, так как

Character	ASCII Code (Hex)
-----------	------------------

/	2F
0	30
1	31
2	32
3	33

Вывод на экран одной шестнадцатеричной цифры

4	34
5	35
6	37
7	37
8	38
9	39
:	3A
;	3B
<	3C
=	3D
>	3E
?	3F
@	40
A	41
B	42
C	43
D	44
E	45
F	46
G	47

Рис.5.1. Таблица ASCII кодов символов, используемых в шестнадцатеричных числах

эти четыре числа включают все граничные условия.

0 и Fh соответственно наименьшее и наибольшее шестнадцатеричные числа, состоящие из одной цифры, и поэтому используя их, мы проверяем начало и конец промежутка. Числа 9 и 0Ah, несмотря на то, что они следуют друг за другом, требуют наличия двух разных схем перевода в нашей программе. Используя 9 и 0Ah, мы убеждаемся, что выбрали правильную точку переключения между двумя схемами.

Версия программы, написанная на машинном языке, содержит немного больше шагов, но по существу она аналогична версии, написанной на Бейсике. Она использует инструкцию CMP и инструкцию условного перехода, называемуюся JL (сокр. от англ. "Jump if Less Than" - перейти, если меньше). Вот текст программы, которая берёт одну шестнадцатеричную цифру из регистра BL и печатает её в шестнадцатеричной форме:

```
3985:0100 B402 MOV AH,02
3985:0102 8BDA MOV DL,BL
```



```
3985:0104      80C230    ADD DL, 30
3985:0107      80FA3A    CMP DL, 3A
3985:010A      7C03      JL  010F
3985:010C      80C207    ADD DL, 70
3985:010F      CD21      INT 21
3985:0111      CD20      INT 20
```

Инструкция CMP, как мы видели ранее, вычитает два числа (DL-3Ah), чтобы установить флаги, но она не изменяет регистр DL. Поэтому, если DL меньше чем 3Ah, инструкция "JL 10F" осуществляет переход к инструкции "INT 21 h" по адресу 10Fh. Поместите шестнадцатеричное число, состоящее из одной цифры, в BL и протрассируйте пример, чтобы получить лучшее представление об инструкции CMP и алгоритме перевода шестнадцатеричной цифры в код ASCII. Не забывайте использовать или команду "G" с указанием точки останова, или команду "P", когда запускаете инструкции INT.

```
0107      CMP    DL, 3A
010A      JL     010F  --  Jump if
010C      ADD    DL, 70  |  DL<3Ah
010F      INT    21     <-
```

Рис.5.2. Инструкция JL

Ещё одна инструкция сдвига

Наша программа работает с любым шестнадцатеричным числом, состоящим из одной цифры, но если мы хотим напечатать двузначное шестнадцатеричное число, нам понадобится немного увеличить число шагов. Необходимо выделить каждую цифру (четыре бита, которые часто называются полубайт, или тетрада) двузначного шестнадцатеричного числа. В этом разделе мы увидим, что можно достаточно просто выделить первые, или старшие, четыре бита, соответствующие первой цифре, а в следующем разделе мы познакомимся с понятием логической операции, применяя которую отделим младшие четыре бита - вторую цифру в шестнадцатеричном числе.

Вспомним, что инструкция RCL циклически сдвигает байт или слово влево через флаг переноса. В последней главе мы использовали инструкцию "RCL BL,1", в которой единица является сообщением

микропроцессору 8088 о том, что надо сдвинуть BL на один бит. Мы можем осуществлять циклический сдвиг более чем на один бит, но мы не можем написать инструкцию "RCL BL,2". (Примечание: хотя инструкция "RCL BL,2" невозможна для микропроцессора 8088, она прекрасно работает с микропроцессором 80286 на IBM AT. Но так как более ранняя модель IBM PC используется шире, чем AT, мы будем писать программы именно для микропроцессора 8088.) Для циклического сдвига более чем на один бит, необходимо поместить счётчик сдвигов в регистр CL.

Регистр CL используется здесь также, как регистр CX применялся инструкцией LOOP при определении числа повторений цикла. Так как не имеет смысла осуществлять циклический сдвиг более чем 16 раз, то для записи числа сдвигов вполне подойдет восьмибитовый регистр CL.

Каким образом это связано с печатью двузначного шестнадцатеричного числа? План состоит в сдвиге байта в DL на четыре бита вправо. Чтобы выполнить это, потребуется немного другая инструкция сдвига, называемая SHR (сокр. от англ. "Shift Right" -логический сдвиг вправо). Используя SHR, мы сможем сдвинуть старшие четыре бита шестнадцатеричного числа в крайнюю правую тетраду (четыре бита).

		DL			CF
		-----			---
0 ->	0110110	->	1		
	-----		---		

Рис.5.3. Инструкция "SHR DL,1"

Необходимо также обнулить старшие четыре бита DL для того, чтобы этот регистр стал равным байту, сдвигаемому в правую тетраду. Если мы введём "SHR DL,1", то инструкция сдвинет байт в DL на один бит вправо, и в то же время, она сдвинет бит 0 во флаг переноса, записав в бит 7 ноль (старший, или крайний левый бит в DL). Если повторить эту операцию еще три раза, то в результате получится именно то, что требуется: четыре старшие бита сдвинутся на место четырёх младших битов, а на их месте окажутся нули. Весь этот логический сдвиг можно выполнить одной инструкцией, используя регистр CL как счётчик сдвига. Записав в CL "4" перед выполнением инст-

рукции "SHR DL,CL", мы можем быть уверены, что DL станет равным старшей шестнадцатеричной цифре.

Посмотрим, как всё это срабатывает. Поместите 4 в CL и 5Dh в DL, затем введите и протрассируйте следующую инструкцию сдвига:

```
3985:0100 D2EA SHR DL,CL
```

DL должен теперь быть равным 05h, которая является первой цифрой в числе 5Dh, и теперь её можно вывести на экран с помощью программы, которую мы использовали раньше. Следовательно, складывая вместе отдельные части программ, созданных к данному моменту, можно построить программу, извлекающую число из регистра BL и печатающую первую его шестнадцатеричную цифру:

```
3985:0100 B402 MOV AH,02
3985:0102 88DA MOV DL,BL
3985:0104 B104 MOV CL,04
3985:0106 D2EA SHR DL,CL
3985:0108 80C230 ADD DL,30
3985:010B 80FA3A CMP DL,3A
3985:010E 7C03 JL 010F
3985:0110 80C207 ADD DL,70
3985:0113 CD21 INT 21
3985:0115 CD20 INT 20
```

Логика И (AND)

Теперь, опробовав процедуру печати первой из двух цифр шестнадцатеричного числа, давайте посмотрим, как мы сможем выделить и напечатать вторую цифру этого числа. Здесь мы узнаем, как обнулять старшие четыре бита нашего начального (еще не сдвигавшегося) числа, оставляя DL равным младшим четырём битам. Это просто: старшие четыре бита обнуляются с помощью инструкции, называемой AND (логическое "И"). Эта инструкция относится к логическим инструкциям, которые берут свое начало в формальной логике.

В формальной логике мы можем сказать: "А истинно, если и В, и С оба истинны". Но если или В, или С ложно, то А должно также быть ложно. Если мы возьмём это выражение, подставив единицу вместо

"истинно" и ноль вместо ложно , то для различных комбинаций A, B и C мы можем построить таблицу истинности. Таблица истинности для операции AND, произведённой над двумя битами, приведена ниже:

AND		F	T		AND		0	1
-----	+	-----			-----	+	-----	
F		F	F	=	0		0	0
T		F	T		1		0	1

Слева и сверху приведены значения двух бит. Результаты операции, проведённой над ними, приведены в таблице, так что вы видите, например, что "0 AND 1" даёт 0.

Инструкция AND работает с байтами и словами, выполняя операцию AND над одинаково расположенными битами каждого байта или слова. Например, выражение "AND BL,CL" последовательно выполняет операцию AND сначала над битами 0 регистров BL и CL, затем над битами 1, битами 2, и так далее, и помещает результат в BL. Давайте поясним это с помощью примера в двоичном виде:

```
      1011 0101
AND 0111 0110
-----
      0011 0100
```

Кроме того, выполняя операцию AND над 0Fh и каким-либо числом, мы можем обнулить старшие биты этого числа:

```
      1011 0101
AND 0111 0110
-----
      0011 0100
```

Давайте используем эту логику в небольшой программе, которая берёт число из BL, выделяет младшую шестнадцатеричную цифру, выполняя операцию AND над 0Fh и старшими четырьмя битами, и затем печатает результат как ASCII-символ. Большую часть этой программы мы уже видели при выводе на экран старшей шестнадцатеричной цифры; единственной новой деталью будет инструкция AND:

```
3985:0100 B402 MOV AH,02
3985:0102 88DA MOV DL,BL
```

```

3985:0104      80E20F  AND DL,0F
3985:0107      80C230  ADD DL,30
3985:010A      80FA3A  CMP DL,3A
3985:010D      7C03    JL  0112
3985:010F      80C207  ADD DL,07
3985:0112      CD21    INT 21
3985:0114      CD20    INT 20

```

Перед тем, как мы начнём собирать части программы вместе, поработайте с ней, помещая в BL двузначные шестнадцатеричные числа. Вы должны увидеть правую шестнадцатеричную цифру введённого числа в регистре BL.

Сборка всех частей программы вместе

Нам не так много придётся менять в программе, чтобы собрать все части вместе. Необходимо только изменить адрес второй инструкции JL, которую мы используем для печати второй шестнадцатеричной цифры. Вот вся программа целиком:

```

3985:0100      B402    MOV AH,02
3985:0102      88DA    MOV DL,BL
3985:0104      B104    MOV CL,04
3985:0106      D2EA    SHR DL,CL
3985:0107      80C230  ADD DL,30
3985:010B      80FA3A  CMP DL,3A
3985:010E      7C03    JL  0113
3985:0110      80C207  ADD DL,07
3985:0113      CD21    INT 21
3985:0115      88DA    MOV DL,BL
3985:0117      80E20F  AND DL,0F
3985:011A      80C230  ADD DL,30
3985:011D      80FA3A  CMP DL,3A
3985:0120      7C03    JL  0125
3985:0122      80C207  ADD DL,07
3985:0125      CD21    INT 21
3985:0127      CD20    INT 20

```

После ввода программы необходимо напечатать "U 100" для того, чтобы увидеть разассемблированный листинг. Заметим, что мы повторили один набор из пяти инструкций: инструкции с 108h по 113h, и с 11Ah по 125h. В главе 7 мы увидим, как написать эту последовательность только один раз, используя инструкцию, похожую на оператор Бейсика GOSUB.

Итог

В этой главе мы узнали о том, как Debug переводит числа из двоичного формата микропроцессора 8088 в шестнадцатеричный формат, который мы можем читать. Что мы добавили к нашему растущему запасу знаний?

Прежде всего мы узнали о некоторых флагах, которые мы видим справа на распечатке регистров. Это биты состояния, отображающие информацию о последней арифметической операции. Например, флаг нуля показывает, был ли результат последней операции равен нулю. Мы также узнали, что можем сравнивать два числа с помощью инструкции CMP.

Затем мы узнали о том, как печатать шестнадцатеричное число, состоящее из одной цифры. И, вооружённые этой информацией, мы занялись изучением инструкции SHR, которая дала возможность сдвигать старшую цифру двухзначного шестнадцатеричного числа в четыре младшие бита BL. После этого мы смогли печатать цифру на экране.

Наконец, мы узнали, что инструкция AND позволяет нам отделить младшую шестнадцатеричную цифру от старшей. И, сложив все эти части вместе, мы написали программу, печатающую двузначное шестнадцатеричное число.

Можно было бы продолжить и заняться печатью четырёхзначного шестнадцатеричного числа, но мы иногда повторяем несколько раз одни и те же инструкции. Для того, чтобы избежать повторений при выводе на печать четырёхзначного числа, необходимо в главе 7 изучить процедуры.

Глава 6. Ввод символов с клавиатуры

Зная, как печатать байт в шестнадцатеричном представлении, мы займёмся прямо противоположным делом, а именно считыванием двух символов - шестнадцатеричных чисел с клавиатуры и переводом их в один байт.

Ввод одного символа

Вызов функций DOS "INT 21h", который мы используем, имеет функцию номер 1, считывающую символ, вводимый с клавиатуры. При изучении спо-

собои вызова функций в главе 4, мы видели, что номер функции должен быть помещён в регистр АН перед вызовом "INT 21h". Давайте попробуем поработать с функцией 1 для "INT 21h". Введите "INT 21h" по адресу 0100h:

```
396F:0100 CD21 INT 21
```

Затем поместите 01h в регистр АН и напечатайте либо "G 102", либо "P", чтобы запустить эту инструкцию. Ничего не произошло? - Вроде бы да: всё, что вы видите, это мерцающий курсор. Но на самом деле DOS ждёт, что вы нажмёте на клавишу (пока не делайте этого). Как только вы её нажмёте, DOS поместит ASCII-код символа нажатой клавиши в регистр AL. Мы применим эту инструкцию позже для считывания символов шестнадцатеричного числа, но сейчас посмотрим на то, что произойдёт при нажатии какой-либо клавиши.

Попробуйте нажать клавишу F1. DOS вернет 0 в AL, и вы также увидите точку с запятой (";"), появившуюся около дефиса - приглашения Debug.

Вот что произошло. F1 принадлежит к набору специальных клавиш с расширенными кодами, которые DOS обрабатывает отдельно от клавиш, представляющих обычные ASCII-символы. (Вы найдёте таблицу этих расширенных кодов в Приложении E, а также в конце руководства по Бейсику.) Для каждой из этих клавиш DOS посылает два символа, один за другим. Первый возвращаемый символ всегда ноль, означающий, что следующий символ представляет собой скэн-код (scan code) специальной клавиши.

Чтобы считать оба символа, надо выполнить "INT 21h" дважды. Но в нашем примере мы считываем только один символ и оставляем скэн-код в DOS. Когда Debug закончит выполнение команды "G 102" (или "P"), он начнёт считывать символы, и первым символом, который он считает, будет скэн-код, оставшийся от клавиши F1: а именно 59, который является ASCII кодом символа ";".

Позже, когда мы начнём писать программу Dskpatch, мы будем использовать эти расширенные коды для работы с курсором и функциональными клавишами. Однако до той поры мы будем работать только с обычными ASCII-символами.

Считывание шестнадцатеричного числа, состоящего из одной цифры

Мы применим метод перевода чисел, использованный в главе 5 для того, чтобы из одной шестнадцатеричной цифры получить ASCII-код символа в диапазоне от 0 до 9 и от A до F. При переводе одного символа, например такого, как C или D, из шестнадцатеричного вида в байт, из этого числа необходимо вычесть либо 30h (для промежутка от 0 до 9), либо 37h (для промежутка от A до F). Ниже приводится простая программа, считывающая один ASCII-символ и переводящая его в байт:

```
3985:0100      8401      MOV AH,01
3985:0102      CD21      INT 21
3985:0104      2C30      SUB AL,30
3985:0106      3C09      CMP AL,09
3985:0108      7E02      JLE 010C
3985:010A      2C07      SUB AL,07
3985:010C      CD20      INT 20
```

Большинство инструкций программы уже знакомы нам, но имеется и одна новая, JLE (сокр. от англ. "Jump if Less or Equal to" - перейти, если меньше или равно). В нашей программе эта инструкция осуществляет переход, если AL меньше или равен 9h.

Чтобы увидеть, как происходит переход от ASCII-символа к шестнадцатеричному числу, необходимо просмотреть содержимое регистра AL перед выполнением "INT 20h". Так как Debug восстанавливает регистры после выполнения этой инструкции, нужно также использовать точку останова (это уже делалось в главе 4). Напечатайте "G 10C", и вы увидите, что AL содержит шестнадцатеричное число, полученное из символа.

Попробуйте ввести несколько символов, таких, как "k" или строчная "d", которые не являются шестнадцатеричными цифрами, и вы увидите, что произойдет. Вы поняли, что эта программа работает корректно только для цифр от 0 до 9 и от A до F. Мы исправим этот недостаток в следующей главе, когда узнаем о подпрограммах или процедурах. До этого мы будем временно некорректными и будем игнорировать ошибочные условия: мы просто будем вводить

правильные символы, чтобы программа работала как надо.

Считывание двузначного шестнадцатеричного числа

Считывание двух шестнадцатеричных цифр не намного сложнее считывания одной, однако оно требует гораздо большего количества инструкций. Мы начнём со считывания первой цифры, затем поместим её шестнадцатеричное значение в регистр DL и умножим его на 16. Чтобы выполнить это умножение, мы сдвинем регистр DL на четыре бита влево, поместив шестнадцатеричный ноль (четыре нулевых бита) справа от цифры, которую мы только что считали. Инструкция "SHL DL,CL", используя регистр CL, в который загружено число четыре, выполняет это действие, вставляя нули справа. Инструкция SHL (от англ. "Shift Left" - логический сдвиг влево) ещё известна как арифметический сдвиг, так как она имеет такой же эффект, как и умножение на два, четыре, восемь и так далее, в зависимости от числа (соответственно единицы, двойки или тройки), хранящегося в CL.

CF		DL
---		-----
1	<-	0110110
---		-----
		<- 0

Рис.6.1. Инструкция "SHL DL,1"

Наконец, имея первую, сдвинутую влево, цифру, мы добавляем вторую шестнадцатеричную цифру к числу в DL (которое = первая цифра * 16). Вы можете разобраться со всеми этими деталями, поработав самостоятельно с программой:

```

3985:0100    B401    MOV    AH,01
3985:0102    CD21    INT    21
3985:0104    88C2    MOV    DL,AL
3985:0106    80EA30  SUB    DL,30
3985:0109    80FA09  CMP    DL,09
3985:010C    7E03    JLE    0111
3985:010E    80EA07  SUB    DL,07
3985:0111    B104    MOV    CL,04
    
```

3985:0113	D2E2	SHL DL,CL
3985:0115	CD21	INT 21
3985:0117	2C30	SUB AL,30
3985:0119	3C09	CMP AL,09
3985:011B	7E02	JLE 011F
3985:011D	2C07	SUB AL,07
3985:0121	CD20	INT 20

Теперь, после того, как у нас есть работающая программа, неплохо было бы проверить граничные условия и убедиться, что программа работает правильно. В качестве этих условий используйте числа 00, 09, 0A, 0F, 90, A0, F0 и другие числа, например 3C. Используйте точку, останова для запуска программы без выполнения инструкции "INT 20h". (Для ввода шестнадцатеричных чисел используйте только заглавные буквы.)

Итог

У нас наконец появился шанс проверить на практике то, чему мы научились в предыдущих главах, не особо перегружаясь новой информацией. Используя новую функцию (номер 1) "INT 21" для считывания символов, мы написали программу для считывания двузначного шестнадцатеричного числа. Кроме того, мы уделили особое внимание проверке граничных условий.

Теперь мы готовы закончить Часть I изучением процедур в микропроцессоре 8088.

Глава 7. Процедуры - двоюродные сестры подпрограмм

В следующей главе мы встретимся с MASM-макроассемблером, и начнём использовать язык ассемблера. Но перед тем, как мы расстанемся с Debug, мы рассмотрим последний набор примеров и узнаем о подпрограммах и особом месте хранения чисел, называемом стек.

Процедуры

Процедура - это список инструкций, который можно вызывать из различных мест нашей программы, а не повторять одни и те же инструкции каждый раз, когда они нужны. В Бейсике такие списки называются

подпрограммами, но мы зовем их процедурами по причинам, которые станут ясными позднее.

Мы будем входить и выходить из процедур так же, как мы это делаем в Бейсике. Мы вызываем процедуру с помощью инструкции CALL, которая сходна с оператором Бейсика GOSUB. А выходим мы из процедуры с помощью инструкции RET, которая работает как оператор Бейсика RETURN.

Вот простая программа на Бейсике, которую мы скоро перепишем на машинном языке. Эта программа вызывает подпрограмму десять раз, каждый раз печатая один символ, начиная от "A" и заканчивая "J":

```
10   A=&H41  ASCII для 'A'
20   FOR I= 1 TO 10
30   GOSUB 1000
40   NEXT I
50   END
1000 PRINT CHR$(A) ;
1100 A=A+1
1200 RETURN
```

Подпрограмма, следуя установившейся в Бейсике практике, начинается со строки 1000, оставляя место для добавления инструкций к основной программе, не затрагивая подпрограмму. Мы будем делать то же самое с процедурами, написанными на машинном языке, помещая их по адресу 200h, подальше от основной программы, начинающейся с адреса 100h. Мы также заменим "GOSUB 1000" на инструкцию "CALL 200h", которая вызывает процедуру из ячейки памяти 200h.

Вместо цикла FOR-NEXT в Бейсике, как было представлено ранее главе 4, мы напишем инструкцию LOOP. Остальные части основной программы должны быть вам знакомы.

```
3985:0100   B401   MOV    DL,41
3985:0102   B90A00  MOV    CX,000A
3985:0105   F8F800  CALL   0200
3985:0108   E2FB    LOOP   0105
3985:010A   CD20    INT     20
```

Первая инструкция помещает 41h (ASCII-код "A") в регистр DL, так как инструкция "INT 21h" печатает символ, заданный ASCII-кодом в DL. Сама инструкция

"INT 21h" размещена подальше, в процедуре по адресу 200h. Вот процедура, которую надо ввести по адресу 200h:

3985:0200	B402	MOV	AH,02
3985:0202	CD21	INT	21
3985:0204	FEC2	INC	DL
3985:0206	CE	RET	

В ней представлены две новых и две старых инструкции. Напоминаем, что 02h в регистре AH сообщает DOS о необходимости напечатать символ в DL при выполнении инструкции "INT 21h". "INC DL" первая из двух новых инструкций, увеличивает на единицу регистр DL, то есть, добавляет единицу к DL. Другая новая инструкция RET возвращает управление к инструкции (LOOP), следующей за CALL в основной программе.

Напечатайте "G", чтобы увидеть результат, а затем пошаговым методом выполните её, чтобы подробно рассмотреть, как она работает. Не забывайте о необходимости использовать либо точку останова, либо команду "P" для запуска инструкции "INT 21h".

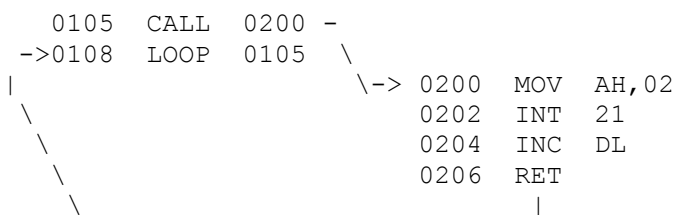


Рис.7.1. Инструкции CALL и RET

Стек и адрес возврата

Инструкция CALL нуждается в хранении в каком-либо месте памяти адрес возврата, необходимого для того, чтобы микропроцессор 8088 знал адрес инструкции, с которой он должен продолжить работу после того как выйдет из процедуры - встретит в ней инструкцию RET. Для этой цели подходит часть памяти, известная под названием стек. Регистр микропроцессора SP (от англ. "Stack Pointer" - указатель стека)

указывает на верх стека, а регистр SS (англ. "Stack Segment" - сегмент стека), содержит номер сегмента.

Работа стека микропроцессора 8088 аналогична в некоторой степени стопке подносов в столовой: когда наверх стопки ставится очередной поднос, нижний опускается. Последний (верхний) поднос в стопке является первым кандидатом на вынимание, поэтому еще одно название стека LIFO (сокр. от англ. "Last In, First Out" - "последним пришёл, первым ушёл"). Этот порядок LIFO именно то, что нам нужно для восстановления адреса возврата после того, как мы выполнили вложенный вызов, например подобный этому:

```
396F:0100    E8FD00    CALL 0200
.
.
396F:0200    E8FD00    CALL 0300
396F:0203    C3        RET
.
.
396F:0300    E8FD00    CALL 0400
396F:0303    C3        RET
.
.
396F:0400    C3        RET
```

Инструкция по адресу 100h вызывает другую инструкцию, расположенную по адресу 200h, которая, в свою очередь, вызывает инструкцию по адресу 300h, которая также вызывает инструкцию по адресу 400h, где, наконец, мы видим инструкцию возврата (RET). RET возвращается к инструкции, следующей за последним (CALL), по адресу 300h, так что микропроцессор продолжает выполнение инструкций с адреса 303h. Но там он встречает инструкцию RET, которая вытаскивает из стека следующий, еще более ранний адрес возврата (203h), так что микропроцессор продолжает выполнять инструкции с адреса 203h, и так далее. Попробуйте ввести программу, аналогичную предыдущей. Установите несколько вложенных вызовов, и затем её протрассируйте. Это поможет понять как происходит вызов процедуры и возврат в программу. Работа стека может показаться не очень интересной, но имеются различные способы его применения, поэтому хорошее понимание того, как он работает, будет вам полезно. (В следующей главе мы разыщем стек в памяти.).

	Address	Stack
	v	v
_____ \	0098:	_____
[SP100 >	0100:	0203
-----/	0102:	=====
	0104:	0103
		=====
		.
		=====
		.
		=====
		.
		=====

Рис.7.2. Стек перед выполнением инструкции "CALL 400"

Использование инструкций PUSH и POP

Стек - это оптимальное место временного хранения слов данных, при том условии, что мы будем аккуратными при восстановлении стека перед инструкцией RET. Мы убедились в том, что инструкция CALL помещает адрес возврата (слово) в начало стека, в то время как инструкция RET извлекает слово из начала стека и загружает его в регистр IP, (содержимое стека поднимается "вверх", и в начало стека попадает слово, располагаемое под извлечённым). Такую процедуру можно выполнить и с помощью инструкций PUSH и POP, соответственно записывающих и извлекающих из стека слова. Для чего это может пригодиться?

	Address	Stack
	v	v
_____ \	0098:	_____
[SP 98 >	0100:	0303
-----/	0102:	=====
	0104:	0203
		=====
		0103
		=====
		.
		=====
		.
		=====
		.
		=====

Рис.7.3. Стек после выполнения инструкции "CALL 400"

В ассемблере очень удобно сохранять значения регистров (они же являются переменными) в начале

процедуры и восстанавливать их в конце, перед самой инструкцией RET. В таком случае внутри процедуры мы сможем использовать эти регистры как нам угодно, пока не восстановим в конце их значения.

Программы состоят из процедур многих уровней, причем каждый уровень осуществляет вызов процедур нижнего за ним уровня. Сохраняя регистры в начале процедуры и восстанавливая их в конце, мы устраняем нежелательные влияния процедур одного уровня на другой, и это значительно облегчает программирование. Вы узнаете побольше о сохранении и восстановлении регистров в главе 13, когда мы будем говорить о модульном конструировании программ. А сейчас приведём пример (вводить его не надо), использующий сохранение и восстановление регистров CX и DX:

```
396F:0200    51      PUSH    CX
396F:0201    52      PUSH    DX
396F:0202    B90800  MOV     DL, 08
396F:0205    E8F800  CALL    0300
396F:0208    FEC2    INC     DL
396F:020C    5A      POP     DX
396F:020D    59      POP     CX
396F:020E    C3      RET
```

Обратите внимание, что инструкции POP расположены по отношению к инструкциям PUSH в обратном порядке, так как POP удаляет слово, помещенное в стек последним, а старое значение DX находится в стеке "глубже" старого значения CX.

Сохранение и восстановление регистров CX и DX позволяет произвольно изменять их значения внутри процедуры (которая начинается с адреса 200h), в то же время значения регистров, используемых процедурами, вызывающими эту, сохранены. Следовательно мы можем использовать эти регистры для хранения локальных переменных - переменных, которые применяются внутри процедур, не влияя на значения переменных, используемых вызывающей программой.

Такие локальные переменные применяются для упрощения задачи программирования. Поскольку мы аккуратно восстанавливаем начальные значения регистров то не нужно беспокоиться о том, что происходит со значениями регистров когда они используются вызывающими их процедурами. Проблема станет яснее после следующего примера, который является проце-

___Более простой способ считывания шестнадцатеричных чисел___

дурой считывания шестнадцатеричного числа. В отличие от программы в главе 6 наша программа теперь позволит использовать только правильные цифры, такие, как "А", но не "К".

Более простой способ считывания шестнадцатеричных чисел

Мы хотим создать процедуру, которая считывает символы, введённые с клавиатуры, и определяет, может или нет преобразовать их в шестнадцатеричное число в диапазоне от 0 до Fh. Те символы, которые преобразовать невозможно, на экран не выводятся. Для этого применяется инструкция "INT 21h", номер 8, которая считывает символ, но не выводит его на экран. Таким образом, мы будем отображать (англ. "echo") на экране только те числа, которые могут быть преобразованы в шестнадцатеричные заданного диапазона.

Поместите 8h в регистр AH и запустите эту инструкцию, напечатав "А" сразу после того, как введёте "G 102":

```
3985:0100      CD21  INT21
```

ASCII-код "А" (41h) теперь находится в регистре AL, но "А" не появилась на экране.

Используя эту функцию, программа может считывать с клавиатуры символы, не отображая их на экране до тех пор, пока не считает подходящую шестнадцатеричную цифру (от 0 до 9 и от А до F) и затем выведет её на экран. Вот процедура, которая делает это и переводит шестнадцатеричный символ в шестнадцатеричное число:

```
3985:0200      52      PUSH    DX
3985:0201      B408      MOV     AH, 08
3985:0203      CD21      INT     21
3985:0205      3C30      CMP     AL, 30
3985:0207      72FA      JB      0203
3985:0209      3C46      CMP     AL, 46
3985:020B      77F6      JA      0203
3985:020D      3C39      CMP     AL, 39
3985:020F      770A      JA      021B
3985:0211      B402      MOV     AH, 02
3985:0213      88C2      MOV     DL, AL
3985:0215      90      NOP
3985:0216      90      NOP
3985:0217      2C30      SUB     AL, 30
```


3985:0219	5A	POP	DX
3985:021A	C3	RET	
3985:021B	3C41	CMP	AL, 41
3985:021D	72E4	JB	0203
3985:021F	B402	MOV	AH, 02
3985:0221	88C2	MOV	DL, AL
3985:0223	CD21	INT	21
3985:0225	2C37	SUB	AL, 37
3985:0227	5A	POP	DX
3985:0228	C3	RET	

Процедура считывает символ в AL (с помощью "INT 21h", по адресу 203h) и проверяет, соответствует ли он условиям, используя CMP и инструкции условного перехода. Если считанный символ не подходит, то инструкции условного перехода посылают 8088 обратно к адресу 203, где "INT 21h" считывает другой символ. (JA - это "перейти, если больше" (англ. "Jump if Above"), а JB - это "перейти, если меньше" (англ. "Jump if Below"); обе инструкции работают с числами без знака, в то время как инструкция JL, рассмотренная ранее, работает с числами со знаком.)

В строке 211h мы уже знаем о том, что у нас есть подходящее число между 0 и 9, поэтому вычитаем код для символа "0" и возвращаем результат в регистре AL, не забывая восстановить регистр DX, который сохранили в начале процедуры. Процесс для цифр от A до F в основном аналогичен. Обратите внимание, что в этой процедуре две инструкции RET, но их может быть и больше, а может быть только одна.

Вот программа проверки этой процедуры:

3985:0100	E8FD00	CALL	0200
3985:0103	CD20	INT	20

Как и ранее, используйте или команду "G" с точкой останова или команду "P". Необходимо выполнить инструкцию "CALL 200h" без выполнения "INT 20h", чтобы увидеть регистры перед тем, как программа прервется и регистры восстановятся.

Курсор появится в левой части экрана и будет ждать ввода символа. Напечатайте "к", который не является правильным символом. Ничего не произойдет. Теперь напечатайте один из заглавных шестнадцатеричных символов. Вы должны увидеть шестнадцатеричное значение символа в AL и сам символ,

появившийся на экране. Испытайте эту процедуру на граничные условия: "\" (символ, стоящий перед нулем), "0", "9", ":" (символ, стоящий после "9"), и так далее.

Теперь, когда у нас есть эта процедура, программа, считывающая двузначное шестнадцатеричное число и обрабатывающая ошибки, стала довольно простой:

3985:0100	E8FD00	CALL	0200
3985:0103	88C2	MOV	DL,AL
3985:0105	B104	MOV	CL,04
3985:0107	D2E2	SHL	DL,CL
3985:0109	E8F400	CALL	0200
3985:010C	00C2	ADD	DL,AL
3985:010E	B4D2	MOV	AH,02
3985:0110	CD21	INT	21
3985:0112	CD20	INT	20

Вы можете запускать эту программу из DOS, так как она считывает двузначное шестнадцатеричное число и затем высвечивает ASCII-символ, соответствующий числу, которое вы ввели.

Если не учитывать процедуру, то основная программа значительно упростилась по сравнению с версией, написанной в последней главе. Добавленная процедура обработки ошибок усложнила программу, но теперь можно быть уверенным в том, что она работает только с правильными числами.

Здесь мы также увидели причину, по которой надо сохранять значение регистра DX в процедуре. Основная программа хранит в DL шестнадцатеричное число, поэтому наша процедура не должна изменять его. С другой стороны, процедура должна сама использовать DL для отображения символов. Таким образом, используя инструкцию "PUSH DX" в начале нашей процедуры и "POP DX" в конце, мы освобождаем себя от таких проблем.

В дальнейшем для того чтобы избежать путаницы, мы будем всегда сохранять значения регистров, используемых в процедуре.

Итог

Наше программирование всё более совершенствуется. Мы узнали о процедурах, позволяющих при-

менять один и тот же набор инструкций, не переписывая его несколько раз. Мы также узнали о стеке, и увидели, что CALL сохраняет адрес возврата на вершине стека, в то время как инструкция RET возвращает к этому адресу управление.

Мы увидели, как использовать стек не только для хранения адресов возврата. Мы использовали стек для сохранения значений регистров (инструкцией PUSH), поэтому смогли применять их не только в основной программе, но и в процедуре. Восстанавливая регистры (с помощью инструкции POP) в конце каждой процедуры, мы избежали нежелательного взаимодействия между процедурами. Всегда сохраняя и восстанавливая регистры в процедурах, можно вызывать другие процедуры, не беспокоясь о том, какие регистры в них используются.

И наконец, вооруженные этими знаниями, мы занялись созданием улучшенной программы считывания шестнадцатеричных чисел на этот раз с обработкой ошибок. Программа, написанная в этой главе, похожа на ту, которую мы будем использовать в последующих главах, когда начнём писать создавать Dskpatch.

Теперь мы готовы к тому, чтобы перейти к Части II, в которой мы познакомимся с ассемблером. В следующей главе мы увидим, как использовать ассемблер для перевода программы в машинный язык. Мы также увидим, что нет необходимости оставлять место между процедурами, помещая процедуру по адресу 200h.

ЧАСТЬ II . Язык ассемблера

Глава 8. Добро пожаловать в ассемблер!

Итак, мы практически готовы к знакомству с ассемблером - программой, значительно упрощающей процесс программирования. Теперь мы сможем писать человекочитаемые программы, используя ассемблер для перевода созданных программ в машинный код.

К сожалению, эта и следующая главы будут осложнены описанием деталей ассемблера, но итоговый результат окупит затраченные усилия. После того как мы научимся применять ассемблер достаточно уверенно, мы вернемся к изучению того, как писать на нем программы. Итак, давайте начнём.

Создание программы без использования Debug

До этого момента мы сначала запускали Debug, а затем вводили инструкции программы. Теперь мы оставим Debug в покое и станем создавать программы без него, используя для их написания любой текстовый редактор. Программы будут иметь более удобочитаемый вид, упрощающий их восприятие. Мы начнём с создания исходного файла - так называется текстовая версия ассемблерной программы. Сейчас мы создадим исходный файл для программы Writestr, которую мы написали в главе 3. Чтобы освежить вашу память, приведём версию программы, созданной с помощью Debug:

396F:0100	B402	MOV	AH, 02
396F:0102	B261	MOV	DL, 2A
396F:0104	CD21	INT	21
396F:0106	CD20	INT	20

Используйте любой текстовый редактор для ввода приведённых ниже строк в файл под названием "WRITESTR.ASM" (расширение .ASM означает, что это

ассемблерный исходный файл). Здесь, как и в Debug, строчные буквы работают так же хорошо, как заглавные, но мы будем продолжать применять заглавные буквы для того, чтобы избежать путаницы между цифрой "1" (единица) и строчной буквой "l":

```
CODE_SEG    SEGMENT
    MOV     AH, 2h
    MOV     DL, 2Ah
    INT     21h
    INT     20h
CODE_SEG    ENDS
    END
```

Это та же самая программа, которую мы создали в главе 3, но она содержит некоторые необходимые изменения и дополнения. Не обращая пока внимания на три новые строчки в нашем исходном файле, заметим, что перед каждым шестнадцатеричным числом стоит "h". Эта буква говорит ассемблеру о том, что число, после которого она стоит, шестнадцатеричное. В отличие от Debug, которая воспринимает все числа шестнадцатеричными, ассемблер предполагает их десятичными. Буква h сообщает ассемблеру о том, что введенное число является шестнадцатеричным.

Примечание: Прежде чем двинуться дальше, хотим предупредить: ассемблер может запутаться с буквы и числами, подобными ACh, начинающимися с буквы и похожими на имя или инструкцию. Чтобы избежать этого, печатайте нуль перед шестнадцатеричным числом начинающимся с буквы. Например, печатайте 0ACh, а не ACh.

Посмотрите, что произойдет, если мы напишем в программе ACh, а не 0ACh. Вот программа:

```
CODE_SEG    SEGMENT
    MOV     DL, ACh
    INT     20h
CODE_SEG    ENDS
    END
```

А вот ответ ассемблера:

```
A:\>MASM TEST
Microsoft (R) Macro Assembler Version 4.00
Copyright (C) Microsoft Corp 1981,1983,1984,1985. All rights reserved.
TEST.ASM(2) : error 9: Symbol not defined AC
```

```
51070 Bytes Symbol space free
```

```
0 Warning Errors
1 Severe Errors
A:\>
```

Это определённо не похвала. Но, изменив ACh на 0ACh, мы удовлетворим ассемблер.

Скажем также о разделении команд в ассемблерных программах. Мы использовали символ "TAB" для того, чтобы сделать исходный текст лучше читаемым. Сравните программу, которую вы ввели, например, с такой версией:

```
CODE_SEG      SEGMENT
               MOV     AH,2h
               MOV     DL,2Ah
               INT     21h
               INT     20h
CODE_SEG      ENDS
               END
```

Выглядит достаточно неупорядоченно и, хотя самому ассемблеру её внешний вид безразличен, аккуратно введенный текст программы настраивает на серьёзное к ней отношение.

```
                This is a label
                Это метка
                |
                ---
MOV             DL,2Ah

                This is a number
                Это число
                |
                ----
MOV             DL,02Ah

                |
The 0 tells MASM that is a number
```

Рис. 8.1. Помещайте нули перед шестнадцатеричными числами, начинающимися с буквы, иначе ассемблер примет их за имена

Теперь мы вернёмся к трём новым строкам в исходном файле. Эти строки являются Псевдооператорами . Они называются так потому, что в отличие от инструкций, генерирующих код, они всего лишь со-

общают ассемблеру дополнительную информацию. Псевдооператор END означает окончание исходного текста, по его наличию ассемблер узнаёт о том, что работа окончена. Позднее мы узнаем, что END полезен также и для других целей. Сейчас мы отложим в сторону дальнейшие дискуссии об этом псевдооператоре и двух других и посмотрим, как использовать сам ассемблер.

Создание исходных файлов

После того, как вы ввели строки WRITESTR.ASM, надо упомянуть ещё одно соглашение перед тем, как мы перейдём к непосредственному ассемблированию нашей программы. Ассемблер может использовать такие исходные файлы, которые содержат только стандартные ASCII-символы. Если вы применяете для создания исходного файла текстовый процессор, то имейте в виду, что не все текстовые процессоры записывают файлы, используя только стандартные ASCII-символы. Word Star является первым таким "преступником", Microsoft Word - вторым. Для этих процессоров используйте специальный недокументный или неформатированный режим записи файла. Перед тем, как вы попытаетесь ассемблировать WRITESTR.ASM, убедитесь, что это именно ASCII-файл. Находясь в DOS, напечатайте:

```
A:\>TYPE WRITESTR.ASM
```

Вы должны увидеть тот же текст, который ввели в текстовом редакторе. Если вы увидите в вашей программе странные символы, то для ввода текста программ нужно использовать другой текстовый редактор. Кроме того, в файле необходимо оставлять пустую строку после оператора END.

Теперь давайте начнем ассемблировать Writestr (не забудьте напечатать точку с запятой, как показано ниже):

```
A:\>MASM WRITESTR;  
The IBM Personal Computer Assembler  
Version 1.00 (C) Copyright IBM Corp 1981
```

```
Warning  Errors  
Severe   Errors
```

```
0 0
A:\>
```

Работа ещё не закончена. На этом этапе ассемблер создал файл, называющийся WRITESTR.OBJ, который вы найдете на диске. Это промежуточный файл, называющийся объектным файлом. Он содержит программу на машинном языке, вместе с большим количеством "бухгалтерской" информации, используемой другой программой, называющейся Linker (редактор связей, компоновщик).

Компоновка

Теперь необходимо, чтобы редактор связей взял .OBJ-файл и сделал его .EXE-версию. Скопируйте LINK.EXE с вашего DOS диска на диск, содержащий исходный файл и ассемблер. Затем напечатайте следующее:

```
A:\>LINK WRITESTR;

IBM-Personal Computer linker
Version 1.10 (C) Copyright IBM Corp 1982

Warning: No STACK segment

There was 1 error detected

A:\>
```

Ошибка? В действительности нет. (В некоторых версиях MS-DOS редактор связей (linker) вообще не считает это сообщение за ошибку.) Несмотря на то, что компоновщик предупреждает нас о том, что отсутствует сегмент стека, он нам и не нужен. После того, как мы узнаем, каким образом можно использовать все особенности микропроцессора, мы увидим, почему нам может позже понадобится сегмент стека.

Наконец, у нас есть .EXE-файл, но и это ещё не всё. Нам надо сделать ещё один шаг - создать .COM-версию, которая является тем же самым, что мы создали с помощью Debug. Опять-таки, вы увидите позже, почему нам нужны все эти шаги. А сейчас давайте создадим .COM-версию Writestr.

Для последнего шага нам понадобится программа EXE2BIN.EXE с дополнительного диска DOS. Exe2bin, как и предполагает её название, превращает .EXE-файл

в .COM, или двоичный (от англ. "binary") файл. Существуют некоторые различия между .EXE и .COM-файлами, мы будем разбирать их значительно позже, а сейчас создадим .COM-файл. Напечатайте:

```
A:\>EXE2BIN WRITESTR WRITESTR.COM
A:\>
```

Лаконичный ответ. Чтобы увидеть, сработала ли программа Exe2bin, давайте посмотрим на файлы с именем Writestr, которые мы создали:

```
A:\>DIR WRITESTR.*
 Volume in drive A has label
Directory of A:\

WRITESTR.ASM  78   7-25-635:00p
WRITESTR.OBJ  46   7-25-637:02p
WRITESTR.EXE  640  7-25-637:04p
WRITESTR.COM   8   7-25-637:06p
      4 File(s)  23552 bytes free
A:\>
```

Это точное число файлов, включая WRITESTR.COM. Напечатайте "writestr", чтобы запустить .COM-версию и убедитесь, что ваша программа функционирует правильно (напоминаем, что она должна печатать звездочку на экране). Точная длина первых трёх файлов может немного варьироваться.

Результаты могут показаться немного разочаровывающими, так как кажется, что вернулись к результату в главе 3, но это не так: мы на самом деле сделали большое дело. Это станет ясно, когда мы снова начнём работать с вызовами. Обратите внимание, что мы ещё ни разу пока не побеспокоились о том, где наша программа будет помещена в память, как мы это делали с помощью IP в Debug. Все проблемы размещения в памяти были решены без нашего участия.

Очень скоро вы оцените эту способность ассемблера. Она делает программирование гораздо проще. Например, помните, в последней главе мы потратили память, поместив основную программу по адресу 100h, а процедуру, которую вызывали - по адресу 200h. Мы увидим, что ассемблер позволяет нам помещать процедуры непосредственно после основной программы без какого-либо пропуска. Посмотрим, как наша программа выглядит в Debug.

Обратно в Debug

Давайте введём созданный .COM-файл обратно в Debug и разассемблируем его, чтобы увидеть, как Debug реконструирует программу из машинного кода WRITESTR.COM:

```
A:\>DEBUG WRITESTR.COM
-U
397F:0100      B402      MOV      AH,02
397F:0102      B261      MOV      DL,2A
397F:0104      CD21      INT       21
397F:0106      CD20      INT       20
```

Получили именно то, что мы уже имели в главе 3. Это всё, что Debug видит в WRITESTR.COM. Оператор END и дополнительные инструкции о сегментах - CODE_SEG SEGMENT и CODE_SEG ENDS - отсутствуют. Что же с ними произошло?

Эти инструкции не появились и в последней, написанной на машинном языке, версии программы, так как они являются псевдооператорами, а такие операторы служат только для дополнительной информации, необходимой ассемблеру для корректного функционирования. Ассемблер заботится о большом количестве "бухгалтерии" всего для нескольких оттранслированных строк. Мы будем применять псевдооператоры для облегчения нашего труда, а далее, когда поближе познакомимся с сегментами в главе 11, узнаем, как они воздействуют на нашу программу.

Комментарии

Так как мы больше не работаем в программе Debug, то можем добавить в программу некоторые элементы, которые ассемблер видит, но не транслирует. Возможно, наиболее важными дополнениями, которые мы можем сделать, являются комментарии, необходимые для того, чтобы сделать программу более понятной. В ассемблерных программах мы помещаем комментарии после точки с запятой (";"), которая работает аналогично символу "***" в Бейсике. Ассемблер игнорирует всё, что находится в строке после точки с запятой, так что мы можем писать после него всё, что посчитаем необходимым. Если добавить некоторые комментарии в нашу программу:

CODE_SEG	SEGMENT	
MOV	AX, 2h	;Выбор функции DOS номер 2, выводящей ;СИМВОЛ
MOV	DL, 2Ah	;Загрузить ASCII-код символа '*' для ;вывода на печать
INT	21h	;Распечатать его через INT 21h
INT	20h	;Завершить программу и выйти в DOS
CODE_SEG	ENDS	
END		

то мы сможем понять смысл программы без особых затруднений, не пытаясь вспоминать, что означает каждая строка.

Метки

Перед тем, как закончить эту главу, мы рассмотрим ещё одну возможность ассемблера - метки, делающие программирование более плавным.

Раньше при необходимости перейти из одной части программы в другую с помощью одной из команд условного перехода нам нужно было знать адрес того места в программе, куда мы переходим. Однако, при добавлении новых инструкций в программу, приходилось изменять адреса в инструкциях перехода. Ассемблер решает эту проблему с помощью меток - имён, которые присваиваются адресам или ячейкам памяти. Метка занимает место адреса. Как только ассемблер видит метку, он заменяет её на правильный адрес, перед тем, как послать её в микропроцессор 8088.

```

      |-----> 0111
      |          /
      |      010C  JLE  DIGIT1
      |      010E  SUB  DL
DIGIT1: 0111  MOV  CL
        0113  SHL  DL, 1

```

Рис. 8.2. Ассемблер подсчитывает адреса меток

Метки могут быть до 31 символа в длину и содержать буквы, цифры, и любые из следующих символов: знак вопроса ("?"), точку ("."), символ "@", символ подчеркивания ("_") и знак доллара ("\$"). Метки могут начинаться с цифры (от 0 до 9), а точка может быть использована только в качестве первого символа.

В качестве примера рассмотрим программу из главы 6, считывающую введенное с клавиатуры дву-

значное шестнадцатеричное число. Она содержит два перехода, "JLE 0111" и "JLE 011F". Вот её старая версия:

```
3985:0100 B401 MOV AH,01
3985:0102 CD21 INT 21
3985:0104 88C0 MOV DL,AL
3985:0106 80EA30 SUB DL,30
3985:0109 80FA09 CMP DL,09
3985:010C 7E03 JLE 0111
3985:010E 80EA07 SUB DL,07
3985:0111 B104 MOV CL,04
3985:0113 D2E2 SHL DL,CL
3985:0115 CD21 INT 21
3985:0117 2C30 SUB AL,30
3985:0119 3C09 CMP AL,09
3985:011B 7E02 JLE 011F
3985:0110 2C07 SUB AL,07
3985:011F 00C2 ADD DL,AL
3985:0121 CD20 INT 20
```

Конечно, сразу не ясно, что эта программа делает, и если вы этого не помните, то попробуйте с ней немного поработать и снова разобраться в её особенностях. После добавления меток и комментариев, поясняющих назначение отдельных инструкций, программа будет выглядеть так:

```
CODE_SEG      SEGMENT
    ASSUME     CS:CODE_SEG      ; (Будет объяснено в главе 11)
    MOV        AH,1h           ;Выбрать функцию DOS номер 1, ввод
                                ;символа
    INT 21 h                   ;Считать значение символа, переслать
                                ;его в регистр AL
    MOV        DL,AL           ;Переслать ASCII, переслать в DL
    SUB        DL,30h          ;Вычесть 30h для, того чтобы
                                ;преобразовать в 0-9
    CMP        DL,9h           ;Число в диапазоне 0-9?
    JLE        DIGIT1         ;Да, у нас есть первая цифра (четыре
                                ;бита)
    SUB        DL,7h           ;Нет, вычесть 7h, для того чтобы
                                ;преобразовать в буквы A-F
DIGIT1:
    MOV        CL,4h           ;Подготовка к умножению на 16
    SHL        DL,CL           ;Умножение сдвигом
    INT 21h                   ;Взять следующий символ
    SUB        AL,30h          ;Повторить преобразование
```

```

    CMP     AL, 9h      ;Цифра от 0 до 9
    JLE     DIGIT2      ;Да, есть вторая цифра
    SUB     AL, 7h      ;Нет, вычесть 7h
DIGIT2:
    ADD     DL, AL      ;ADD (добавить вторую цифру)
    INT     20h        ;Выход
CODE_SEG   ENDS
END

```

Метки этой программы, DIGIT1 и DIGIT2, принадлежат к типу, называемому NEAR, так как после них стоит двоеточие (":"). Термин NEAR связан с сегментами, которые мы рассмотрим в главе 11 вместе с псевдооператорами SEGMENT, ENDS и ASSUME. Теперь, если вы проассемблируете предыдущую программу и затем её разассемблируете с помощью Debug, то увидите, что DIGIT1 заменено на 0111h, а DIGIT2 заменено на 011Fh.

Итог

Это была значительная глава. Как будто бы мы попали в новый мир (фактически так оно и было). Оказалось, что с ассемблером работать гораздо проще, чем с Debug, так что теперь мы можем начать писать настоящую программу, в которой ассемблер сделает за нас большую часть рутинной работы.

О чём мы здесь узнали? Мы начали с изучения того, как создать исходный файл и затем прошли через все этапы создания работающей программы: ассемблирование, компоновка и перевод файла из .OBJ в .EXE, а затем и в .COM-файл, используя в качестве примера простую программу из главы 3. Ассемблерная программа, которую мы создали, содержала несколько псевдооператоров, которых мы до этого не видели, но мы с ними близко познакомимся в дальнейшем, как только привыкнем к использованию ассемблера. Мы будем помещать по мере необходимости псевдооператоры SEGMENT, ENDS и END в создаваемые программы, хотя до главы 11 мы не будем знать точно, для чего мы это делаем.

Затем мы узнали о комментариях. Комментарии добавляют так много к читаемости программ, что в дальнейшем мы будем сопровождать ими все создаваемые программы.

В конце главы мы узнали о метках, которые делают программы ещё более читаемыми и понятными. Мы будем использовать все эти идеи и методы до самого конца книги. Сейчас мы перейдем к следующей главе и посмотрим, каким образом ассемблер упрощает применение процедур.

Глава 9. Ассемблер и процедуры

Теперь, после того как мы немного познакомились с самим ассемблером, давайте попытаемся побольше узнать о том, как пишутся ассемблерные программы. В этой главе мы вернёмся к процедурам. Вы увидите, что с помощью ассемблера мы можем значительно упростить написание процедур. Затем мы перейдём к написанию таких процедур, которые будут применяться в дальнейшем, при создании программы Dskpatch.

Мы начнём с двух процедур для печати байта в шестнадцатеричном виде. По дороге мы повстречаем ещё несколько псевдооператоров. Но, также как и псевдооператоры SEGMENT, END и ENDS из последней главы, они попрежнему останутся для нас "таинственными незнакомцами" до главы 11, где мы расширим знания о сегментах.

Процедуры ассемблера

Когда мы только начинали использовать процедуры, то оставляли некоторый промежуток памяти между расположением основной программы и её процедурами для того, чтобы предотвратить возможное перекрытие кодом основной программы кода процедуры. Теперь же, когда у нас есть ассемблер, и он сам делает всю работу по присваиванию адресов инструкциям, нам больше не нужно оставлять этот промежуток. С помощью ассемблера каждый раз после внесения необходимых изменений мы можем оттранслировать программу заново.

В главе 7 мы создали небольшую программу с одной инструкцией CALL. Программа печатала буквы от А до J и выглядела следующим образом:

3985:0100	B241	MOV	DI, 41
3985:0102	B90A00	MOV	CX, 000A

```

3985:0105 E8F800 CALL 0200
3985:0108 E2F8 LOOP 0105
3985:010A CD20 INT 20
3985:0200 B402 MOV AH,02
3985:0202 CD21 INT 21
3985:0204 FEC2 INC DL
3985:0206 C3 RET

```

Давайте переведем эту программу на ассемблер. Читать её без меток и комментариев будет трудно, поэтому мы добавим эти полезные "украшения".

Листинг 9.1. Программа PRINTAJ.ASM.

```

CODE_SEG      SEGMENT
    ASSUME     CS:CODE_SEG
    ORG        100h      ;Сделать этот файл в виде .COM (будет
                        ;объяснено дальше)

PRINT_A_J     PROC NEAR
    MOV        DL,'A'    ;Старт с символа A
    MOV        CX,10     ;Печатать 10 символов, начиная с A
PRINT_LOOP:
    CALL       WRITE_CHAR ;Печатать символ и перейти к
                        ;следующему
    LOOP       PRINT_LOOP ;Продолжить для 10 символов
    INT        20h       ;Возврат в DOS
PRINT_A_J     ENDP

WRITE_CHAR    PROC NEAR
    MOV        AH,2      ;Установить код функции для вывода
                        ;символа
    INT        21h       ;Печатаемый символ уже в DL
    INC        DL        ;Перейти к следующему символу в алфавите
    RET
WRITE_CHAR    ENDP

CODE_SEG      ENDS
END           PRINT_A_J

```

Здесь появились четыре новых псевдооператора: ASSUME, ORG, PROC и ENDP. Псевдооператор ASSUME имеет отношение к сегментам, а ORG к тому способу, которым DOS загружает программы (подробнее мы узнаем об этом в главе 11).

PROC и ENDP - псевдооператоры определения процедур. Как можно видеть из листинга, и основная программа, и процедура, расположенная по адресу 200h, заключены в соответствующие пары псевдо-

операторов PROC и ENDP, которые, в свою очередь, заключены между псевдооператорами SEGMENT и ENDS (от англ. "End Segment" - конец сегмента).

PROC определяет начало процедуры, а ENDP - её конец. Метка перед каждой из процедур - это имя, которое мы ей присвоили. Таким образом, в главной процедуре, PRINT_A_J, мы можем заменить инструкцию "CALL 200" на более удобное и понятное выражение "CALL WRITE_CHAR". Достаточно только вставить имя процедуры в текст программы, а ассемблер присвоит необходимые адреса сам.

Псевдооператоры NEAR и FAR (более подробно о FAR также будет сказано позднее) поставляют ассемблеру информацию о том, как мы используем сегменты. Ассемблер применяет эту информацию при трансляции инструкции CALL. Существуют два типа инструкций вызова CALL и RET: "близкий" и "дальний". Дальний CALL, который мы пока не будем использовать, вызывает процедуру, располагаемую в другом сегменте. Близкий CALL, наоборот, вызывает процедуру из того же сегмента, в котором находится сам.

В этой книге мы будем иметь дело с программами, которые помещаются в одном 64 Кбайтном сегменте, так что все процедуры будут NEAR-процедурами. NEAR информирует ассемблер о том, что процедура находится в том же сегменте, что и процедура, вызывающая её. Когда ассемблер видит "CALL WRITE_CHAR", то он по наличию NEAR в программе "WRITE_CHAR PROC NEAR" понимает, что WRITE_CHAR находится в том же сегменте, что и PRINT_A_J.

Ассемблер нуждается в информации о сегментах, так как имеются две версии инструкций CALL и RET - одна используется тогда, когда мы не изменяем сегменты, вторая - когда мы это делаем. Из программы ясно, что обе процедуры принадлежат к одному сегменту, так как мы поместили их внутри пары псевдооператоров, определяющей этот сегмент: SEGMENT и ENDS. Позже, когда мы разобьём программу на части и поместим их в разные исходные файлы, правильное использование NEAR и FAR станет очень важным.

И наконец, так как в программе имеются две процедуры, то необходимо сообщить ассемблеру, какую

из них использовать в качестве главной процедуры, иначе говоря, откуда микропроцессор 8088 должен начать выполнение программы. На эту деталь указывает псевдооператор END. Напечатав "END PRINT_A_J", мы сообщили ассемблеру, что PRINT_A_J - основная процедура. Позже мы увидим, что главная процедура может располагаться в любом месте исходного файла. Но так как сейчас мы имеем дело с .COM-файлами, то главную процедуру необходимо помещать в исходном файле первой.

Если вы еще этого не сделали, то введите программу в файл PRINTAJ.ASM, и получите .COM-версию, используя те же шаги, которые делали в предыдущей главе:

```
MASM      PRINTAJ;  
LINK      PRINTAJ;  
EXE2BIN   PRINTAJ      PRINTAJ.COM
```

Затем попытайтесь запустить Printaj. (Убедитесь, что вы запустили Exe2bin перед запуском Printaj. В противном случае будет выполняться .EXE версия Printaj, которая, несомненно, не даст вам того результата, который вы ожидали.)

Когда вы будете удовлетворены сделанным, используйте Debug, чтобы разассемблировать программу и посмотреть, как ассемблер соединяет две процедуры вместе. Помните, что мы можем считывать полученный файл в Debug, напечатав его название как часть командной строки. Например, мы можем напечатать "DEBUG PRINTAJ.COM", и после этого получим следующее:

```
3985:0100      B241      MOV     DL,41  
3985:0102      B90A00    MOV     CX,000A  
3985:0105      E80400    CALL    010C  
3985:0108      E2FB      LOOP   0105  
3985:010A      CD20      INT     20  
3985:010C      B402      MOV     AH,02  
3985:010E      CD21      INT     21  
3985:0110      FEC2      INC     DL  
3985:0112      C3        RET
```

Наша программа красива и экономно использует память, промежуток между процедурами отсутствует.

```

0100  MOV  DL,41
0102  MOV  CX,0A
0105  CALL 010C
0108  LOOP 0105
010A  INT  20

      ^
      |

010C  MOV  AH,02
010C  INT  21
0110  INC  DL
0112  RET

```

Рис. 9.1. Транслятор ассемблера MASM соединяет процедуры без промежутков

Процедуры, выводящие на экран шестнадцатеричные числа

Ранее в главе 5 мы уже рассматривали процедуры с шестнадцатеричным выводом, где узнали о том, как печатать шестнадцатеричное число, и в главе 7, где смогли упростить программу, используя процедуру печати одной шестнадцатеричной цифры. Теперь мы собираемся добавить еще одну процедуру печати одного шестнадцатеричного символа. Почему? Давайте пока назовём это предусмотрительностью.

Используя процедуру WRITE_HEX_DIGIT для вывода символа на экран, мы сможем изменять способ, которым она печатает символ, не затрагивая остальной программы. В дальнейшем мы будем изменять эту процедуру несколько раз. Введите следующую программу в файл VIDEO_IO.ASM:

Листинг 9.2. Новый файл VIDEO_IO.ASM.

```

CODE_SEG      SEGMENT
    ASSUME     CS:CODE_SEG
    ORG        100h
TEST_WRITE_HEX PROC    NEAR
    MOV        DL,3Fh      ;Тестировать с 3Fh
    CALL       WRITE_HEX
    INT        20h         ;Возврат в DOS
TEST_WRITE_HEX ENDP

```

```

PUBLIC WRITE_HEX
;
;Процедура преобразует байт в регистре DL в шестн. форму и
;записывает две цифры в текущей позиции курсора
; DL Байт, преобразуемый в шестн. форму.
; Используется: WRITE_HEX_DIGIT
;
WRITE_HEX PROC NEAR ;Точка входа
    PUSH CX ;Сохранить значения регистров,
;использованных в этой процедуре
    PUSH DX
    MOV DH,DL ;Скопировать байт
    MOV CX,4 ;Взять старшие 4 бита в DL
    SHR DL,CL
    CALL WRITE_HEX_DIGIT ;Записать первую шестн. цифру
    MOV DL,DH ;Взять младшие 4 бита в DL
    AND DL,0Fh ;Удалить старшие 4 бита
    CALL WRITE_HEX_DIDIT ;Записать вторую шестн. цифру
    POP DX
    POP CX RET
WRITE_HEX ENDP
PUBLIC WRITE_HEX_DIGIT
;
;Процедура преобразует 4 младших бита DL в шестн. цифру и
;выводит её на экран
; DL Младшие 4 бита есть число выводимое в виде
;шестн. цифры
; Используется: WRITE_CHAR
;
WRITE_HEX PROC NEAR
    PUSH DX ;Сохранить значение регистра
    CMP DL,10 ;Значение DL <10?
    JAE HEX_LETTER ;Нет, преобразовать в букву
    ADD DL,"0" ;Да, преобразовать в цифру
    JMP Short WRITE_DIGIT ;Сейчас запишем этот символ
HEX_LETTER:
    ADD DL,"A"-10 ;Преобразовать в шестн. букву
WRITE_DIGIT
    CALL WRITE_CHAR ;Вывести букву на экран
    POP DX ;Восстановить старое значение AX
    RET
WRITE_HEX_DIGIT ENDP
PUBLIC WRITE_CHAR
;
;Процедура печатает символ на экране, применяя вызов функции
;DOS

```

```
; DL Байт, выводимый на экран
;
WRITE_CHAR PROC NEAR
    PUSH    AX
    MOV     AH, 2          ;Вызов вывода символа
    INT     21h           ;Выводимый символ в регистре DL
    POP     AX             ;восстановить старое значение в AX
    RET
WRITE_CHAR ENDP
CODE_SEG ENDS
END TEST_WRITE_HEX
```

Функция DOS, выводющая символы на экран, обрабатывает некоторые из них особым образом. Например, при выводе на экран символа 07, компьютер подаст звуковой сигнал, а сам символ (изображение небольшого бриллианта) на экран выведен не будет. Новую версию WRITE_CHAR мы увидим в Части III, когда узнаем подробности о подпрограммах. Сейчас, однако, мы будем использовать для печати символов функцию DOS.

Новый псевдооператор PUBLIC написан здесь для будущего использования: мы будем применять его в главе 13, когда начнём осваивать модульное структурирование программ. PUBLIC сообщает ассемблеру о том, что необходимо сгенерировать дополнительную информацию для редактора связей, который позволяет собрать вместе части программы, размещённые в различных исходных файлах, в одну программу. И PUBLIC информирует ассемблер о том, что процедура, имя которой следует после псевдооператора PUBLIC, должна быть сделана общей, то есть доступной для процедур в других файлах.

Итак, Video_io содержит три процедуры для печати байта в виде шестнадцатеричного числа, а небольшая основная программа предназначена для тестирования этих процедур. По мере того, как мы будем создавать Dskpatch, мы добавим к файлу ещё некоторые процедуры и, к концу этой книги, в VIDEO_IO.ASM будет содержаться достаточно много процедур общего назначения.

Процедура TEST_WRITE_HEX, которую мы включили сюда, делает именно то, что говорит - она предназначена для проверки процедуры WRITE_HEX, которая, в свою очередь, использует WRITE_HEX_DIGIT и WRITE_CHAR.

Как только мы убедимся в том, что эти три процедуры работают правильно, мы удалим TEST_WRITE_HEX из VIDEO_IO.ASM.

Создайте .COM-версию Video_io и используйте Debug для тщательной проверки WRITE_HEX. Измените 3Fh по адресу 101h на каждое из граничных условий, которые мы испытывали в главе 5, когда использовали "G" для запуска TEST_WRITE_HEX.

Мы будем использовать достаточно много простых тестовых программ, проверяющих написанные нами процедуры. Таким образом, мы сможем создать программу из отдельных отлаженных элементов. Метод такого "восходящего" программирования значительно быстрее и проще, так как позволяет производить отладку отдельных частей, а не всей программы сразу.

Начала модульного конструирования программ

Обратите внимание, что в начале каждой процедуры в Video_io мы поместили блок комментария, кратко характеризующий её действие. Но ещё важнее то, что в этих комментариях указаны регистры, используемые в процедуре для получения и передачи информации, а также имена тех процедур, которые она сама вызывает. Как одна из возможностей модульного подхода, блок комментариев позволяет использовать любую процедуру, лишь взглянув на её описание. Нет необходимости снова разбираться с тем, как работает эта процедура. Отметим также, что подобный подход позволяет при изменениях программы ограничиться только изменением процедур.

Мы также использовали инструкции PUSH и POP для сохранения и восстановления состояния регистров, которые применяли внутри каждой процедуры. Мы будем выполнять такую операцию для каждой написанной процедуры, за исключением тестовых процедур. Этот подход также является частью модульного стиля.

Напоминаем, что мы сохраняем и восстанавливаем состояние каждого задействованного регистра для того, чтобы не беспокоиться о нежелательном взаимодействии между процедурами, пытающимися бороться между собой за право распоряжаться небольшим числом регистров микропроцессора 8088. Каждая

___ Структура программы ___

процедура свободна использовать столько регистров, сколько захочет, если она восстанавливает их перед инструкцией RET. Это небольшая цена за существенное упрощение. Кроме того, без сохранения и восстановления регистров задача переписывания процедуры была бы умопомрачительно сложной. Можете быть уверены, что в процессе её решения вы бы существенно подорвали свое здоровье.

Мы также будем по мере возможности создавать несколько небольших процедур вместо того, чтобы писать одну большую. Это тоже поможет упростить задачу программирования, хотя иногда нам всё же придется писать длинные процедуры, когда конструкция станет особенно сложной.

Эти методы и идеи будут более полно раскрыты в последующих главах. В следующей главе, например, мы добавим в Video_io ещё одну процедуру, которая берёт слово из регистра DX и печатает соответствующее число в десятичном виде на экране.

Структура программы

Как мы видели в этой и в предыдущей главах, ассемблер заставляет нас помещать определённое количество строк в качестве заголовка программ, которые мы пишем. Другими словами, нам нужно каждый раз записывать несколько псевдооператоров, которые сообщают ассемблеру основную информацию. В качестве рекомендации на будущее ниже приведён абсолютный минимум, необходимый для программ, которые вы пишете:

CODE_SEG	SEGMENT
ASSUME	CS:CODE_SEG
ORG	100h
Некая_процедура	PROC NEAR
.	
.	
.	
INT	20h
Некая_процедура	ENDP
CODE_SEG	ENDS
END	Некая_процедура

В следующих главах мы добавим несколько новых псевдооператоров в структуру программы, но вы

можете использовать её и в таком виде в качестве начальной точки для создания программ. Или, что ещё лучше, для этой цели вы можете использовать некоторые из программ и процедур, приведённых в этой книге.

Итог

Мы делаем реальные успехи. В этой главе мы научились писать процедуры на языке ассемблера. Теперь мы будем использовать процедуры постоянно, и, используя небольшие процедуры, мы сделаем наши программы лучше управляемыми.

Мы узнали, что процедура начинается с определения PROC и заканчивается псевдооператором END. Мы переписали PRINT_A_J заново, чтобы проверить знание процедур, затем перешли к переписыванию программы для вывода на экран шестнадцатеричного числа - на этот раз с дополнительной процедурой. Теперь, когда мы научились легко управляться с процедурами, почему бы не разбить программу на несколько процедур, фактически, мы увидели, что есть много причин для использования нескольких маленьких процедур вместо одной большой.

В конце этой главы мы коротко поговорили о модульном конструировании программ, использование которого сэкономит уйму времени и сил. Модульные программы будут проще для написания, чтения и внесения в них изменений, чем программы, базирующиеся на порядке устаревшей "логике спагетти": с длинными процедурами и большим количеством взаимодействий между ними.

Теперь мы готовы к созданию очередной полезной процедуры. Затем в главе 11 мы глубже изучим сегменты. И, начиная оттуда, будем реально использовать приёмы модульного конструирования.

Глава 10. Вывод на экран десятичных чисел

Мы обещали, что напомним процедуру, которая берёт из регистра слово и печатает его в десятичном представлении. WRITE_DECIMAL использует некоторые новые трюки - способы, заключающиеся в том, чтобы сохранить байт сначала в одном месте памяти, а через несколько микросекунд в другом. Может показаться,

что эти трюки не оправдывают затраченных на них усилий. Но если вы их запомните, то найдете, что можете их использовать для сокращения размеров и повышения быстродействия программ. Используя эти трюки, мы также узнаем о двух новых типах логических операций в дополнение к инструкции AND, описанной в главе 5. Сейчас мы рассмотрим процесс перевода слова в десятичные цифры.

Вспоминаем перевод чисел

Деление - это ключ к переводу слова в десятичные цифры. Напоминаем, что инструкция DIV подсчитывает как частное от деления, так и его остаток. Так что вычисление $12345/10$ даст частное 1234, и 5 в остатке. В этом примере 5-это самая правая цифра. И, если мы опять разделим число на 10, то получим следующую цифру слева. Последовательное деление выделяет цифры справа налево, каждый раз помещая их в остаток.

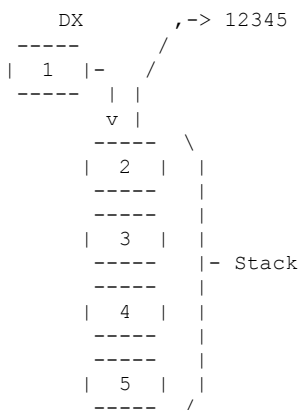


Рис. 10.1. Помещение цифр в стек изменяет порядок их следования на прямо противоположный

Конечно, цифры поступают в обратном порядке, но в программировании на ассемблере имеется для этого специальное приспособление. Помните стек? Он похож на стопку подносов в столовой: когда кладут верхний поднос, то нижний опускается, а первым извлекается из стопки верхний. Если мы подставим цифры вместо подносов и поместим цифры одну над другой, в той последовательности, в которой они получаются в остатке, то мы получим именно то, что нам нужно. Мы сможем "вытащить" обратно цифры в правильном порядке.

Верхняя цифра в стеке является первой цифрой нашего числа, остальные цифры находятся под ним. Так что если мы будем помещать остатки в стек в том порядке, в котором их получаем, а печатать их в той

последовательности, в которой они появляются из стека, то цифры будут расположены в правильном порядке.

Нижеследующая программа является законченной программой для печати числа в десятичном представлении. Как мы уже отмечали, в этой процедуре скрываются некоторые трюки. Мы вскоре к ним вернёмся, но давайте сначала попробуем работает ли вообще WRITE_DECIMAL, перед тем как беспокоиться о том, как она работает.

Поместите WRITE_DECIMAL в VIDEO_IO.ASM вместе с процедурами печати байта в шестнадцатеричном виде. Убедитесь, что вы поместили WRITE_DECIMAL после TEST_WRITE_HEX, которую мы заменим на TEST_WRITE_DECIMAL. Для экономии труда WRITE_DECIMAL использует WRITE_HEX_DIGIT для перевода одной тетрады (четырёх битов) в цифру.

Листинг 10.1. Дополнения к VIDEO_IO.ASM.

```

PUBLIC  WRITE_DECIMAL
;Процедура записывает 16-битовое число без знака в
;десятичной форме.
; DX      N: 16-битовое, беззнаковое число
; Используется:      WRITE_HEX_DIGIT
WRITE_DECIMAL      PROC    NEAR
    PUSH    AX          ; Сохранить используемые здесь регистры
    PUSH    CX
    PUSH    DX
    PUSH    SI
    MOV     AX,DX
    MOV     SI,10        ;Делить на 10 применяя SI
    XOR     CX,CX        ;Счёт цифр, помещаемых в стек
NON_ZERO:
    XOR     DX,DX        ;Установить старшее слово N в 0
    DIV     SI            ;вычислить N/10 и (N mod 10)
    PUSH    DX           ;Поместить одну цифру в стек
    INC     CX
    OR      AX,AX        ;N = 0?
    JNE     NON_ZERO     ;Нет оператора, продолжить
WRITE_DIGIT_LOOP:
    POP     DX           ;Взять цифры в обратном порядке
    CALL    WRITE_HEX_DIGIT
    CALL    WRITE_DIGIT_LOOP
END_DECIMAL:
    POP     SI

```

```
POP    DX
POP    CX
POP    AX
RET
WRITE_DECIMAL    ENDP
```

Обратите внимание на то, что мы используем новый регистр, SI (сокр. от англ. "Source Index" - индекс источника). Позже мы увидим, почему ему было дано такое имя, и мы также встретимся с его братом, регистром DI (от англ. "Destination Index" - индекс назначения). Оба регистра имеют специальное назначение, но они могут также использоваться и как регистры общего назначения. Так как WRITE_DECIMAL нуждается в четырех регистрах общего назначения, мы применим SI, хотя могли бы использовать и BX, просто для того, чтобы показать, что SI (или DI) могут в случае необходимости служить и регистрами общего назначения.

Перед тем, как мы опробуем новую процедуру, нам нужно внести пару изменений в VIDEO_IO.ASM. Во-первых, нам надо удалить процедуру TEST_WRITE_HEX и вставить эту тестовую процедуру на её место:

Листинг 10.2. Замените TEST_WRITE_HEX в VIDEO_IO.ASM на эту процедуру.

```
TEST_WRITE_DECIMAL    PROC    NEAR
    MOV    DX,12345
    CALL    WRITE_DECIMAL
    INT     20h          ;Возврат в DOS
TEST_WRITE_DECIMAL    ENDP
```

Эта процедура тестирует WRITE_DECIMAL с помощью числа 12345 (которое ассемблер переведёт в слово 3039h.)

Во-вторых, нам нужно изменить оператор END в конце VIDEO_IO.ASM, чтобы читалось "END TEST_WRITE_DECIMAL", так как теперь TEST_WRITE_DECIMAL является главной процедурой.

Внесите эти изменения и сделайте с VIDEO_IO все необходимые шаги, чтобы получить .COM-файл, а затем посмотрите, как он работает. Если он не работает или работает с ошибками, то проверьте исходный файл. Если у вас есть авантюристическая жилка, попробуйте отыскать ошибку с помощью Debug. В конце концов, ведь именно для этого программа Debug и предназначена.

Некоторые трюки

Трюки, которые прячутся в WRITE_DECIMAL, использовались авторами процедур ROM BIOS, с которыми мы встретимся в главе 17. Первый - это эффективная инструкция для обнуления регистра. Она не намного эффективнее, чем "MOV AX,0", и, возможно, не оправдывает усилий, на неё затраченных, но этот трюк широко используется, поэтому мы его здесь и приводим.

Инструкция:

```
XOR      AX, AX
```

обнуляет регистр AX. Чтобы понять, как это происходит, нам надо узнать о логической операции XOR (исключающее "или").

Исключающее "или" похоже на обычное "или" (которое мы рассмотрим следующим), но результат проведения этой логической операции:

XOR		0	1

0		0	1
1		1	0

будет "истинно", только тогда, когда один бит равен единице ("истинно"), а не оба. Таким образом, если мы проводим операцию XOR над самим числом, то получим нуль:

	1011	0101
XOR	1011	0101

	0000	0000

Вот и весь трюк. Мы в этой книге не будем использовать XOR для других целей, но думаем, что вы найдете эту операцию интересной.

Кроме того, вы также увидите, что существуют и другие способы обнуления регистра. Вместо инструкции XOR, чтобы установить регистр AX в нуль, мы могли также применить:

```
SUB      AX, AX
```

Теперь о другом способе. Он немного сложнее предыдущего, и использует другую логическую операцию - OR ("или").

Нам надо проверить регистр AX, чтобы выяснить равен ли он нулю. Для этого можно было бы при-

менить "CMP AX,0", но вместо этого мы используем новый способ: он оригинальнее и немного эффективнее. Итак, мы пишем "OR AX,AX" и после этой инструкции помещаем условный переход JNE (от англ. "Jump if Not Equal" - перейти, если не равно). (Мы также могли использовать JNZ - перейти, если не нуль.) Инструкция OR, как и любая другая математическая инструкция, устанавливает флаги, включая флаг нуля. Как и AND, OR - логическая инструкция. Но результат выполнения OR над двумя битами будет "истинно" только в том случае, если оба бита установлены в "истинно" :

OR		0	1

0		0	1
1		1	1

Логическая операция OR, выполненная по отношению к одному и тому же числу, даёт в результате это же число:

	1011	0101
OR	1011	0101

	1011	0101

Инструкция OR также полезна при установлении одного бита в байт. Например, мы можем установить бит 3 в числе, которое только что использовали:

	1011	0101
OR	0000	1000

	1111	1101

До конца этой книги мы рассмотрим ещё несколько способов, но эти два приведены только для забавы.

Внутренняя работа

Чтобы увидеть, как WRITE_DECIMAL выполняет свою задачу, изучите листинг; мы не будем здесь вдаваться в подробности. Отметим лишь ещё ряд вопросов.

Прежде всего регистр CX используется для подсчёта того, сколько цифр помещено в стек, чтобы знать, сколько вынимать. Регистр CX выбран не случайно, так как мы можем построить цикл с помощью инструкции LOOP и использовать CX для хранения счётчика повторений. Выбор CX делает цикл вывода

цифр (WRITE_DIGIT_LOOP) почти что тривиальным, так как инструкция LOOP применяет регистр CX в качестве счётчика. Такое решение является почти стандартным: использовать CX для хранения значений счётчика.

Кроме того, будьте осторожны при проверке граничных условий. Первое граничное условие 0 проблемы не представляет. Другое граничное условие - это 65535 или FFFFh, которое вам лучше проверять с помощью Debug. Загрузите VIDEO_IO.COM в Debug, напечатав "DEBUG VIDEO_IO.COM", и измените 12345 (3039h) по адресу 101h на 65535 (FFFFh). (WRITE_DECIMAL работает с беззнаковыми числами. Если хотите, можете попробовать написать версию для работы с числами со знаком).

Возможно, вы заметили здесь одну особенность, связанную не с программой, а с самим микропроцессором 8088. Debug работает в основном с байтами (по крайней мере, команда "E"), а необходимо изменить всё слово. Сложность заключается в том, что 8088 хранит байты в другом порядке. Вот разассемблированный вариант инструкции MOV:

```
3985:0100  BA3930  MOV     DX, 3039
```

Вы можете сказать, глядя на строку из распечатки "BA3930", что байт, находящийся по адресу 101h это 39h, а по адресу 102h - 30h (BA это инструкция MOV). Эти два байта составляют 3039h, но записаны в обратном порядке. Путаница? На самом деле это логичный порядок и он станет понятным после короткого объяснения.

Слово состоит из двух частей: младшего байта и старшего байта. Младший байт является наименее значащим (самым младшим) байтом (39h в 3039h), остальная часть-старший байт (30h). Это означает, что младший байт находится по младшему адресу в памяти, по отношению к старшему байту. (Некоторые компьютеры реверсируют эти два байта, что может привести к путанице, если вы применяете несколько разных компьютеров.)

Попробуйте использовать различные числа для слова, находящегося по адресу 101h, и вы увидите, как работает это устройство хранения. Примените TEST_WRITE_DECIMAL для проверки того, правильно ли

вы всё делаете или разассемблируйте первую инструкцию.

```
MOV  DX, 3039
-----
-
0102: 30 |- 3039h
0101: 39 |
-
0100  BA <- MOV instruction
```

Рис. 10.2. Микропроцессор 8088
хранит в памяти
первым младший байт числа

Итог

Мы добавили к нашему репертуару несколько новых инструкций и способов, а также узнали о двух новых регистрах, SI и DI, которые можно использовать как регистры общего назначения. Они также полезны и для других целей, которые мы рассмотрим в следующих главах.

Мы узнали о логических инструкциях XOR и OR, позволяющих производить логические операции с отдельными битами двух байтов или слов. В процедуре WRITE_DECIMAL мы использовали инструкцию "XOR AX,AX" для обнуления регистра AX, а "OR AX,AX" как "обходной маневр" для записи эквивалента "CMP AX,0" для того, чтобы проверить, равен ли регистр AX нулю.

И, наконец, при проверке граничных условий новой процедуры WRITE_DECIMAL мы узнали о том, как микропроцессор 8088 хранит в памяти слово.

Создана ещё одна процедура общего назначения, WRITE_DECIMAL, которую мы сможем использовать в будущем.

Отдыхитесь. Дальше пойдут немного другие главы. В главе 11 подробно рассматриваются сегменты, которые, пожалуй, являются наиболее сложной частью микропроцессора 8088, поэтому глава может оказаться тоже непростой. После сегментов мы произведём небольшую коррекцию курса и вернёмся обратно к изучению того, с чем мы собирались работать, используя программу Dskpatch. Мы немного прозондируем диски и узнаем о секторах, треках и других подобных вещах.

После этого мы можем составить простой план предварительных версий Dskpatch.

Заодно у вас будет шанс увидеть, как создаются большие программы. Программисты не пишут и отла-

живают программы целиком, они поступают иначе - пишут отдельные части программ, затем тестируют работоспособность каждой части и, наконец, соединяют их вместе. Программирование именно таким образом значительно упрощается. Мы частично использовали этот метод при написании и тестировании WRITE_HEX и WRITE_DECIMAL, для которых тестовые программы были простыми. В дальнейшем тестовые программы будут более сложными, но также и более интересными.

Глава 11. Сегменты

В предыдущих главах мы сталкивались с несколькими псевдооператорами, работающими с сегментами. Настало время рассмотреть сами сегменты и то, каким образом микропроцессор 8088 управляет целым Мегабайтом (1.048.576 байт) памяти. Разобрав этот вопрос, мы поймём, почему для сегментов нужны свои собственные ассемблерные псевдооператоры, а в последующих главах будем применять и разные сегменты (до этого использовался только один). Затем, в главе 13 мы научимся модульному конструированию программ и увидим, как сгруппировать сегменты вместе в .COM-файл.

Давайте начнем с того, как микропроцессор 8088 вычисляет 20-разрядный адрес, необходимый для адресации мегабайта памяти.

Секционирование памяти микропроцессором 8088

Сегменты являются единственной частью микропроцессора 8088, которую мы ещё не разобрали, и они, пожалуй, представляют собой наибольшую трудность при изучении этого микропроцессора. Фактически, сегменты это то, что мы называем "крудж" (от. англ. "kludge"): - компьютерное (в данном случае) приспособление для временного устранения проблемы.

Проблема в данном случае заключается в адресации более 64К памяти - предела, который устанавливается длиной слова, так как 65535-это наибольшее число, которое может содержать одно слово. Специалисты фирмы Intel, которая является разработчиком микропроцессора 8088, для решения этой проблемы использовали сегменты и регистры сегментов,

и из-за этого процесс адресации в микропроцессоре 8088 немного усложнился.

Ранее мы не интересовались этой проблемой. Мы использовали регистр IP для хранения адреса следующей инструкции, которую должен выполнить 8088, с тех пор как познакомились с Debug в главе 2. Возможно вы помните, что мы говорили, что фактический (абсолютный) адрес формируется из двух регистров: CS и IP. Но мы пока не объяснили, как это происходит. Теперь мы это сделаем.

Хотя адрес формируется из двух регистров, микропроцессор 8088 не формирует для адреса числа, состоящего из двух слов. Если бы регистры CS:IP использовались как 32-битовое число (два 16-битовых числа, одно за другим), 8088 адресовал бы около миллиарда байт, намного больше того миллиона байт, которые он адресует на самом деле. Метод микропроцессора 8088 немного более сложный: регистр CS содержит начальный адрес сегмента кода, где сегмент - это 64 Кбайт памяти. Вот так он работает.

Как вы видите из рис. 11.1, микропроцессор 8088 разбивает память на много перекрывающихся сегментов, и каждый новый сегмент начинается через каждые 16 байт. Первый сегмент (сегмент 0) начинается в памяти с ячейки 0; второй (сегмент 1) начинается с 10h (16); третий начинается с 20h (32), и так далее.

Абсолютный адрес - это $CS \cdot 16 + IP$. Например, если регистр CS содержит 3FA8, а IP-D017, то абсолютный адрес будет равен:

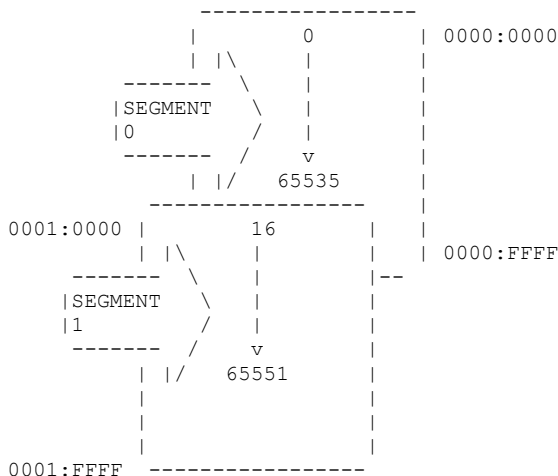


Рис. 11.1. Перекрывающиеся сегменты начинаются каждые 16 байт и могут достигать 65535 байт длины.


```

CS*16      : 0011 1111 1010 1000 0000
IP         :      1101 0000 0001 0111
-----
              0100 1100 1010 1001 0111
  
```

Мы умножили на 16, сдвинув CS влево на четыре бита и приписав справа нули.

Такой способ адресации более 64Кбайт памяти кажется довольно странным, однако он работает. Вскоре мы увидим, как это происходит на практике.

```

<== |0|0|1|1|1|1|1|1|1|0|1|0|0|0|      <- Segment (CS)
      - - - - -
      +
      |1|1|0|1|0|0|0|0|0|0|0|0|1|0|1|1|1| <- Offset (IP)
      - - - - -
-----
0 1 0 0 1 1 0 0 1 0 1 0 1 0 1 1 1
  
```

Рис. 11.2. Абсолютный адрес, на который указывает CS:IP равен CS*16 + IP.

Микропроцессор 8088 имеет четыре регистра сегментов: CS (сокр. от англ. "Code Segment"-сегментный регистр кодов), DS ("Data Segment"-сегментный регистр данных), SS ("Stack Segment"-сегментный регистр стека) и ES ("Extra Segment"-регистр дополнительного сегмента). Регистр CS, который мы уже рассматривали ранее, используется микропроцессором для хранения номера сегмента, в котором содержится следующая инструкция. Соответственно, в регистре DS хранится номер сегмента, в котором 8088 должен искать данные, а в SS - номер сегмента, в котором 8088 стек.

Перед тем, как двигаться дальше, рассмотрим небольшую программу, которая очень отличается от виденных ранее тем, что использует уже два сегмента. Введите эту программу в файл TEST_SEG.ASM:

Листинг 11.1. Программа TEST_SEG_ASM.

```

CODE_SEG      SEGMENT
  ASSUME      CS:CODE_SEG
TEST_SEGMENT  PROC NEAR
  MOV         AH,4Ch      ;Запрос функции выхода в DOS
  INT         21h         ;Выход в DOS
TEST_SEGMENT  ENDP
CODE_SEG      ENDS
STACK_SEGMENT SEGMENT STACK
  DB          10 DUP ("Stack ") ;Три пробела после слова Stack
STACK_SEGMENT ENDS
END           TEST_SEGMENT
  
```

Оттранслируйте и проведите компоновку Test_seg, но не генерируйте для него .COM-файл. В результате мы получим файл TEST_SEG.EXE, который немного отличается от .COM-файла.

Примечание: Для выхода в DOS из .EXE-файлов мы используем другой метод. Для .COM-файлов "INT 21h" работает прекрасно, но она не работает с .EXE-файлами, потому что организация сегментов значительно отличается, что мы увидим в этой главе позже.

При использовании Debug для работы с .COM-файлом, он помещал во все регистры сегментов номер того, в котором на смещении 100h начиналась программа. Первые 256 байт используются для хранения различной информации, которая нас не очень интересует, но мы тем не менее рассмотрим кратко часть этой области памяти.

Попробуйте загрузить TEST_SEG.EXE в Debug чтобы увидеть, что же происходит с регистрами в .EXE-срайле:

```
A:>DEBUG TEST_SEG.EXE
-R
AX=0000 BX=0000 CX=0080 DX=0000 SP=0050 BP=0000 SI=0000 DI=0000
DS=3985 ES=3985 SS=3995 IP=0000 NV UP DI PL N2 NA PO NC
3995:0000 CD20 INT 20
```

Значения регистров SS и CS отличаются от значений регистров DS и ES.

В программе определены два сегмента. STACK_SEGMENT - сегмент в котором размещён стек (об этом говорит слово "STACK" после слова "SEGMENT"). Мы определили длину стека в 80 байт: Инструкция "DB 10 DUP ("Stack ")" сообщает ассемблеру о необходимости перевести строку в байты и поместить в память десять раз. DB (сокр. от англ. "Define Byte" - определить байт) объясняет ассемблеру, что мы определяем байты памяти. Здесь мы инициализируем стек, десять раз повторяя ASCII-код слова "Stack" и три пробела. Код этой строки будет 53 74 61 63 6B 20 20 20, поэтому если мы посмотрим на сегмент стека, то увидим эти числа, повторенные десять раз. Сделайте дамп этой области памяти, используя следующую команду Debug: -D SS:0. Это означает, что Debug должна дампить память, начиная со смещения 0 в сегменте стека (SS:0):

```
-D SS:0
3996:0000 53 74 61 63 68 20 20 20-53 74 61 63 68 20 20 20 Stack Stack
3996:0010 53 74 61 63 68 20 20 20-53 74 61 63 68 20 20 20 Stack Stack
3996:0020 53 74 61 63 68 20 20 20-53 74 61 63 68 20 20 20 Stack Stack
```

```

5996:0030 53 74 61 63 68 20 20 20-53 74 61 63 68 20 20 20 Stack Stack
3996:0040 83 74 61 63 68 20 20 20-53 74 61 63 68 20 00 00 Stack Stack..
3996:0050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
.
.
.
|      |
SS     SP

```

Адрес вершины сегмента получается из SS:SP.
 SP (сокр. от англ. "Stack Pointer" - указатель стека), как IP и CS для кода, это смещение в текущем сегменте стека.

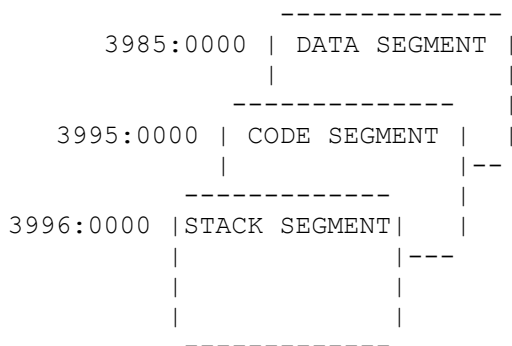


Рис. 11.3. Структура размещения TEST_SEG.EXE в памяти

Фактически "вершина стека" (англ. top of stack)- это неправильный термин, так как стек растёт от старших адресов к младшим. Таким образом, вершина стека на самом деле находится в памяти внизу стека, а новые точки входа стека расположены в убывающем порядке в сторону младших адресов. В нашем случае SP равен 50h, то есть 80 в десятичной системе, так как размер стека определён в 80 байт. Пока в стек ничего не помещено и поэтому его вершина находится в памяти там же, где мы поместили стек: 50h.

Теперь, когда вы знаете, как найти стек, то, возможно, захотите посмотреть, как он изменяется в программах предыдущих глав. Продолжим работу с примером уже в Debug.

Обратите внимание на то, что сегмент стека ("Stack Segment") имеет номер 3996 (у вас он может быть другим), в то время как сегмент кода ("Code Segment") 3995, то есть он расположен в памяти на 16 байт раньше.

Это означает, что если мы произведём разassemблирование, начиная с CS:0, мы увидим нашу программу (инструкцию "INT 20h"), за которой следует 14 байт, равных нулю (инструкция "INT 20h" занимает два байта), и затем мы видим байты из сегмента стека. Мы также видим символы слова "Stack", после которых записаны три пробела, в разassemблированном виде:

```
-U CS:0
3995:0000    CD20      INT  20
3995:0002    0000      ADO  [BX+SI],AL
3995:0004    0000      ADD  [BX+SI],AL
3995:0006    0000      ADD  [BX+SI],AL
3995:0008    0000      ADD  [BX+SI],AL
3995:000A    0000      ADD  [BX+SI],AL
3995:000C    0000      ADD  [BX+SI],AL
3995:000E    0000      ADD  [BX+SI],AL
3995:0010    53        PUSH  BX
3995:0011    7461      JZ    0074
3995:0013    63        DB    63
3995:0014    6B        DB    B
3995:0015    2020      AND  [BX+SI],AH
3995:0017    205374    AND  [BP+DI+74],DL
3995:001A    61        DB    1
3995:001B    63        DB    3
3995:001C    6B        DB    8
3995:001D    2020      AND  [BX+SI],AH
3995:001F    205374    AND  [BP+DI+74],DL
```

Как мы и ожидали, число 53h (ASCII-код "S") является первой буквой в стеке - на смещении 10h (16) от начала нашего сегмента кода.

Взглянув на распечатку регистров, вы можете заметить, что регистры ES и DS содержат 3985h, на 10h меньше, чем начало сегмента программы (3995h). Умножив на 16, чтобы получить число байт, мы можем видеть, что перед началом программы 100h (или 256) байт. Точно такое же количество памяти помещается перед .COM-файлом.

Кроме всего прочего, эта 256-байтная рабочая область (англ. "scratch area") в начале программ содержит символы, которые мы печатаем после имени программы. Например:

```
A:>DEBUG TEST_SEG.EXE

And now for some characters we'll see in the memory dump

-D    DS:80
```

```

3996:0080  39 20 41 6E 64 20 6E 6F - 77 20 66 6F 72 20 73 6F 9 And now for so
3996:0090  6D 65 20 63 68 61 72 61 - 63 74 65 72 73 20 77 65 me characters we
3996:00A0  27 6C 6C 20 73 65 65 20 - 69 6E 20 74 68 65 20 6D 'll see in the m
3996:00B0  65 6D 6F 72 79 20 64 75 - 6D 70 0D 20 6D 65 6D 6F emory dump, memo
3996:00C0  72 79 20 64 75 6D 70 0D - 00 00 00 00 00 00 00 00 ry dump.....

```

Первый байт сообщает, что мы напечатали 39h (или 57) символов, включая пробел после TEST_SEG.EXE. Мы не будем использовать эту информацию в книге, но она поясняет, для чего нужна такая большая рабочая область.

Примечание: "Рабочая область" в действительности называется PSP ("Program Segment Prefix" - префикс программного сегмента) и содержит информацию, используемую DOS. Другими словами, вы должны запомнить, что не можете использовать эту область памяти.

Рабочая область также содержит информацию, применяемую DOS при выходе из программы с помощью инструкций "INT 20h" или "INT 21h", функция 4Ch. Но по причинам, сейчас уже не совсем ясным, инструкция "INT 20h" требует, чтобы регистр CS указывал на начало рабочей области, что и сделано для .COM, но не для .EXE-программ. Это вопрос истории. Фактически инструкция выхода ("INT 21h", функция 4Ch) была добавлена к DOS, начиная с версии 2.00.

Код .COM-файлов всегда должен начинаться со смещения 100h в сегменте кода, чтобы оставалось место для 256-байтной рабочей области. К .EXE-файлам это не относится, их код начинается с IP = 0000 потому, что сегмент кода начинается через 100h байт после начала области в памяти.

Напоминаем, что в файлах в главе 10 в начале программ устанавливался псевдооператор "ORG 100h", выделяющий необходимые 100h байт. Псевдооператор "ORG 100h" устанавливает начало (англ. "origin") кода со 100h. Вот практически и всё, но мы будем продолжать использовать "ORG 100h", так как будем работать с .COM-программами до конца книги.

Мы представили здесь .EXE-файл только для того, чтобы вы получили понятие о сегментах. Позднее вы узнаете о них ещё больше, но, начиная отсюда, мы будем использовать .COM-файлы, так как они меньше и загружаются в память быстрее. Причину этого вы также узнаете, когда мы достигнем последней главы,

но сейчас давайте продолжим. Теперь мы займёмся псевдооператорами для сегментов.

Псевдооператоры сегментов

Мы рассмотрим здесь несколько псевдооператоров: SEGMENT, ENDS, ASSUME и псевдооператоры NEAR и FAR. Нам также потребуется поближе рассмотреть инструкции CALL и RET. После этого мы расширим знания об инструкции INT и увидим, чем она похожа на инструкцию CALL.

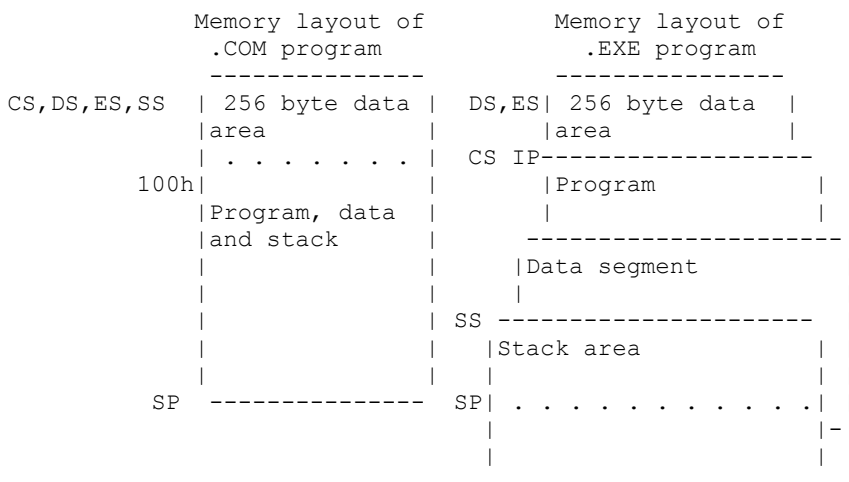


Рис. 11.4. Различие между .COM и .EXE-программами.

Но давайте все делать по порядку, и начнём с SEGMENT и ENDS.

Псевдооператоры SEGMENT и ENDS похожи на PROC и ENDP, с которыми мы познакомились в главе 9. Мы определяем сегмент, заключая часть исходного файла в пару псевдооператоров SEGMENT/ENDS также, как мы определяли процедуру, используя пару PROC/ENDP. Имя, стоящее перед псевдооператором SEGMENT, - метка.

Мы будем использовать эту метку в главе 13, когда разделим исходный файл на несколько отдельных файлов и два сегмента: сегмент данных и сегмент кодов. Применение двух сегментов позволяет легко отделить в памяти переменные от самой программы. Переменные в памяти подробнее будут рассмотрены также в главе 13, где мы заодно немного расширим наши познания о псевдооператоре SEGMENT. В этом вопросе существует множество деталей, но мы пока не будем терять на них время. Если вам потребуется

информация по этому вопросу, вы можете найти её в вашем руководстве по ассемблеру.

Псевдооператор ASSUME

Действие псевдооператора ASSUME немного сложнее, чем SEGMENT. Он сообщает ассемблеру информацию о сегментах и о том, как мы собираемся использовать сегментные регистры. Чтобы разобраться с ASSUME, нам надо понять, как ассемблер контролирует перемещение меток и имён переменных.

Каждый раз, когда вы создаете метку, например, процедуры (как WRITE_CHAR PROC NEAR) или переменной, ассемблер запоминает несколько параметров, связанных с именем: тип (процедура, байт, слово и т.д.), адрес имени и сегмент, где оно определено. С последним параметром связано применение ASSUME.

Ассемблер не распознает по умолчанию, что все процедуры программы находятся в одном сегменте. Во многих случаях для больших программ, таких, как например Lotus 1-2-3, это не верно. Такие программы используют несколько различных сегментов кодов. Поэтому в интересах единообразия, нам надо сообщать информацию ассемблеру в виде операторов ASSUME, которые объясняют, на какой сегмент какой именно регистр сегмента указывает.

Например, рассмотрим оператор ASSUME, который мы использовали в предыдущих главах:

```
ASSUME CS:CODE_SEG
```

Это выражение сообщает ассемблеру, что регистр CS указывает на сегмент кода, который мы назвали CODE_SEG. Без этой информации ассемблер в случае применения метки (например как в "CALL WRITE_CHAR"), выдаст сообщение "No or unreachable CS" (отсутствует или не достижим сегмент кода). Это означает, что он не знает, в каком сегменте мы в данный момент находимся.

Так как регистр CS всегда указывает на выполняемый код, то может показаться немного странным, что ассемблер "ругается" при отсутствии оператора ASSUME. Фактически, оператор ASSUME не был бы нужен, если бы не существовало нечто, называющееся переопределение сегмента.

___ "Ближние" и "дальние" вызовы процедур ___

Обычно микропроцессор 8088 считывает данные (например "MOV AL,SOME_VARIABLE") из сегмента данных (DS). Но он может также, используя переопределение сегмента, считывать информацию из любых других сегментов, например, из сегмента кода (CS). Именно поэтому ассемблеру необходим псевдооператор ASSUME: по его наличию он узнает, какой сегментный регистр использовать, при считывании или записи в память.

Не беспокойтесь, если вы еще не вполне поняли объяснение действия псевдооператора ASSUME. Мы будем использовать его минимально, пока не достигнем главы 29. В ней мы рассмотрим многосегментные программы и также расширим знания как о псевдооператоре ASSUME, так и переопределении сегментов.

Остальная информация в этой главе приведена только для ознакомления, мы не будем её использовать в дальнейшем. Вы можете пропустить следующие два раздела и прочитайте их позже, если вы встретитесь с трудностями или если вам не захочется возвращаться к программированию.

"Ближние" и "дальние" вызовы процедур

Давайте сделаем шаг назад и более подробно рассмотрим инструкции CALL, применяемые в предыдущих главах. А именно, давайте взглянем на короткую программу из главы 7, в которой впервые узнали об инструкции CALL. Тогда была написана короткая программа, которая выглядела следующим образом (без процедуры по адресу 200h):

```
3985:0100  B241  MOV    DL, 41
3985:0102  B90A00 MOV    CX, 000A
3985:0105  E8F800 CALL   0200
3985:0108  E2FB  LOOP   0105
3985:010A  CD20  INT     20
```

Взглянув на машинные коды слева, вы увидите, что инструкция CALL занимает всего три байта (E8F800) Первый байт (E8h) - это код инструкции CALL, а следующие два байта составляют смещение. Микропроцессор 8088 подсчитывает адрес процедуры, которую мы вызываем, складывая смещение 00F8h (помните, что 8088 хранит младший байт слова в памяти перед старшим байтом, поэтому мы поменяли их

местами) и адрес следующей инструкции (в нашей программе 108h). Итак, получаем $F8h + 108h = 200h$, что мы и ожидали.

Тот факт, что эта инструкция для смещения использует одно слово, означает, что вызываемая процедура находится в том же сегменте, который равен 64K. Так каким образом мы можем писать программы, такие как Lotus 1-2-3, превышающие 64Кбайт? Мы делаем это, используя FAR (дальние), а не NEAR (близкие) вызовы.

Близкие вызовы, как мы видели, ограничены сегментом. Другими словами, они изменяют только регистр IP, не затрагивая регистра CS. По этой причине они иногда называются внутрисегментными вызовами.

Но мы также можем применять и дальние (FAR) вызовы, изменяющие как регистр CS, так и IP. Такие вызовы часто называются межсегментными, потому что вызывают процедуры из других сегментов.

В соответствии с двумя версиями инструкции CALL существуют и две версии инструкции RET.

Как мы видели в главе 7, адрес возвращения, сохраняемый в стеке близким вызовом (NEAR CALL), состоит из одного слова. И соответствующая инструкция RET восстанавливает это слово из стека в регистр IP.

В случае использования дальнего вызова и возвращения (FAR CALL и RET), одного слова недостаточно, так как мы имеем дело с другим сегментом. Другими словами, нам нужно сохранять в стеке адрес возвращения, состоящий из двух слов: одно для указателя команд (IP) и другое для сегментного регистра кода (CS). Инструкция возврата из дальней процедуры (FAR RET) затем восстанавливает из стека два слова - одно для регистра CS, другое для IP.

Теперь мы подошли к вопросу, каким именно образом ассемблер распознает эти два CALL и RET? Когда он должен использовать FAR CALL (дальний вызов), а когда NEAR CALL (близкий вызов)? Этим распоряжаются псевдооператоры NEAR и FAR.

В качестве примера рассмотрим следующую программу:

```
PROC_ONE    PROC    FAR
    .
    .
    .
    RET
PROC_ONE    ENDP
```

"Ближние" и "дальние" вызовы процедур

```

PROC_TWO      PROC    NEAR
CALL          PROC_ONE
.
.
.
RET PROC_TWO  ENDP
    
```

Когда ассемблер встречается инструкцию "CALL PROC_ONE", он смотрит на определение PROC_ONE, которое в данном случае является "PROC_ONE PROC FAR". Определение процедуры сообщает ассемблеру о том, что данная процедура является близкой или дальней.

В случае процедуры типа NEAR ассемблер генерирует близкий вызов. И наоборот, он генерирует далекий вызов, если процедура, которую вы вызываете, имеет тип FAR. Другими словами, ассемблер использует определение вызываемой процедуры для определения нужного типа инструкции CALL.

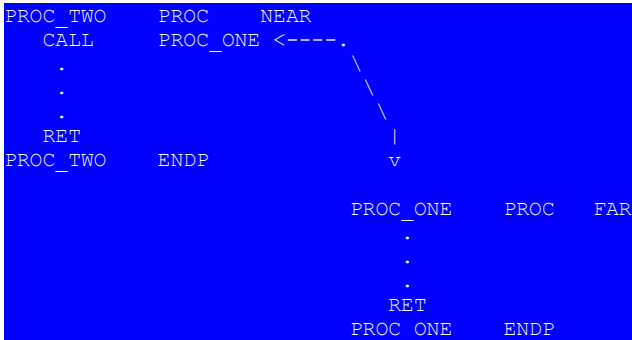


Рис. 11.5. Ассемблер выполняет FAR CALL

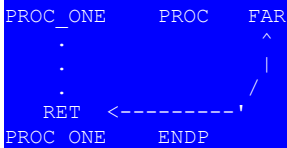


Рис. 11.6. Ассемблер выполняет FAR RET

Для инструкции RET в свою очередь ассемблер смотрит на определение процедуры, содержащей эту инструкцию. В программе инструкция RET для

PROC_ONE является типом FAR (FAR RET), потому что PROC_ONE объявлена процедурой типа FAR. Аналогично RET в процедуре PROC_TWO имеет тип NEAR (NEAR RET).

Подробнее об инструкции INT

Инструкция INT похожа на инструкцию CALL, но с небольшим отличием. Название INT произошло от слова "interrupt" - прерывание. Прерывание - это внешний сигнал, который заставляет микропроцессор 8088 выполнить процедуру обработки данного прерывания и затем вернуться к тому, что он делал до получения сигнала прерывания. Инструкция INT не прерывает микропроцессор 8088, но она обрабатывается, как будто оно было.

Когда микропроцессор 8088 получает прерывание, ему нужно сохранить в стеке больше информации, чем просто два слова, как это было с адресом возврата. Он должен сохранить значения флагов состояния - флаг переноса, флаг нуля, и так далее. Эти значения содержатся в одном слове, известном как "Flag Register" (регистр флагов), и 8088 сохраняет эту информацию в стеке перед адресом возврата. Объясним, почему нам надо сохранять флаги состояния.

Ваш IBM PC постоянно реагирует на большое количество различных прерываний. Микропроцессор 8088 IBM PC получает, например, сигнал прерывания от таймера 18,2 раза в секунду. Каждый из этих сигналов заставляет 8088 приостановить выполнение текущей программы и выполнить процедуру подсчета числа импульсов таймера.

Предположим, что прерывание произошло между выполнением двух инструкций программы:

```
CMP     AH, 2
JNE     NOT_2
```

Допустим, AH = 2, так что после инструкции CMP будет установлен флаг нуля, и это означает, что инструкция JNE не осуществит переход на метку NOT_2.

Теперь представьте, что прерывание таймера происходит между этими двумя инструкциями. Это означает, что 8088 займется обработкой прерывания

перед тем, как проверить флаг нуля (с помощью инструкции JNE). Если микропроцессор не сохранил и затем не восстановил регистр флагов, инструкция JNE будет использовать флаги, установленные процедурой обработки прерывания, а не нашей инструкцией CMP. Для предотвращения подобных бедствий микропроцессор 8088 всегда сохраняет и восстанавливает регистр флагов при прерываниях. Прерывание сохраняет флаги, а инструкция IRET (сокр. от англ. "Interrupt Return") восстанавливает флаги в конце процедуры обработки прерывания.

То же самое выполняется и для инструкции INT. Поэтому после выполнения инструкции:

```
INT      21
```

стек микропроцессора 8088 будет выглядеть следующим образом:

```
Top of stack ->> Old IP (return address part I)
                  Old CS (return address part II)
                  Old Flag Register
```

(Стек увеличивается в сторону младших адресов памяти, поэтому вершина стека находится ниже "Old Flag Register" - старого регистра флагов).

Однако, когда мы помещаем инструкцию INT в программу, прерывание уже не будет сюрпризом. Зачем же тогда нам сохранять флаги? Ведь сохранение флагов полезно только, когда мы имеем внешнее прерывание, которое происходит в непредсказуемое время, не так ли? Как оказывается, нет. Имеется важная причина для сохранения и восстановления флагов для инструкции INT. Фактически без этой особенности программа Debug не смогла бы работать.

Debug использует специальный флаг из регистра флагов, называющийся флаг трассировки (англ. "Trap Flag"). Это флаг переводит микропроцессор 8088 в специальный режим, именуемый пошаговым, применяемый Debug для трассировки программ по одной инструкции за шаг. Когда установлен флаг трассировки, микропроцессор 8088 после выполнения каждой инструкции выдает прерывание "INT 1".

"INT 1" также сбрасывает флаг трассировки, и поэтому микропроцессор 8088 не будет находиться в пошаговом состоянии до тех пор, пока мы находимся

внутри процедуры "INT 1" в Debug. Но так как "INT 1" сохранила флаги в стеке, то инструкция IRET при возвращении к программе, которую мы отлаживали, восстановит флаг трассировки. Затем мы получим другое прерывание "INT 1" после следующей инструкции программы и т.д. Это является примером того, как бывает полезным сохранение регистра флагов в стеке. Но, как мы увидим в дальнейшем, эта возможность восстановления флагов не всегда целесообразна. Некоторые процедуры прерываний обходят восстановление регистра флагов. Например, процедура DOS "INT 21h" иногда изменяет регистр флагов чтобы произвести "короткое замыкание" нормального процесса возвращения. Многие функции процедуры "INT 21h", которые считывают или записывают информацию на диск, устанавливают флаг переноса, если произошла какая-либо ошибка (например, нет диска в дисковом).

Вектора прерываний

Откуда эти инструкции прерываний берут адреса процедур? Каждая инструкция прерывания имеет свой собственный номер, например такой, как 21h в "INT 21h". Микропроцессор 8088 находит адреса процедур прерываний в таблице векторов прерываний, размещаемой в самом начале памяти. Например, адрес процедуры для "INT 21h", состоящий из двух слов, равен 0000:0084. Этот адрес получается умножением номера прерывания на $4(4 \times 21h = 84h)$, так как для каждого вектора, или адреса процедуры необходимы четыре байта (два слова).

Эти вектора чрезвычайно полезны для добавления к DOS новых возможностей, потому что они позволяют перехватывать обращения к процедурам прерываний, изменяя адреса в таблице векторов. Однако в нашей книге мы этого делать не будем. Подобные трюки для нас пока еще слишком сложны.

Все эти идеи и методы станут яснее, когда мы рассмотрим больше примеров. Значительная часть книги, начиная отсюда, будет заполнена примерами, поэтому будет представлена прекрасная возможность поучиться. Если вы почувствовали, что уже завалены новой информацией, расслабьтесь. В следующей главе мы немного передохнем, произведем некоторую переориентацию и затем опять двинемся вперед.

Итог

Как мы и говорили, в этой главе содержалось большое количество информации. Мы не будем использовать её всю целиком, но нам было необходимо побольше узнать о сегментах. Глава 13 приведёт нас к модульному конструированию, и мы будем применять некоторые свойства сегментов, чтобы облегчить свой труд.

Мы начали эту главу с изучения того, как 8088 разбивает память на сегменты. Чтобы более детально разобраться с сегментами, мы создали .EXE-программу, состоящую из двух различных сегментов. Мы не будем использовать в этой книге .EXE-программы, но здесь они прекрасно продемонстрировали идею сегментов.

Мы также узнали, что находящаяся в начале программы рабочая область размером в 100h (256 байт) содержит копию того, что мы напечатали в командной строке. Мы также не будем использовать в книге эту информацию, но она помогает понять, для каких именно целей DOS выделил такой большой кусок памяти.

И, наконец, мы завершили эту главу изучением псевдооператоров SEGMENT, ENDS, ASSUME, NEAR и FAR. Все эти псевдооператоры помогают работать с сегментами. В книге мы не будем полностью использовать возможности этих псевдооператоров, так как .COM-программы используют только один сегмент. Но для программистов, пишущих на ассемблере огромные программы, эти псевдооператоры очень значительны. Если вы ими интересуетесь, то можете найти их детальное описание в руководстве по макроассемблеру.

В самом конце этой главы мы узнали о "корнях" полезной инструкции INT. Теперь мы готовы снова неспеша двинуться вперёд и узнать, как пишутся большие по размеру и более полезные ассемблерные программы.

Глава 12 . Коррекция курса

Мы познакомились со множеством новых и интересных вещей. Возможно, порой вы удивлялись тому,

что мы делали это бесцельно. На самом деле это совсем не так. Мы теперь достаточно освоились с новым окружением для того, чтобы установить определенное направление нашей деятельности и проложить курс до конца книги. Что мы и сделаем в этой главе: пристально рассмотрим конструкцию программы Dskpatch, а остальную часть книги посвятим непосредственно его созданию. Надеемся, что подобному правилу вы будете следовать при создании своих собственных программ.

Мы не будем сразу представлять законченную версию Dskpatch; мы сами написали её не таким путем. Наоборот, мы будем представлять небольшие тестовые программы, проверяющие каждый этап программы по мере того, как мы будем их создавать. Для того чтобы поступить именно так, нам необходимо точно знать, куда мы хотим идти, то есть здесь мы произведем коррекцию курса.

Так как Dskpatch будет работать с информацией на дисках, с неё мы и начнём.

Дискеты, сектора и Dskpatch

Информация на флоппи-дисках разделена на сектора, каждый из которых содержит 512 байт информации. Двусторонний диск, отформатированный с помощью DOS 2.0 и выше, состоит из 720 секторов, или $720 \times 512 = 368.640$ байт. Если мы сможем непосредственно просматривать эти сектора, то сможем исследовать как директорию, так и файлы на диске. Пока мы этого не можем, но Dskpatch сможет. Применим Debug для того, чтобы побольше узнать о секторах и получить представление о том, как мы будем выводить на экран сектор с помощью Dskpatch.

У Debug есть команда "L" (от англ. "Load"), считывающая содержимое секторов с диска в память, где впоследствии мы можем рассматривать их как данные. Просмотрим, например, директорию, которая начинается с сектора 5 двустороннего диска. Загрузите сектор 5 в память с диска в дисковом A (который у Debug имеет номер 0), используя команду "L" следующим образом:

```
-L 100 0 5 1
```

Как вы можете видеть из рисунка 12.1, команда загружает сектора в память, начиная с сектора 5 (мы загружаем всего один сектор), на смещении 100h от начала сегмента данных. Чтобы увидеть загруженный сектор, мы можем использовать команду дампирования:

```
-D 100
396F:0100 49 42 4D 42 49 4F 20 20 - 43 4F 4D 27 00 00 00 00 IBMBIO COM....
396F:0110 00 00 00 00 00 00 00 60 - 68 06 02 00 00 12 00 00 .....h.....
396F:0120 49 42 4D 44 4F 53 20 20 - 43 4F 4D 27 00 00 00 00 IBMDOS COM....
```

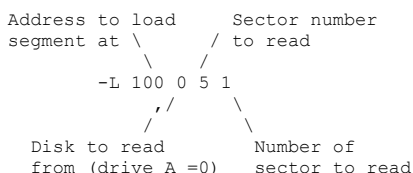


Рис. 12.1. Команда DEBUG "Load" - загрузка

```
396F:0130 00 00 00 00 00 00 00 60 - 68 06 07 00 00 43 00 00 .....h....C..
396F:0140 43 4F 4D 4D 41 4E 44 20 - 43 4F 4D 20 00 00 00 00 COMMAND COM....
396F:0150 00 00 00 00 00 00 00 60 - 68 06 18 00 00 45 00 00 ...h..B.....
396F:0160 41 53 53 45 4D 42 4C 45 - 52 20 20 08 00 00 00 00 ASSEMBLER ....
396F:0170 00 00 00 00 00 00 33 9C - B0 06 00 00 00 00 00 00 .....3.0.....

-D
396F:0180 46 57 20 20 20 20 20 20 - 43 4F 4D 20 00 00 00 00 FW.....COM.....
396F:0190 00 00 00 00 00 00 00 00 - 6F 05 2A 00 80 AF 00 00 .....o.../..
396F:01A0 46 57 20 20 20 20 20 20 - 4F 56 4C 20 00 00 00 00 FW.....OVL.....
396F:01B0 00 00 00 00 00 00 00 00 - 72 05 56 00 81 02 00 00 .....t.V.....
396F:01C0 46 57 20 20 20 20 20 20 - 53 57 50 20 00 00 00 00 FW...SWP .....
396F:01D0 00 00 00 00 00 00 9B 8A - FF 06 57 00 00 C8 00 00 .....W.H...
396F:01E0 43 4F 4E 46 49 47 20 20 - 44 41 54 20 00 00 00 00 CONFIG..DAT....
396F:01F0 00 00 00 00 00 00 ID 82 - A1 06 89 00 00 28 00 00 .....!....(..
```

Аналогичный формат будет применен и для Dskpatch, но с дополнительными улучшениями. Dskpatch будет практически полностью эквивалентен полноэкранному редактору дисковых секторов. Мы сможем выводить содержимое сектора на экране и перемещать курсор по экрану, произвольно изменяя символы или числа. Мы также сможем записать этот изменённый сектор обратно на диск, и именно поэтому мы назвали нашу программу Disk Patch ("Отладчик дисков"), иначе, Dskpatch (так как мы не можем использовать имя файла длиннее восьми символов).

Dskpatch будет служить мотивацией для всех создаваемых процедур. Однако в конце концов не в нём все дело. Применяя Dskpatch в качестве примера, мы также покажем много полезных процедур, которые окажутся весьма эффективными, если вы начнёте писать свои собственные программы. Это означает, что вы получите много процедур общего назначения для вывода на экран, работы с экраном, ввода с клавиатуры, и т.д.

Посмотрим внимательнее на улучшения, которыми будет обладать Dskpatch по сравнению с дампом сектора, выполняемым программой Debug. Дамп памяти, выполненный Debug, показывает только "высвечиваемые" символы - 96 из 256 различных символов, которые может выдать IBM PC. Почему так? Потому что MS-DOS-близкий родственник PC-DOSa - работает на многих различных компьютерах. Некоторые из этих компьютеров могут печатать только 96 символов, поэтому фирма Microsoft (автор Debug) предпочла написать одну версию Debug, которая могла бы работать на всех машинах.

Dskpatch предназначен для персональных компьютеров фирмы IBM и совместимых с ними, поэтому мы можем печатать все 256 различных символов. Чтобы это сделать, потребуется немного поработать. Используя функцию 2 DOS для вывода символа, мы можем высветить почти все символы, но некоторым из них DOS придает особое значение, например, символ номер 7 включает "звуковой сигнал", тем не менее в части III мы узнаем, как их выводить на экран.

Мы также будем постоянно использовать функциональные клавиши. Например для того, чтобы высветить следующий сектор, будет достаточно нажать клавишу "F2". Мы сможем изменить любой байт, подведя курсор к этому байту и введя новое число. Это будет похоже на использование текстового редактора, где мы можем легко вносить изменения. Большая часть этих деталей появится в процессе создания Dskpatch. (Рис. 12.2 показывает обычный вид экрана при работе Dskpatch - по сравнению с дампом Debug огромное количество улучшений.)

План игры

В главе 13 мы узнаем о том, как разбить программу на несколько различных исходных файлов. Затем, в главе 14, мы начнем серьёзную работу над Dskpatch. И в конце концов у нас будет всего девять исходных файлов для Dskpatch, которые необходимо

объединить вместе с помощью Link. Даже если вы не будете вводить и запускать их сразу, они всегда будут ждать вас в этой книге до тех пор, пока вы не захотите позаимствовать некоторые из приведённых здесь процедур общего назначения. В любом случае после того, как вы прочтете следующие главы, у вас будет хорошее представление о том, как создавать длинные программы.

Мы уже создали несколько полезных процедур, таких, как WRITE_HEX, для записи байта в виде двузначного шестнадцатеричного числа и WRITE_DECIMAL, для записи числа в десятичном виде. Теперь мы напишем несколько программ для вывода на экран блока памяти почти тем же способом, каким это делает команда Debug "D". Мы начнем с вывода на экран 16 байт памяти, одной строчки распечатки Debug, и затем будем работать над выводом 16 строк по 16 байт каждая (половина сектора).

```
Disk A      Sector
 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
-----
00|EB 21 90 49 42 4D 20 20 33 2E 31 00 02 02 01 00||
10|
20|
30|
40|
50|
60|
70|
80|
90|
A0|
B0|
C0|
D0|
E0|
F0|89 01 EB 55 90 01 06 1E 00 11 2E 20 00 C3 A1 18||
-----
```

Рис. 12.2. Вид экрана при выполнении программы Dskpatch

Весь сектор целиком не поместится на экране в формате, который мы выбрали, поэтому Dskpatch включает процедуры для скроллинга (прокрутки) изображения сектора, использующие прерывания ROM BIOS, а не DOS. Однако этим мы займемся лишь после того, как сможем получать полноэкранную распечатку половины сектора.

После того как мы научимся создавать дампы 256 байт памяти, мы напишем другую процедуру - для считывания сектора с диска в память. Мы будем дамповать половину сектора на экран и сможем затем применить Debug для изменения программы

для того, чтобы мы смогли дампить различные сектора. С этого момента у нас будет функциональное, но не очень привлекательное изображение, поэтому следующее, чем мы займёмся, - это сделаем его симпатичным.

Немного поработав и написав ещё несколько процедур, мы переделаем изображение половины сектора, улучшив его с эстетической точки зрения. Но это ещё не будет действительно полноэкранное изображение, так как его можно будет "прокручивать" только тем же способом, каким это делалось с дампом, полученным с помощью Debug. Но полноэкранное изображение появится дальше и, разрабатывая его, мы освоим применение процедур ROM BIOS, которые позволят управлять выводом изображения на экран, перемещать курсор, и т.д. Затем мы будем готовы к тому, чтобы узнать о том, как применять другие процедуры ROM BIOS. Следующими будут ввод с клавиатуры и процедуры команд, что позволит вступить с Dskpatch в двусторонний диалог. К тому времени нам понадобится ещё одна коррекция курса.

Итог

У нас теперь должно быть лучшее понимание того, куда мы направляемся, поэтому перейдём к следующей главе, в которой познакомимся с основами модульного конструирования и узнаем, как разделить программу на несколько различных исходных файлов. Затем, в главе 14, создадим несколько пробных процедур отображения на экране секций памяти.

Глава 13. Модульное конструирование

Без модульного конструирования написание Dskpatch не доставило бы большого удовольствия. Использование модульного конструирования значительно облегчает задачу написания любых, даже небольших программ. В этой главе мы изучим основные правила модульного конструирования и будем им следовать в дальнейшем. Давайте начнём с изучения того, как разделить большую программу на несколько исходных файлов.

___Раздельное ассемблирование___
Раздельное ассемблирование

В главе 10 мы добавили к VIDEO_IO.ASM короткую тестовую процедуру TEST_WRITE_DECIMAL. Возьмём её оттуда и поместим в собственный отдельный файл, называющийся TEST.ASM. Затем мы проассемблируем эти файлы по отдельности и с помощью Link соединим их вместе в одну программу. Вот файл TEST.ASM:

Листинг 13.1. Файл TEST.ASM.

```
CODE_SEG      SEGMENT      PUBLIC
    _ASSUME    CS:CODE_SEG
    ORG        100h
    EXTRN      WRITE_DECIMAL:NEAR

TEST_WRITE_DECIMAL    PROC      NEAR
    MOV        DX, 12345
    CALL       WRITE_DECIMAL
    INT        20h          ;Возврат в DOS
TEST_WRITE_DECIMAL    ENDP
CODE_SEG          ENDS
END               TEST_WRITE_DECIMAL
```

Большую часть этого файла мы видели ранее, но кое-что в нём для нас ново, поэтому давайте начнём работу с начала файла и постепенно будем продвигаться вниз. Во-первых, после SEGMENT появилось слово PUBLIC. Это слово сообщает ассемблеру о том, что мы хотим, чтобы этот сегмент (CODE_SEG) был скомбинирован с одним из сегментов, имеющим такое же имя, в данном случае, сегментом кода. Ассемблер передает эту информацию редактору связей (LINK), который соединяет разные файлы в один. Редактор связей выполняет работу по "сшиванию" различных частей одного сегмента вместе.

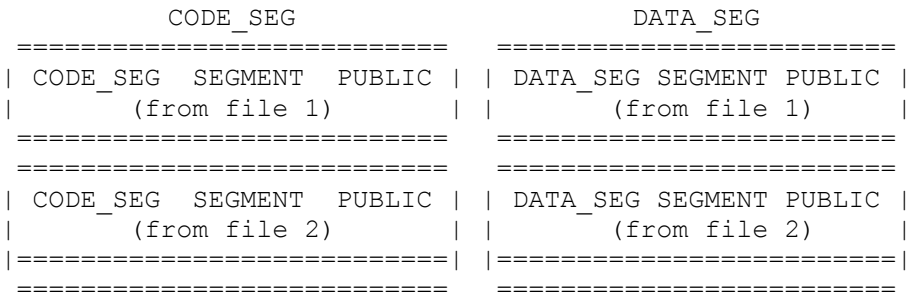


Рис. 13.1. LINK соединяет вместе сегменты из разных файлов

Наш файл теперь также содержит псевдооператор `EXTRN`. Выражение `"EXTRN WRITE_DECIMAL: NEAR"` сообщает ассемблеру две вещи: что `WRITE_DECIMAL` находится в другом, внешнем файле и что в этом файле процедура объявлена как `NEAR`. Поэтому она должна быть расположена в том же сегменте. Таким образом ассемблер генерирует "близкий" вызов (`"NEAR CALL"`) для этой процедуры. "Дальний" вызов (`"FAR CALL"`) был бы сгенерирован в случае, если бы мы поместили `FAR` после `WRITE_DECIMAL`.

```

      EXTRN WRITE_DECIMAL:NEAR
TEST_WRITE_DECIMAL    ^
                       |
        .              /|
      CALL WRI/TE_DECIMAL | LINK provides
                        /  the address
                      ???? <---

```

Рис. 13.2. LINK присваивает адреса для внешних переменных и процедур

Всё это касается изменений, необходимых для отдельных исходных файлов до того времени, пока мы не начнём сохранять данные в памяти. Тогда для них потребуется ещё один сегмент. Теперь внесём изменения в VIDEO_IO.ASM, и затем проведём ассемблирование и компоновку (с помощью LINK) этих двух файлов.

Удалите процедуру `TEST_WRITE_DECIMAL` из `VIDEO_IO.ASM`. Мы поместили её в `TEST.ASM`, поэтому в `Video_io` она нам не нужна. Затем удалите из `Video_io` выражение `"ORG 100h"`. Мы также перенесли его в `TEST.ASM`, который теперь содержит первую процедуру программы. Как мы знаем из главы 11, выражение `"ORG 100h"` необходимо для сохранения 256 байт рабочей области в начале программы, то есть перед `TEST_WRITE_DECIMAL` в исходном файле `TEST.ASM`.

Затем необходимо поместить слово PUBLIC после SEGMENT, как показано ниже:

```
CODE SEG      SEGMENT PUBLIC
```

Это делается для того, чтобы компоновщик знал о том, что он должен скомбинировать этот сегмент с таким же сегментом в TEST.ASM.

Наконец, измените "END TEST_WRITE_DECIMAL" в конце VIDEO_IO.ASM на просто "END". Это тоже

делается из-за того, что мы переместили главную процедуру в TEST.ASM. Процедуры в VIDEO_IO.ASM теперь внешние ("external") процедуры, то есть у них нет своих собственных функций и они должны быть с помощью LINK присоединены к процедурам, которые вызывают их из других файлов. Мы больше не нуждаемся в наличии имени после псевдооператора END в VIDEO_IO.ASM, так как главная программа находится теперь в TEST.ASM.

После внесения этих изменений, исходный файл должен выглядеть следующим образом:

```
CODE_SEG      SEGMENT      PUBLIC
    ASSUME     CS:CODE_SEG

    PUBLIC     WRITE_HEX_DIGIT
    .
    .
    .
WRITE_HEX_DIGIT      ENDP

    PUBLIC     WRITE_HEX
    .
    .
    .
WRITE_HEX      ENDP

    PUBLIC     WRITE_CHAR
    .
    .
    .
WRITE_CHAR      ENDP

    PUBLIC     WRITE_DECIMAL
    .
    .
    .
WRITE_DECIMAL      ENDP

CODE_SEG      ENDS
    END
```

с псевдооператором ASSUME в самом начале.

Ассемблируйте эти два файла так же, как вы раньше ассемблировали Video_io. TEST.ASM знает всё, что нужно о VIDEO_IO.ASM из содержащегося в нём

выражения EXTRN. Остальное произойдет, когда мы проведём компоновку этих файлов с помощью LINK.

У вас теперь должны быть файлы TEST.OBJ и VIDEO.OBJ. Используйте следующую команду, чтобы провести компоновку этих программ и соединить их в один файл:

```
A:\>LINK TEST VIDEO_IO;
```

LINK связывает процедуры двух файлов вместе для того, чтобы создать один файл, содержащий всю программу целиком. Он использует имя первого файла, которое мы ввели, в качестве названия результирующего .EXE-файла, так что теперь у нас есть TEST.EXE.

Наконец, создайте .COM-файл, так же, как это делалось раньше, введя "EXE2BIN TEST TEST.COM". Итак, мы создали нашу программу из двух отдельных файлов. Окончательный вариант .COM-файл идентичен тому, что мы создали из одного файла VIDEO_IO.ASM, когда он содержал главную процедуру TEST_WRITE_DECIMAL.

Мы теперь часто будем использовать несколько отдельных исходных файлов, и их значение станет яснее по мере того, как процедуры станут расти. В следующей главе мы напишем пробную программу для дампирования секций памяти в шестнадцатеричном виде. Мы обычно будем писать простую пробную версию процедуры перед тем, как написать законченную версию. Делая так, мы сможем увидеть, как написать добротную окончательную версию, а также сэкономим усилия и избежим умственных перегрузок в процессе работы.

Существует ещё несколько методов экономии усилий. Мы назовем их три закона модульного конструирования.

Три Закона Модульного Конструирования

Эти три закона на самом деле они не являются законами, это всего лишь рекомендации, но мы будем использовать их в этой книге. Если хотите, определите свои собственные законы, но в любом случае, всё время придерживайтесь одного и того же. Ваша работа значительно упростится, если вы будете последовательными.

Три Закона Модульного Конструирования

1. Сохраняйте и восстанавливайте всегда все регистры (их значения), кроме тех случаев, когда процедура передаёт значение в регистре.

2. Будьте последовательны в применении для передачи информации одних и тех же регистров.
Например:

- " DL,DX - для передачи байта или слова.
- " AL,AX - для возвращения байта или слова.
- " BX:AX - для возвращения двойного слова
- " DS:DX - передача и возвращение адресов
- " CX - счетчик повторения и другие счетчики
- " CF - устанавливается при ошибке; код ошибки возвращается в одном из регистров, например в AL или в AX.

3. Определяйте все внешние взаимодействия процедуры в заголовке-комментарии:

- " Информация, необходимая на входе.
- " Возвращаемая информация (изменяемые регистры).
- " Вызываемые процедуры.
- "Используемые переменные (считывания, записи и т.д.).

Существует чётко просматриваемая связь между модульным конструированием в программировании и модульным конструированием в инженерном искусстве. Инженер-электрик, например, может создать очень сложный прибор из блоков, выполняющих различные функции, не зная точно, как именно работает каждый блок в отдельности. Но если каждый блок применяет свое напряжение питания или оригинальную, нестыкующуюся с другими блоками форму контактов, то это отсутствие единообразия составляет главную причину головной боли бедняги-инженера, который должен умудриться как-то подавать разное напряжение к каждому блоку и создавать специальные виды соединений между блоками. Не очень-то приятное развлечение, но к счастью для инженера, существуют стандарты, допускающие лишь небольшое число различных напряжений. Таким образом получается, что надо обеспечить подачу, скажем, всего четырех разных

напряжений вместо того, чтобы к каждому блоку подавать своё определенное напряжение.

Модульное конструирование и стандартные интерфейсы в ассемблере не менее важны, и именно поэтому мы приняли эти законы (скажем так) и будем использовать эти законы в дальнейшем. Как вы увидите к концу этой книги, эти правила значительно упростят нашу задачу. Давайте рассмотрим эти законы детальнее.

Сохраняйте и восстанавливайте всегда все регистры (их значения), кроме тех случаев, когда процедура передаёт значение в регистре. У микропроцессора 8088 не так много регистров. Сохраняя их значения в начале процедуры, мы освобождаем их для использования внутри самой процедуры. Но при этом необходимо быть весьма осторожным. Вы увидите, что мы будем это делать во всех процедурах, с помощью инструкций PUSH в начале процедуры и инструкций POP в конце.

Единственное исключение составляют процедуры, которые должны вернуть некоторую информацию процедурам, их вызывающим. Например, процедура, считывающая символ с клавиатуры, должна как-то вернуть этот символ. Мы не будем сохранять те регистры, которые собираемся использовать для возвращения информации.

Короткие процедуры также помогают в решении проблемы сокращения числа используемых регистров. Иногда мы будем писать процедуры, которые используют только один регистр. Это правило не только помогает сократить число используемых регистров, но также делает программу более лёгкой для написания и часто облегчает её чтение. Мы ближе познакомимся с этим правилом, когда начнём писать процедуры для Dskpatch.

Будьте последовательными в использовании для передачи информации одних и тех же регистров. Наша работа станет проще, если мы установим стандарты для обмена информацией между процедурами. Будем применять один регистр для передачи, а другой для получения информации. Нам также необходимо посылать адреса больших массивов информации, и для этого мы будем использовать регистровую пару DS:DX, так что данные могут быть расположены в любом месте памяти. Об этом вы узнаете больше, когда мы

представим новый сегмент для данных и начнём использовать регистр DS.

Мы резервируем регистр CX для счётчика повторений. Мы скоро напишем процедуру для записи одного символа несколько раз, так что мы сможем записать десять пробелов, вызвав эту процедуру (WRITE_CHAR_N_TIMES) с CX, установленным в 10. Мы будем использовать регистр CX всегда, когда нам понадобится счётчик повторений или когда мы захотим вернуть результат какого-либо подсчёта, например числа символов, считанных с клавиатуры (мы сделаем это, когда напишем процедуру, называющуюся READ_STRING).

Наконец, мы будем устанавливать флаг переноса (CF) всякий раз, когда происходит ошибка. Не все процедуры используют флаг переноса. Например, WRITE_CHAR срабатывает всегда, поэтому нет причины для возвращения сообщения об ошибке. Но процедура записи на диск может столкнуться со многими ошибками (нет диска, диск защищён от записи, и т.д.). В этом случае мы будем использовать регистр для возвращения кода ошибки. Стандартов здесь не существует, так как DOS использует разные регистры для разных функций. Это просчёт, но не наш.

Определяйте все внешние взаимодействия процедуры в заголовке-комментарии. Нет необходимости знать, как работает процедура, если всё, что мы хотим с ней делать - это использовать ее, и поэтому мы будем перед каждой процедурой помещать детальный комментарий-заголовок. Этот заголовок содержит всю информацию, которую нам необходимо знать. Он говорит нам о том, что именно необходимо помещать в каждый регистр перед вызовом процедуры, а также говорит, какую информацию процедура возвращает, большая часть процедур использует регистры для хранения своих переменных, но некоторые из процедур, которые мы скоро увидим, используют переменные, хранящиеся в памяти.

Комментарий-заголовок должен сказать, какие из этих переменных только считываются, а какие изменяются. И наконец, каждый заголовок должен содержать список вызываемых процедур. Ниже приведён пример такого заголовка.

```
;
; Это пример полного заголовка. В этом месте обычно
; приводится полное описание того, что данная
; процедура делает. Например: Процедура записывает
; сообщение "Sector" в первую строку.
; DS:DX Адрес сообщения "Sector"
```

```
; Calls:   GOTO_XY, WRITE_STRING  (вызываемая процедура)
; Reads:   STATUS_LINE_NO         (переменные памяти
;                                       считываемые)
; Writes:  DUMMY                  (переменные памяти
;                                       чтения и записи)
```

Когда бы мы не захотели впоследствии использовать любую из написанных процедур, нам достаточно просто взглянуть на заголовок, чтобы узнать, как её использовать. Не будет нужды вникать в тонкости работы процедуры, чтобы разобраться в том, что именно она делает.

Эти законы упрощают программирование на ассемблере, и мы конечно будем твёрдо их придерживаться, но необязательно с первой попытки написания процедуры - в этом просто нет необходимости. Первая версия процедуры или программы лишь пробный вариант. Часто мы не знаем точно, как написать программу, которую мы держим в голове, поэтому в черновом варианте мы будем писать программу, не придерживаясь законов модульного конструирования. Мы только сделаем примерный набросок будущей программы и убедимся, что она работает. Затем мы можем вернуться назад и переписать каждую процедуру, чтобы она отвечала этим законам.

Программирование - это процесс, который протекает очень быстро. В этой книге мы отразили большую часть трудностей, с которыми можно столкнуться при написании Dskpatch, но, конечно, мы не могли показать их всех. Просто не хватит места, чтобы напечатать все версии, которые мы написали, прежде чем пришли к окончательной версии. Наши первые попытки имеют очень мало сходства с окончательными версиями, которые вы увидите, поэтому когда вы пишете программы, не старайтесь всё написать правильно с первого раза. Будьте готовы к тому, чтобы переписать каждую процедуру, после того как вы больше узнаете о том, что вам нужно.

В следующей главе мы напишем простую пробную программу для печати блока памяти. Это не будет окончательная версия; мы пройдем ещё через несколько вариантов, прежде чем будем полностью удовлетворены результатами своих трудов, и даже потом, после написания окончательного варианта, будут ещё изменения, которые мы захотим внести. Мораль:

программа никогда не бывает закончена... но желательно где-нибудь остановиться.

Итог

Эта была глава, которую вам надо запомнить и использовать в будущем. Мы начали с изучения того, как разделить программу на несколько различных исходных файлов, которые мы можем ассемблировать независимо друг от друга, а затем соединить вместе с помощью LINK. Мы использовали псевдооператоры PUBLIC и EXTRN, чтобы информировать редактор связей о том, что надо соединить два исходных файла. PUBLIC говорит, что процедура, перед именем которой он поставлен, может вызываться из других исходных файлов, в то время как EXTRN сообщает ассемблеру, что процедура, которую мы хотим использовать, находится в другом файле.

Мы также использовали PUBLIC после определения SEGMENT, чтобы редактор связей соединил вместе сегменты, имеющие одинаковое имя, но находящиеся в разных исходных файлах.

Затем мы перешли к изучению трёх законов модульного конструирования. Эти правила призваны облегчить работу по программированию, в чем вы убедитесь после того, как увидите их в действии. Вы также найдете, что программы, отвечающие требованиям этих трёх законов, легче писать, отлаживать и читать.

Глава 14. Дампирование памяти

Начиная с этой главы мы сконцентрируемся на создании Dskpatch. Некоторые из инструкций в процедурах могут быть для вас неизвестными. Мы кратко расскажем о каждой из них, как только встретим её на своем пути, но для детальной информации вам нужна дополнительная книга, содержащая их детальное описание.

Вместо того, чтобы описывать инструкции микропроцессора 8088, мы сосредоточимся на новых понятиях, например таких, как режим адресации памяти, о которых мы расскажем в этой главе. В Части III мы

ещё больше удалимся от деталей инструкций и начнём изучать информацию, специфическую для IBM Personal Computer и совместимых с ним.

Сейчас мы займёмся режимами адресации, написав небольшую пробную программу дампирования 16 байт памяти в шестнадцатеричном представлении. Для начала давайте узнаем, как использовать память в качестве переменной.

Режимы адресации

Мы видели два режима адресации, известных как регистровый и непосредственный. Первый режим адресации, которому мы научились, был регистровый режим, который использует регистры как переменные. Например, инструкция:

```
MOV     AX, BX
```

использует регистры AX и BX как переменные.

Затем мы перешли к непосредственному режиму адресации, в котором загружали число прямо в регистр, например:

```
MOV     AX, 2
```

В этом режиме байт или слово памяти загружается непосредственно в записанный после инструкции регистр. В этом смысле инструкция MOV в нашем примере имеет в длину один байт с двумя байтами для данных:

```
396F:0100 B80200 MOV AX,0002
```

Код инструкции B8h, а два байта данных (02h и 00h) следуют за ней (помните, что 8088 хранит младший байт первым в памяти).

Теперь мы узнаем, как использовать память в качестве переменной. Непосредственный режим позволяет нам считывать фиксированный кусочек памяти, непосредственно следующий за инструкцией, но не позволяет нам изменять память. Для этого нам требуются другие режимы адресации.

Давайте начнём с примера. Приведённая ниже программа считывает 16 байт памяти, по одному за шаг, и высвечивает каждый байт в шестнадцатеричном

представлении, вставляя пробел между каждым из 16 шестнадцатеричных чисел. Введите программу в файл DISP_SEC.ASM и ассемблируйте её. Позже нам понадобится немного изменить VIDEO_IO.ASM, но сначала давайте позаботимся о DISP_SEC.ASM:

Листинг 14.1. Новый файл DISP_SEC.ASM.

```
CGROUP      GROUP      CODE_SEG, DATA_SEG ;Объединяет два
                                           ;сегмента вместе
      ASSUME      CS:CGROUP, DS:CGROUP

CODE_SEG      SEGMENT PUBLIC
      ORG      100h
      EXTRN      WRITE_HEX:EAR
      EXTRN      WRITE_CHAR:NEAR
;
;Пример программы, дампирующей 16 байт памяти в одну строку
;шестнадцатеричных чисел
;
DISP_LINE      PROCNEAR
      XOR      BX,BX      ;BX устанавливается в 0
      MOV      CX,16      ;Дампируются 16 байт
HEX_LOOP:
      MOV      DIRECTOR [BX]      ;Получить 1 байт
      CALL     WRITE_HEX      ;Этот байт переводится в
                               ;шестнадцатеричное число
      MOV      DL,' '      ;Вводится пробел между числами
      CALL     WRITE_CHAR
      INC      BX
      LOOP     HEX_LOOP
      INT      20h      ;Возврат в DOS
DISP_LINE      ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
      PUBLIC    SECTOR
SECTOP      DB      10h,11h,12h,13h,14h,15h,16h,17h ;Образец теста
              DB      18h,19h,1Ah,1Bh,1Ch,1Dh,1Eh,1Fh
DATA_SEG      ENDS

      END      DISP_LINE
```

Обратите внимание на то, что мы поместили сегмент данных (DATA_SEG) после сегмента кода (CODE_SEG). Мы поместили его в конце файла для

того, чтобы компоновщик загрузил данные в конце программы.

Мы также добавили к этой программе несколько новых трюков. Поэтому придётся внести в VIDEO_IO.ASM небольшие изменения. Во-первых, удалите оператор ASSUME из Video_io и поместите следующие две строки в начало VIDEO_IO.ASM:

```
CGROUP    GROUP    CODE_SEG          ;Объединяет вместе два сегмента
          ASSUME    CS:CGROUP
```

Отныне мы будем помещать эти строки в начало каждого файла, с небольшими вариациями. Мы будем писать:

```
CGROUP    GROUP    CODE_SEG, DATA_SEG          ;Объединяет вместе
                                                    ;два сегмента ASSUME
          CS:CGROUP, DS:CGROUP
```

(с DATA_SEG) тогда, когда нам понадобится иметь в одном файле и сегмент кода, и сегмент данных.

ASSUME заменил здесь старый оператор ASSUME. Позднее мы увидим, что именно делают эти два выражения. Но сейчас попробуем новую программу в действии. Ассемблируйте Disp_sec и Video_io.

Теперь мы готовы провести компоновку DISP_SEC.OBJ и Video_io.OBJ и пропустить результат через Exe2bin, но сначала применим LINK для того, чтобы создать .EXE-файл, называющийся DISP_SEC.EXE. Первым в командной строке LINK должно быть имя файла, содержащего основную процедуру (Disp_sec в этом случае), а в конце списка файлов должна быть поставлена точка с запятой, поэтому напечатайте:

```
A:\>LINK DISP_SEC Video_io;
```

Компоновка будет проходить всегда одинаково, лишь количество имен файлов перед точкой с запятой будет увеличиваться по мере того, как у нас будет больше файлов, но основная процедура должна быть всегда в первом файле из списка.

Теперь переведите .EXE-файл в .COM, напечатав:

```
A:\>EXE2BIN DISP_SEC DISP_SEC.COM
```

В общем случае два предыдущих шага для файлов file1, file 2 и так далее, будут выглядеть следующим образом:

```
LINK file1 file2 file3...; EXE2BIN file1 file1.COM
```

Теперь запустите .COM-файл. Убедитесь, что вы запустили Eхе2bin ПЕРЕД DISP_SEC. В противном случае произойдет запуск .EXE версии Disp_sec, и никто не сможет предсказать, что случится. В худшем случае вам придется выключить компьютер, подождать с минуту, и затем снова включить его.

Если после запуска программы вы не увидите:

```
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
```

то вернитесь назад и найдите ошибку.

Посмотрим, как работает Disp_sec. Инструкция:

```
MOV     DL, SECTOR[BX]           ;Получить 1 байт
```

использует новый режим адресации, называющийся косвенным - адресация памяти с помощью регистра базы и смещения, или более коротко, по базе. Посмотрим, что же это означает.

```
MOV     DL, SE\  CTOR[BX] ^
          \      |
          |      | 0434:0013
          |      | 0434:0012
          v      | 0434:0011
          SECTOR: 0434:0010
```

Рис. 14.1. Результат трансляции SECTOR[BX]

Взглянув на DISP_SEC, вы увидите, что метка SECTOR находится в сегменте, называемом DATA_SEG. Это новый сегмент, использующийся для хранения переменных в памяти. Всегда, когда мы хотим записать или считать данные из памяти, мы выделяем место в этом сегменте. Вернемся к переменным в памяти через минуту, но сначала узнаем еще кое-что о сегментах.

"ASSUME DS:CGROUP" сообщает ассемблеру о том, где именно в памяти искать переменные. Вы, возможно, предполагали, что мы напишем "ASSUME DS:DATA_SEG". Не совсем, так как если мы хотим создать .COM-файл, то можем использовать только

один сегмент. Однако работать удобно с двумя: одним для кода, одним - для данных. Вот здесь на сцену и выходит псевдооператор GROUP. Он группирует различные сегменты в один, названный именем, которое мы указали перед псевдооператором GROUP. Таким образом, выражение:

```
CGROUP    GROUP    CODE_SEG, DATA_SEG
```

соединит сегменты CODE_SEG и DATA_SEG в один 64-килобайтный сегмент, называющийся CGROUP. Процессы, происходящие внутри группы, немного сложнее, но нам не обязательно знать детали. Если вы всё же интересуетесь этими деталями, то можете найти их в руководстве по макроассемблеру. Однако хотим предупредить: это сложный материал.

Настало время вернуться к режиму адресации по базе. Две строки:

```
SECTOR DB    10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h,      ;Образец теста
        DB    18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh
```

устанавливают 16 байт памяти в сегменте данных, начиная с SECTOR, который ассемблер переводит в адрес. DB, как вы может быть помните, означает "Define Byte" ("определить байт") числа. Числа, написанные после каждого DB, являются иницилирующими значениями. Таким образом, когда мы запустим DISP_SEC.COM память, начиная с SECTOR, будет содержать 10h, 11h, 12h и т.д. Если мы напомним:

```
MOV        DL, SECTOR
```

инструкция будет пересылать первый байт (10h) в регистр DL. Такой метод адресации называется прямым. Но мы так не напишем. Наоборот, мы поместим [BX] после SECTOR. Это выглядит подозрительно похоже на индекс в массиве, как в выражении Бейсика:

$K = L(10)$

которое пересылает 10-й элемент L в K.

Фактически, инструкция MOV делает то же самое. Регистр BX содержит смещение в памяти относительно SECTOR. Поэтому, если BX равен 0, "MOV DL,SECTOR[BX]" перешлёт первый байт (в данном случае 10h)

в DL. Если BX равен 0Ah, эта инструкция MOV пересылает одиннадцатый байт (1Ah, помните, что мы начали с 0) в DL.

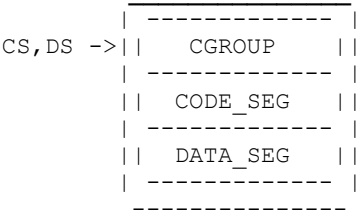


Рис. 14.2. Группа сегментов рассматривается как единый сегмент

С другой стороны, инструкция "MOV DX,SECTOR[BX]" должна пересылать в DX шестое слово, так как смещение в 10 байт это то же самое, что и смещение в 5 слов, и первое слово расположено на нулевом смещении. (Для энтузиастов: последняя инструкция MOV нелегальна, так как SECTOR - это байтовая метка, в то время как DX - это регистр длиной в слово. Чтобы сообщить ассемблеру о том, что мы хотим использовать SECTOR как метку слова, нам надо было бы записать эту инструкцию следующим образом: "MOV DX,Word Ptr SECTOR[BX]").)

Существует много других режимов адресации, с некоторыми из которых мы столкнемся позже, но большая их часть нам не понадобится. Все режимы адресации суммированы в табл. 14.1.

Таблица 14.1. Режимы адресации

Режим адресации, сегментные регистры	Формат адресов	Использу- емые регистры
Регистровый	Регистр (напр. AX)	Нет
Непосредственный	Данные (напр. 12345)	Нет
Режимы адресации памяти		
Косвенный регистровый	[BX]	DS
	[BP]	SS
	[DI]	DS

__Язык ассемблера__

	[SI]	DS
По базе*	Метка[BX]	DS
	Метка[BP]	SS
Прямой с индексированием*	Метка[DI]	DS
	Метка[SI]	DS
По базе с индексированием*	Метка[BX + SI]	DS
	Метка[BX + DI]	DS
	Метка[BP + SI]	SS
	Метка[BP + DI]	SS
Строковые команды:		Считывают из DS:DI
(MOVSW, LODSB и т.д.)		Пишут в ES:DI
* - Метка[...] может быть заменена на [смещение + ...], то есть мы можем писать [10 + BX], и адрес будет 10 + BX.		

Добавление в дампы символов

Мы почти закончили процедуру вывода дампа на экран, похожего на дампы, полученные с помощью Debug. До этого мы дампируем только одну строку шестнадцатеричных чисел, следующим шагом будет добавление символов, высвечивающихся за ними. Это не особо сложно, поэтому без дальнейшего промедления мы приводим ниже новую версию DISP_LINE (в DISP_SEC.ASM) со вторым циклом, добавленным для распечатки символов:

Листинг 14.2. Изменения к процедуре DISP_LINE в файле DISP_SEC.ASM.

```
DISP_LINE    PROC    NEAR
    XOR      BX,BX          ;Устанавливает BX в 0
    MOV      CX,16          ;Дампируются 16 байт
HEX_LOOP:
    MOV      DL,SECTOR[BX]  ;Выбрать 1 байт
    CALL     WRITE_HEX      ;Преобразовать его в
                             ;шестнадцатеричное число
    MOV      DL,' '         ;Записать пробел между числами
    CALL     WRITE_CHAR
    INC      BX
    LOOP     HEX_LOOP
    MOV      DL,' '         ;Добавить пробел между числами
```

```
CALL    WRITE_CHAR
MOV     CX,16
XOR     BX,BX           ;Установить BX снова в 0
ASCII_LOOP:
MOV     DL,SECTOR[BX]
CALL    WRITE_CHAR
INC     BX
LOOP    ASCII_LOOP
INT     20h             ;Возврат в DOS
DISP_LINE ENDP
```

Ассемблируйте программу, с помощью LINK присоедините к Video_io, пропустите через Exe2bin и опробуйте в действии. На экране появится именно то, что мы хотели (рис. 14.3)

```
A:\>disp_sec
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
A:\>
```

Рис. 14.3. Изображение - результат действия DISP_LINE

Измените данные так, чтобы они включали 0Dh или 0Ah. Вы увидите еще более странный результат, и вот почему: 0Ah и 0Dh - символы, означающие "перевод строки" и "возврат каретки". DOS интерпретирует их как команды движения курсора, но для нас нужно, чтобы в данной части изображения они рассматривались как обычные символы. Чтобы сделать это, надо изменить WRITE_CHAR так, чтобы она печатала все символы, не придавая некоторым из них какого-либо специального назначения. Мы сделаем это в части III, но сейчас давайте немного изменим WRITE_CHAR, чтобы она печатала пробел вместо младших (между 0 и 1Fh) символов.

```
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F.....
A:\>
```

Рис. 14.4. Модифицированная версия DISP_LINE

Замените WRITE_CHAR в VIDEO_IO.ASM на эту новую процедуру:

Листинг 14.3. Новая процедура WRITE_CHAR в файле VIDEO_IO.ASM.

```
PUBLIC    WRITE_CHAR
;
;Эта процедура печатает символ на экране, применяя функцию
```

```
;вызова DOS.
; WRITE.CHAR заменяет символы с 0 по 1Fh точками
;
;           DL           байт, выводимый на экран
;
WRITE_CHAR    PROC        NEAR
    PUSH     AX
    PUSH     DX
    CMP      DL,32        ;Перед символом есть пробел?
    JAE      IS_PRINTABLE ;Нет, напечатать символ
    MOV      DL,' '       ;Да, заменить точкой
IS_PRINTABLE:
    MOV      AH,2         ;Вызов вывода символа
    INT      21h          ;Вывод символа, хранящегося в DL
    POP      DX           ;Восстанавливаются значения в DX и AX
    POP      AX
RET WRITE_CHAR            ENDP
```

Испытайте эту процедуру в действии с помощью DISP_SEC, и попробуйте изменять данные так, чтобы проверить граничные условия.

Дампирование 256 байт памяти

Мы научились дампировать одну строку (или 16 байт) памяти. Следующий шаг - это дампирование 256 байт. Как оказывается, это ровно половина числа байт в секторе, поэтому мы будем работать над созданием изображения половины сектора. Мы ещё внесём достаточно улучшений, это только пробная версия.

Нам здесь понадобятся две новые процедуры и модифицированная версия DISP_LINE. Новые процедуры это - DISP_HALF_SECTOR, которая в дальнейшем развернётся в законченную процедуру изображения на экране половины сектора, и SEND-CRLF, которая посылает курсор в начало следующей строки ("CRLF" означает "Carriage Return-Line Feed" - "Возврат каретки - Перевод строки"-пара символов, которые переводят курсор на следующую строку).

SEND_CRLF очень проста, поэтому давайте начнём с неё. Поместите следующую процедуру в файл, называющийся CURSOR.ASM:

Листинг 14.4. Новый файл CURSOR.ASM.

```
CR      EQU      13      ;Возврат каретки
LF      EQU      10      ;Перевод строки

CROUP   GROUP     CODE_SEG
        ASSUME     CS:CGROUP

CODE_SEG SEGMENT      PUBLIC

        PUBLIC     SEND-CRLF
;
;Эта процедура только посылает пару символов Возврат каретки -
; Перевод строки,
;применяя процедуру Dos, поэтому прокрутка будет выполнена
;правильно.
;
SEND_CRLF PROC NEAR
        PUSH      AX
        PUSH      DX
        MOV       AH, 2
        MOV       DL, CR
        INT       21h
        MOV       DL, LF
        INT       21h
        POP       DX
        POP       AX
RET
SEND_CRLF ENDP
CODE_SEG ENDS

        END
```

Эта процедура посылает пару символов CR и LF, используя функцию DOS номер 2 для выдачи символов. Выражение:

```
CR      EQU      13      ;Возврат каретки
```

использует псевдооператор EQU для того, чтобы установить, что имя CR идентично 13. Таким образом, инструкция "MOV DL,CR" эквивалентна "MOV DL,13". Как показано на рис. 14.5, ассемблер подставляет 13 везде, где видит CR. Аналогично, он подставляет 10 везде, где видит LF.

```
CR      EQ    13
        .
        .
        .      / 13
MOV     DL,C/  R
        /
```

Рис. 14.5. Псевдооператор EQU позволяет использовать имена вместо чисел

Файл Disp_sec теперь требует серьезной работы. Вот новая версия Disp_sec.ASM. Начиная отсюда, добавления к нашим программам будут печататься на сером фоне, а текст, который вам нужно уничтожить, будет печататься синим цветом:

Листинг 14.5. Новая версия Disp_sec.ASM.

```
CROUP    GROUP    CODE_SEG, DATA_SEG    ;Group two segments
                                                ;together

    ASSUME    CS:CGROUP

CODE_SEG    SEGMENT PUBLIC
    ORG      100h

    PUBLIC    DISP_HALF_SECTOR
    EXTRN     SEND_CRLF:NEAR
;
;Эта процедура отображает половину сектора (256 байт)
;
Uses:      DISP_LINE, SEND_CRLF

DISP_HALF_SECTOR    PROC    NEAR
    XOR      DX,DX    ;Старт в начале SECTOR'a
    MOV      CX,IL    ;Отображает 16 строк
HALF_SECTOR:
    CALL     DISP_LINE
    CALL     SEND_CRLF
    ADD      DX,16
    LOOP     HALF_SECTOR
    INT      20h
DISP_HALF_SECTOR    ENDP
;
;    PUBLIC    DISP_LINE
;    EXTRN     WRITE_HEX:NEAR
;    EXTRN     WRITE_CHAR:NEAR
```

Дампирование 256 байт памяти

```
;Процедура отображает строку данных, или 16 байт сначала в
;шестнадцатеричном виде, затем в ASCII.
;
;      DS:DX      Смещение в секторе, в байтах
;      Используется:  WRITE_CHAR,WRITE_HEX
;      Читается:  SECTOR

DISP_LINE      PROC      NEAR
    XOR        BX,BX
    PUSH       BX
    PUSH       CX
    PUSH       DX
    MOV        BX,DX      ;Смещение более полезно в BX
    MOV        CX,16      ;Дампируются 16 байт
    PUSH       BX          ;Сохраняется смещение для ASCII_LOOP
HEX_LOOP:
    MOV        DL,SECTOR[BX] ;Выбирается 1байт
    CALL       WRITE_HEX     ;Переводится в шестнадцатеричную
                           ;форму
    MOV        DL,' '      ;Вводится пробел между числами
    INC        BX
    LOOP       HEX_LOOP
    MOV        DL,' '      ;Вводится пробел перед символом
    CALL       WRITE_CHAR
    MOV        CX,16
    POP        BX          ;Возврат смещения в SECTOR
    XOR        BX,BX ASCII_LOOP:
    MOV        DL,SECTOR[BX]
    CALL       WRITE_CHAR
    INC        BX
    LOOP       ASCII_LOOP
    POP        DX
    POP        CX
    POP        BX RET
    INT        20h
DISP_LINE      ENDP

CODE_SEG      ENDS
DATA_SEG      SEGMENT PUBLIC
    PUBLIC     SECTOR
SECTOR        DB 10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h      ;Образец теста
               DB 18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh
               DB 16DUP(11h)
```



```
DB 16 DUP(12h)
DB 16 DUP(13h)
DB 16 DUP(14h)
DB 16 DUP(15h)
DB 16 DUP(16h)
DB 16 DUP(17h)
DB 16 DUP(18h)
DB 16 DUP(19h)
DB 16 DUP(1Ah)
DB 16 DUP(1Bh)
DB 16 DUP(1Ch)
DB 16 DUP(1Dh)
DB 16 DUP(1Eh)
DB 16 DUP(1Fh)
DATA_SEG ENDS
END DISP_HALT_SECTOR
```

Все вносимые изменения довольно просты для понимания. В DISP_LINE мы добавили "PUSH BX" и "POP BX" вокруг HEX_LOOP, так как мы хотим использовать начальное значение в ASCII_LOOP. Мы также добавили инструкции PUSH и POP для сохранения и восстановления всех регистров, которые мы используем в DISP_LINE. Фактически DISP_LINE почти готов; изменения, которые нам осталось внести, - чисто эстетические - следует добавить пробелы и графические символы, чтобы изображение выглядело более привлекательным; их мы внесём позже.

Когда вы будете проводить компоновку, помните, что теперь создано три файла: DISP_SEC, Video_io и Cursor. В этом списке DISP_SEC должен быть первым. После того как с помощью Eхе2bin вы получите .COM-версию, то увидите изображение, идентичное приведённому на рис. 14.6.

```
A:\>disp_sec
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10.....
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11.....
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12.....
13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13.....
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14.....
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15.....
16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16.....
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17.....
18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18.....
19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19.....
```

```
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A .....  
1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B .....  
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C .....  
1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D .....  
1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E .....  
1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F .....  
A:\>
```

Рис. 14.6. Изображение на экране в результате работы Disp_sec

Итак, файлов у нас прибавилось, но теперь давайте перейдём к следующей главе, где мы считаем сектор прямо с диска перед тем, как распечатать половину сектора.

Итог

Мы узнали о различных режимах адресации памяти и регистров в микропроцессоре 8088. Мы заодно обучились косвенной адресации, которую прежде всего использовали для считывания 16 байт памяти.

Мы также использовали косвенную адресацию в некоторых программах, которые написали в этой главе, начиная с программы печати 16 шестнадцатеричных чисел на экране. Эти 16 чисел были взяты из области памяти, помеченной SECTOR, которую мы позже расширили для того, чтобы иметь возможность напечатать дамп памяти в 256 байт - половину сектора.

И наконец, мы увидели дампы в том виде, в каком они появляются на экране, а не вводятся заранее. Мы будем использовать эти экранные дампы в следующих главах.

Глава 15. Дампирование сектора диска

Теперь, когда у нас есть программа, дампирующая 256 байт памяти, мы можем добавить несколько процедур считывания сектора с диска и помещения его в память начиная с SECTOR. Затем наша процедура произведет дамп половины этого сектора диска.

Облегчение жизни

Получив уже три исходных файла, с которыми работали в предыдущей главе, мы заметно усложнили себе жизнь. Изменили ли мы все три файла, с которыми работали, или только два? Вы, вероятно, ассемблировали все три вместо того, чтобы проверять вносились ли какие-либо изменения по сравнению с предыдущим ассемблированием в отдельные файлы.

Но ассемблирование всех исходных файлов при изменении только одного из них происходит довольно медленно и будет еще более замедляться по мере увеличения Dskpatch в размере. Нам было бы очень удобно ассемблировать только те файлы, в которые мы внесли изменения.

К счастью, если вы используете одну из наиболее современных версий Macro Assembler фирмы Microsoft (или их компилятор с языка C), то можете претворить это пожелание в жизнь. Вышеназванные пакеты включают в себя программу, называемую "Make" ("производить, составлять"), которая делает именно то, что мы хотим. Чтобы использовать её, создадим файл (назовем его Dskpatch), который сообщит Make обо всем, что ему надо сделать после обращения:

```
A:\>MAKE DSKPATCH
```

Make ассемблирует только те файлы, которые были изменены.

В файле Dskpatch находится информация, по которой Make определяет имеющиеся зависимости между файлами. Каждый раз, когда вы изменяете файл, DOS обновляет время последней модификации файла (вы можете это увидеть с помощью команды DIR). Make просто сравнивает время создания .ASM и .OBJ-версий файла. Если .ASM-версия имеет более позднее время модификации, чем .OBJ-версия, то Make знает, что необходимо вновь ассемблировать именно этот файл.

Вот и всё, но надо отметить, что Make будет работать правильно, только если вы прилежно выставляете дату и время при загрузке. Без этой информации Make не всегда будет знать, когда вы внесли изменения в файл.

Формат файла для Make

Формат нашего файла Dskpatch, который мы будем использовать с Make, довольно прост:

Листинг 15.1. Файл Make для DSKPATCH.

```
disp_sec.obj:      disp_sec.asm
    masm disp_sec;

Video_io.obj:      VIDEO_IO.ASM
    masm Video_io;

cursor_sec.obj:    cursor.asm
    masm cursor;

DISP_SEC.com:      DISP_SEC.obj Video_io.obj cursor.obj
    link DISP_SEC Video_io cursor;
    exe2bin DISP_SEC DISP_SEC.com
```

Каждый элемент списка имеет одно имя файла слева (перед двоеточием) и одно или более имён файлов справа. Если любой из файлов справа (например, DISP_SEC.ASM в первой строке) имеет более позднее время модификации, чем первый файл (DISP_SEC.OBJ), то Make будет исполнять все приведенные ниже на некотором расстоянии от левой границы экрана команды.

ПРИМЕЧАНИЕ: Вы должны отделять строки команд символом табуляции ("tab"), а не пробелами.

Если ваш ассемблер имеет программу Make, введите эти строки в файл Dskpatch (без расширения) и внесите какие-нибудь небольшие изменения в DISP_SEC.ASM. Затем напечатайте:

```
A:\>MAKE DSKPATH
```

и вы увидите следующее:

```
Microsoft (R) Macro Assembler Version 4.00
Copyright (C) Microsoft Corp 1981,1983,1984,1985. All rights
reserved.

48984 Bytes symbol space free

0 Warning Errors
0 Severe Errors
    link DISP_SEC Video_io cursor;
```

```
Microsoft (R) 8086 Object Linker Version 3.05
Copyright (C) Microsoft Corp 1983,1984,1985. All rights reserved.

Warning:   no stack segment
exe2bin disp_sec disp_sec.asm

A:\>
```

Make выполнил минимальное количество работы, необходимой для создания программы.

Если у вас нет одной из последних версий Microsoft Macro Assembler, которая включает Make, то вы найдете, что эта программа стоит того, чтобы её приобрести. Вы тогда также получите хорошую замену Debug, которая называется Symdeb (от англ. "Symbolic Debugger" - "Символический отладчик"), мы её рассмотрим позже. Теперь опять вернёмся к Dskpatch.

"Починка" DISP_SEC

DISP_SEC в том состоянии, в каком мы его оставили, включает версию процедуры DISP_HALF_SECTOR, применяемую как тестовую процедуру, и главную процедуру. Теперь мы изменим DISP_HALF_SECTOR так, чтобы она стала обычной процедурой, которую мы сможем вызывать из процедуры, которую назовем Disk_io. Тестовая процедура будет находиться в Disk_io вместе с пробной версией процедуры считывания сектора диска.

Прежде всего давайте изменим DISP_SEC так, чтобы получить файл процедур, как мы это делали с Video_io. Замените "END DISP_HALF_SECTOR" на просто END, так как главная процедура теперь будет находиться в Disk_io. Затем удалите выражение "ORG 100h" из CODE_SEG, опять-таки потому, что мы его переместили в другой файл.

Так как мы планируем считать сектор в память, начиная с SECTOR, уже нет необходимости сохранять в программе тестовые данные. Мы можем заменить 16 операторов DB после SECTOR на одну строку:

```
SECTOR      DB      8192      DUP (0)
```

которая резервирует 8192 байт для хранения сектора.

Но ранее мы говорили, что сектора имеют в длину 512 байт. Зачем же нам такая большая область

хранения? Как оказывается, некоторые жёсткие диски (например 300-Мегабайтные) используют сектора очень больших размеров. Столь большие размеры секторов редко используются, но мы должны быть уверены в том, что не считаем сектор, который будет слишком велик для того, чтобы поместиться в области памяти, зарезервированной для SECTOR. В остальной части книги, за исключением SECTOR, о котором мы вскоре расскажем поподробнее, мы будем подразумевать, что сектора имеют в длину только 512 байт.

Теперь нам нужна новая версия DISP_HALF_SECTOR. Старая версия - это не более чем тестовая процедура, которую мы использовали для тестирования DISP_LINE. В новой версии мы будем использовать смещение внутри сектора для того, чтобы иметь возможность вывода на экран 256 байт, начиная с любой точки сектора. Кроме всего прочего - это означает, что мы сможем делать дампы первой половины, второй половины и средних 256 байт сектора. Как обычно, смещение берётся из DX. Ниже приводится новая (окончательная) версия DISP_HALF_SECTOR в DISP_SEC:

Листинг 15.2. Окончательная версия процедуры DISP_HALF_SECTOR.

```
        PUBLIC      DISP_HALF_SECTOR
        EXTRN       SEND_CRLF:NEAR
; Эта процедура отображает половину сектора (256 byte)
;
; DC:DX Смещение в секторе, в байтах должно быть кратно 16 (?)
; какие DC:DX
;
; Используются: DISP_LINE,SEND_CRLF

DISP_HALF_SECTOR PROC    NEAR
    XOR     DX,DX
    PUSH    CX
    PUSH    DX
    MOV     CX,16          ;Отображается 16 строк HALF_SECTOR
    CALL    DISP_LINE
    CALL    SEND_CRLF
    ADD     DX,16
    LOOP    HALF_SECTOR
    POP     DX
    POP     CX
    RET
    INT     20h
DISP_HALF_SECTOR      ENDP
```

А сейчас мы перейдём к процедуре считывания сектора.

__Язык ассемблера__
Считывание сектора

В первой версии READ_SECTOR мы намеренно будем игнорировать возможные ошибки, например отсутствие диска в дисковом. Мы не сможем объяснить в этой книге, как производится обработка ошибок, но вы найдёте процедуры обработки ошибок в версии Dskpatch, поставляемой на диске, который может быть куплен дополнительно к этой книге. Сейчас, однако, нашей задачей является считывание сектора с диска. Вот пробная версия файла DISK_IO.ASM:

Листинг 15.3. Новый файл DISK_IO.ASM.

```
CGROUP      GROUP      CODE_SEG, DATA_SEG
      ASSUME      CS:CGROUP, DS:CGROUP

CODE_SEG      SEGMENT PUBLIC
      ORG      100h

EXTRN      DISP_HALF_SECTOR:NEAR
;
;Процедура считывает первый сектор с диска А: и дампирует
;первую половину сектора.
;
READ_SECTOR      PROC NEAR
      MOV      AL,0          ;Диск А (номер 0)
      MOV      CX,1          ;Считать только один сектор
      MOV      DX,0          ;Считать сектор номер 0
      LEA      BX,SECTOR      ;Сохранить значения сектора здесь
      INT      25h           ;Считать сектор
      POPF          ;Сбросить флаг установки сектора DOS
                      ;Скэнлон, стр.89 и
                      ;E:\PC-SOFT\_RG\_SM\232-16\232-16-
0179-prn.htm
      XOR      DX,DX          ;Установить смещение в SECTOR в 0
      CALL     DISP_HALF_SECTOR ;Дампировать первую половину
      INT      20h
READ_SECTOR      ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
      EXTRN      SECTOR:BYTE
DATA_SEG      ENDS

      END      READ_SECTOR
```

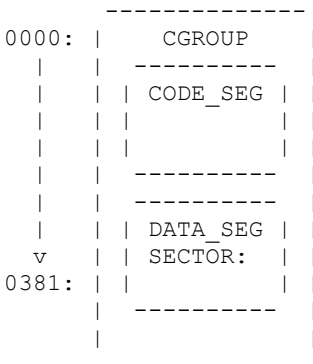
В этой процедуре вы встретили три новых инструкции.
Первая:

```
LEA      BX:SECTOR
```

пересылает АДРЕС или смещение SECTOR (от начала CGROUP) в регистр BX; LEA означает "Загрузить эффективный адрес" (англ. "Load Effective Address"). После выполнения этой инструкции DS:BX будет содержать полный адрес SECTOR, и DOS будет использовать этот адрес для второй новой инструкции, "INT 25h", с которой мы разберёмся чуть позже. (Фактически LEA загружает смещение в регистр BX, не устанавливая регистра DS; нам надо было убедиться, что DS указывает на нужный нам сегмент.)

SECTOR находится в разных файлах с READ_SECTOR. Он содержится в DISP_SEC.ASM. Как мы сообщим ассемблеру, где он находится? Мы используем псевдооператор EXTRN (рис. 15.1):

```
DATA_SEG      SEGMENT PUBLIC
               EXTRN      SECTOR:BYTE
DATA_SEG      ENDS
```



```
LEA BX,SECTOR <--> MOV BX,0381
```

Рис. 15.1.

Инструкция LEA загружает исполнительный адрес

Этот набор инструкций сообщает ассемблеру о том, что SECTOR определён в DATA_SEG, который находится в другом исходном файле, и что SECTOR -переменная, состоящая из байт (а не из слов). В следующих главах мы часто будем использовать псевдооператор EXTRN подобным образом; таким способом мы можем использовать одну и ту же переменную в нескольких исходных файлах. Нам только надо быть аккуратными в определении переменных, то есть каждая переменная должна быть объявлена только один раз.

А сейчас мы вернёмся к инструкции "INT 25h". Это специальная функция, вызываемая из DOS для считывания секторов с диска. Когда DOS получает вызов от "INT 25h", он использует информацию, содержащуюся в регистрах следующим образом:

AL	Номер дисководов (0 = A, 1 = B и т.д.)
CX	Число считываемых секторов
DX	Номер первого сектора, который надо считать (первый сектор - 0)
DS:BX	Адрес пересылки: куда записывать считанные сектора

Число в регистре AL определяет дисковод, с которого DOS будет считывать сектора. Если AL = 0, DOS будет производить считывание с дисковода A.

DOS может за один вызов считать больше, чем один сектор, он считывает число секторов, заданное в CX. В данном случае мы установили CX в единицу, чтобы DOS считал только один сектор длиной 512 байт.

Мы установили DX в нуль и поэтому DOS будет считывать только первый сектор диска. Если хотите, можете изменить номер считываемого сектора, что мы позже также сделаем.

DS:BX содержит полный адрес области памяти, в которую мы хотим поместить сектор(а), считанный с помощью DOS. В данном случае мы записали в DS:BX адрес SECTOR, чтобы потом вызвать DISP_HALF_SECTOR и сделать дамп первой половины первого сектора, считанного с диска, находящегося в дисковом A.

Наконец, обратите внимание на инструкцию POPF, следующую сразу за "INT 25h". Как мы отмечали выше, у микропроцессора 8088 есть регистр состояния, содержащий различные флаги, например флага нуля и переноса. POPF - это специальная инструкция, которая записывает слово из стека в регистр состояния. Зачем нам нужна эта специальная инструкция?

Инструкция "INT 25h" сначала сохраняет в стеке регистр состояния (статуса), а затем адрес возврата. Когда DOS осуществляет возврат из процедуры обработки прерывания 25h, то он оставляет регистр состояния в стеке. DOS поступает таким образом для того, чтобы в случае ошибки при работе с диском (например попытка чтения с дисковода A: без диска в нём) он мог установить флаг переноса. В этой книге мы не будем осуществлять обработку ошибок, но нам

надо удалить регистр состояния из стека, что и делает инструкция POPF.

Теперь вы можете ассемблировать DISK_IO.ASM и переассемблировать DISP_SEC.ASM. Затем произведите компоновку четырёх файлов DISK_IO, DISP_SEC, Video_io и Cursor. Первым в списке файлов в командной строке LINK должен идти Disk_io. Или, если у вас есть Make, добавьте к файлу Dskpatch две строки:

```
disk.io.obj:    disk.io.asm
               masm disk_io;
```

и замените последние три строки на:

```
DISK_IO.com:    disk_io.obj disp_sec.obj video_io.obj cursor.obj
               link Disk_io disp_sec video_io cursor;
               exe2bin Disk_io disk_io.com
```

После того, как вы создадите и запустите .COM-версию Disk_io, то должны увидеть на экране нечто похожее на рис. 15.2.

Позже мы вернёмся к Disk_io, чтобы добавить к нему кое-что, но пока достаточно. В следующей главе мы создадим привлекательное изображение сектора, добавив к изображению несколько графических символов и некоторое количество информации.

```
A:\>disk_io
EB 21 90 49 42 4D 20 20 33 2E 31 00 02 02 01 00          3.1....
02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00        .....
00 00 00 C4 5C 08 33 ED B8 C0 07 8E DB 33 C3 0A    ...
D2 79 0E 89 1E 1E 00 8C 06 20 00 88 16 22 00 B1
02 8E C5 8E D5 BC 00 7C 51 FC 1E 36 C5 38 78 00
...
A:\>
```

Рис. 15.2. Дамп на экране, полученный с помощью DISK_IO.COM

Итог

Теперь, когда у нас есть четыре различных исходных файла, Dskpatch становится более усложнённым. В этой главе мы рассмотрели программу Make, которая помогла упростить работу, ассемблируя только те файлы, в которые мы внесли изменения.

Мы также написали новую процедуру, DISC_IO. Она и SECTOR находятся в разных исходных файлах, поэтому мы использовали в DISC_IO определение EXTRN, чтобы сообщить ассемблеру о SECTOR и о том, что SECTOR - байтовая переменная.

Мы также узнали об инструкции LEA ("Load Effective Address" - загрузить эффективный адрес), которую использовали для загрузки в регистр BX адреса SECTOR.

DISC_IO использует новое прерывание "INT 25h" для считывания секторов с диска в память. Мы применили "INT 25h" для считывания одного сектора в переменную памяти SECTOR, поэтому смогли сделать дамп на экране с помощью DISP_HALF_SECTOR.

Мы также узнали об POPF, инструкции восстановления из стека слова в регистр состояния. Мы использовали эту инструкцию для того, чтобы удалить из стека те флаги, которые DOS не удалил сам после того, как вернулся из "INT 25h".

Созданное изображение половины сектора пока не очень привлекательно, но в следующей главе мы, используя некоторые графические символы, применяемые на IBM PC, сделаем его более приятным с эстетической точки зрения.

Глава 16. Улучшение изображения сектора

Мы подошли к последней главе части II. Всё, что мы делали ранее, было применимо к MS_DOS и микропроцессору 8088 (или 8086 и другим "родственникам" 8088). В части II мы начнём писать процедуры, специфические для IBM Personal Computer и совместимых с ним.

Но прежде чем перейти к ним, мы используем эту главу для добавления к VIDEO_IO ещё нескольких процедур. Мы также модифицируем DISP_LINE в Disp_sec. Все изменения и дополнения будут касаться вывода информации на экран. Значительная их часть будет предназначена для улучшения внешнего вида

изображения, но одна добавит новую информацию: она будет выводить слева числа, соответствующие адресам, как и в дампе Debug. Начнем с графики.

Добавление графических символов

IBM Personal Computer имеет ряд символов рисования линий, которые мы можем использовать для построения прямоугольников вокруг различных частей изображения дампа. Один прямоугольник построим вокруг шестнадцатеричного дампа, а другой вокруг ASCII-дампа. Эти изменения не требуют особой умственной деятельности, надо лишь немного поработать.

Введите следующие определения в начале файла DISP_SEC.ASM между псевдооператором ASSUME и первым псевдооператором SEGMENT, оставив одну или две пустые строки до и после этих определений:
Листинг 16.1. Дополнения к верхней части DISP_SEC.ASM.

```
;
;Графические символы для построения рамки вокруг
;сектора изображения
;
VERTICAL_BAR      EQU 0BAh
HORIZONTAL_BAR    EQU 0CDh
UPPER_LEFT        EQU 0C9h
UPPER_RIGHT       EQU 0BBh
LOWER_LEFT        EQU 0C8h
LOWER_RIGHT       EQU 0BCh
TOP_T_BAR         EQU 0CBh
BOTTOM_T_BAR      EQU 0CAh
TOP_TICK          EQU 0D1h
BOTTOM_TICK       EQU 0CFh
```

Это определения графических символов. Обратите внимание, что мы поместили нуль перед каждым шестнадцатеричным числом. Это сделано для того, чтобы ассемблер знал, что это имена, а не метки. (?) м/б вместо "имена" должно быть "числа" [pause]

Мы могли бы в нашей процедуре просто писать шестнадцатеричные числа вместо этих меток, но определения облегчают понимание процедур. Например, сравните две следующие инструкции:

```
MOV      DL, VERTICAL_BAR
MOV      DL, 0BAh
```

Первая инструкция выглядит более понятной. Ниже приведена новая процедура DISP_LINE, отделяющая различные части изображения символом VERTICAL_BAR,

номер которого 186 (0BAh). Как обычно, дополнения выделены серым фоном:

Листинг 16.2. Изменения в DISP_LINE в DISP_SEC.ASM.

```
DISP_LINE    PROC NEAR
    PUSH     BX
    PUSH     CX
    PUSH     DX
    MOV      BX,DX      ;Смещение более полезно в BX
                        ;Записать разделитель
    MOV      DL,' '
    CALL     WRITE_CHAR
    MOV      DL,VERTICAL_BAR ;Построить левую линию
    CALL     WRITE_CHAR
    MOV      DL,' '
    CALL     WRITE_CHAR
                        ;Записать 16 байт
    MOV      CX,16      ;Дампировать 16 байт
    PUSH     BX          ;Сохранить смещение для
                        ;ASCII_LOOP
HEX_LOOP:
    MOV      DL,SECTOR[BX] ;Выбрать 1 байт
    CALL     WRITE_HEX    ;Перевести его в
                        ;шестнадцатеричную форму
    MOV      DL,' '      ;Записать пробел между числами
    CALL     WRITE_CHAR
    INC      BX
    LOOP     HEX_LOOP

    MOV      DL,VERTICAL_BAR ;Записать разделитель
    CALL     WRITE_CHAR
    MOV      DL,' '
    CALL     WRITE_CHAR

    MOV      CX,16
    PUSH     BX          ;Вернуть смещение в SECTOR
ASCII_LOOP:
    MOV      DL,SECTOR[BX]
    CALL     WRITE_CHAR
    INT      BX
    LOOP     ASCII_LOOP
    MOV      DL,' '
    CALL     WRITE_CHAR
    MOV      DL,VERTICAL_BAR ;Построить правую линию
    CALL     WRITE_CHAR
    POP      DX
    POP      CX
    POP      BX
    RET
DISP_LINE    ENDP
```

Ассемблируйте эту новую версию Disp_sec и снова скомпонуйте файлы (помните о необходимости поместить Disk_io в списке файлов в командной строке LINK первым). После запуска программы вы увидите на экране две линии, разделяющие изображение на две части. Результат показан на рис. 16.1.

Добавление к изображению адресов

Попробуем сделать нечто более серьезное: добавим шестнадцатеричные адреса слева от изображения. Эти числа будут смещениями от начала сектора, поэтому первое число будет 00, следующее 10, затем 20 и т.д.

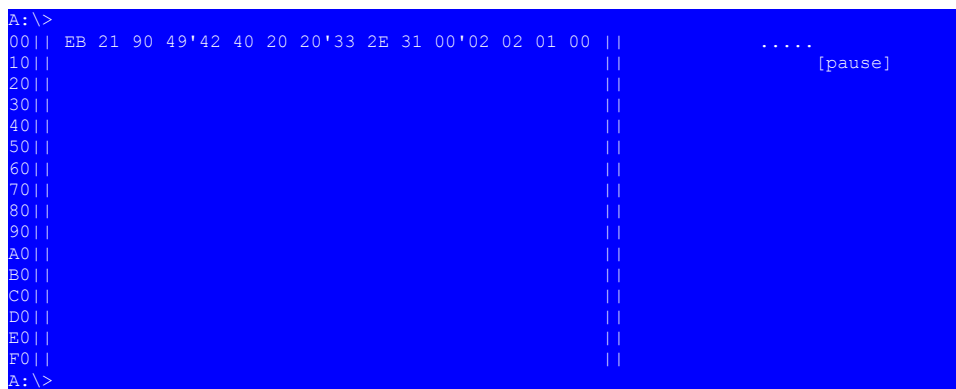


Рис. 16.1. Добавление вертикальных полос.

Процесс довольно прост, так как у нас уже есть процедура WRITE_HEX для записи числа в шестнадцатеричном виде. Но у нас возникает проблема с сектором длиной 512 байт: WRITE_HEX печатает только двузначные шестнадцатеричные числа, в то время как нам нужно три шестнадцатеричных цифры для чисел, превышающих 255.

Решение заключается вот в чём. Так как числа находятся между нулем и 511 (от 0h до 1FFh), первая слева цифра будет либо пробелом, если число (например BCh) меньше 100h, либо единицей. Поэтому, если число будет больше 255, мы просто напечатаем единицу перед шестнадцатеричным числом для младшего байта, а в противном случае, вместо единицы напечатаем пробел. Ниже приводятся дополнения к DISP_LINE, которые будут печатать это трёхзначное шестнадцатеричное число:

Листинг 16.3. Дополнения к процедуре DISP_LINE в DISP_SEC.ASM.

```
DISP_LINE    PROC NEAR
    PUSH     BX
    PUSH     CX
    PUSH     DX
    MOV      BX,DX      ;Смещение более полезно в BX
    MOV      DL,' '
    ;Запись смещения в шестнадцатеричном виде
    CMP      BX,100h    ;Первая цифра 1?
    JB       WRITE_ONE  ;Нет, пробел уже в DL
    MOV      DL,'1'     ;Да, поместить 1 в DL для вывода
WRITE_ONE:
    CALL     WRITE_CHAR
    MOV      DL,BL      ;Скопировать младший байт в DL для
    ;шестнадцатеричного вывода
    CALL     WRITE_HEX
    ;Записать разделитель
    MOV      DL,' '
    CALL     WRITE_CHAR
    MOV      DL,VERTICAL_BAR ;Построить левую линию
```

Результат вы может увидеть на рис. 16.2.

```
A:\>
00|| EB 21 90 49'42 40 20 20'33 2E 31 00'02 02 01 00 || .....
10|| || [pause]
20|| ||
30|| ||
40|| ||
50|| ||
60|| ||
70|| ||
80|| ||
90|| ||
A0|| ||
B0|| ||
C0|| ||
D0|| ||
E0|| ||
F0|| ||
A:\>
```

Рис. 16.2. Добавление чисел слева.

Давайте внимательно посмотрим на изображение. Оно расположено не совсем в центре экрана. Необходимо сдвинуть его вправо примерно на три пробела. Давайте внесём это последнее изменение,

после которого у нас будет законченная версия
DISP_LINE.

Мы могли бы внести это изменение, вызывая
WRITE_CHAR три раза для печати пробела, но мы так
не будем делать. Наоборот, мы добавим к Video_io
ещё одну процедуру, называющуюся
WRITE_CHAR_N_TIMES. Как и предполагает её имя,
эта процедура печатает символ N раз. То есть мы
поместим число N в регистр CX и код символа в DL и
вызовем WRITE_CHAR_N_TIMES для печати N копий
символа, чей ASCII-код помещён в регистр DL. Таким
образом, мы сможем напечатать три пробела,
поместив 3 в регистр CX и 20h (ASCII-код пробела)
в DL.

Вот процедура, которую надо добавить к
VIDEO_IO.ASM:

Листинг 16.4. Добавьте эту процедуру к VIDEO_IO.ASM.

```
PUBLIC      WRITE_CHAR_N_TIMES
;
; Процедура записывает символ N раз
;
; DL          Код символа
; CX          Сколько раз записать символ
;
; Используется:      WRITE_CHAR
;
WRITE_CHAR_N_TIMES PROC NEAR
    PUSH    CX
N_TIMES:
    CALL    WRITE_CHAR
    LOOP    N_TIMES
    POP     CX
    RET
WRITE_CHAR_N_TIMES ENDP
```

У нас уже есть WRITE_CHAR, и поэтому процедура
WRITE_CHAR_N_TIMES получилась простой. Мы решили
написать эту процедуру для того, потому что

(StIV) : или "потому что", или "для того, чтобы"

программа Dskpatch становится яснее для понимания,
если вызывается WRITE_CHAR_N_TIMES, а не
появляется непонятный цикл печати нескольких
одинаковых символов. Кроме того, мы и в дальнейшем
будем использовать эту процедуру.

Ниже приведены изменения в DISP_LINE необходимые
для добавления трёх символов слева от изображения.

Внесите эти изменения в файл DISP_SEC.ASM:

```
    PUBLIC      DISP_LINE
    EXTRN      WRITE_HEX:NEAR
    EXTRN      WRITE_CHAR:NEAR
    EXTRN      WRITE_CHAR_N_TIMES:NEAR
;
;Процедура отображает строку данных или 16 байт, сначала в
;шестнадцатеричном виде, а затем в ASCII-символах.
;DS:DX - Смещение в секторе в байтах
; Используется: WRITE_CHAR, WRITE_HEX, WRITE_CHAR_N_TIMES
; Читается: SECTOR
;
DISP_LINE    PROC    NEAR
    PUSH      BX
    PUSH      CX
    PUSH      DX
    MOV       BX,DX      ;Смещение более полезно в BX
    MOV       DL,' '
    MOV       CX,3       ;Записать три пробела перед строкой
    CALL      WRITE_CHAR_N_TIMES
                ;Записать смещение в шестнадцатеричном виде
    CMP       BX,100h     ;Первая цифра 1?
    JB        WRITE_ONE   ;Нет, пробел уже в DL
    MOV       DL,'1'     ;Да, поместить 1 в DL для вывода
WRITE_ONE:
    .
    .
```

Мы изменили текст файла в трёх местах. Прежде всего, мы добавили один оператор EXTRN для WRITE_CHAR_N_TIMES, так как эта процедура находится в Video_io, а не в этом файле. Мы также изменили блок комментария, чтобы показать, что мы используем эту новую процедуру. Смысл нашего третьего изменения двух строк, которые вызывают WRITE_CHAR_N_TIMES, вполне понятен и не нуждается в разъяснении.

Испытайте новую версию программы так, чтобы увидеть, что изображение теперь центрировано. Затем мы добавим к нему ещё несколько деталей - верхние и нижние линии прямоугольников.

Добавление горизонтальных линий

Добавить горизонтальные линии к изображению не так просто, как это может показаться, потому что есть несколько особых случаев, о которых надо подумать.

Линии на краях должны образовывать углы, а внутри изображения имеются Т-образные стыки.

Для создания горизонтальных линий можно было бы написать длинный список инструкций (с WRITE_CHAR_N_TIMES), но мы не будем этого делать. Есть более простой путь. Напишем ещё одну процедуру, которая называется WRITE_PATTERN и которая будет печатать на экране заданную последовательность символов. Всё, что нам потребуется, это небольшая область памяти для хранения описания каждой последовательности. Используя эту новую процедуру, мы легко можем добавить разделяющие символы в шестнадцатеричное окно.

WRITE_PATTERN использует две новые инструкции LODSB и CLD. Мы дадим их описание после того, как рассмотрим WRITE_PATTERN и способ, которым мы опишем последовательность символов. А теперь введите эту процедуру в файл VIDEO_IO.ASM.

Листинг 16.5. Добавьте эту процедуру к VIDEO_IO.ASM.

PUBLIC WRITE_PATTERN

```
;
;Процедура рисует линию на экране в соответствии с данными,
;занесёнными в форму
;
; DB (символ, число записей символа), 0
; (X) означает, что X может быть повторен любое число раз;
; DS:DX Адрес данных
;
; Используется: WRITE_CHAR_N_TIMES
;
WRITE_PATTERN PROC NEAR
    PUSH    AX
    PUSH    CX
    PUSH    DX
    PUSH    SI
    PUSHF                    ;Сохраняет флаг направления
    CLD                      ;Устанавливает флаг направления на увеличение
    MOV     SI,DX             ;Переносит смещение в регистр SI для
                                ;LODSB
PATTERN_LOOP:
    LOD     SB                ;Поместить символ данных в AL
    OR      AL,AL             ;Достигнут конец данных(0h)?
    JZ      END_PATTERN       ;Да, возврат
    MOV     DL,AL              ;Нет, установить запись символа N раз
    LODSB                    ;Установить счётчик повторений в AL
    MOV     CL,AL              ;и в CX для WRITE_CHAR_N_TIMES
```

```

XOR     CH,CH           ;Обнулить старший байт CX
CALL    WRITE_CHAR_N_TIMES
JMP     PATTERN_LOOP
END_PATTERN:
POPF    ;Восстановить флаг направления
POP     SI
POP     DX
POP     CX
POP     AX
RET
WRITE_PATTERN      ENDP

```

Перед тем, как увидеть работу этой процедуры, давайте посмотрим, как записываются данные для последовательностей символов. Мы поместим данные шаблона верхней линии в файл Disp_sec, туда, где мы будем их использовать. К концу этого файла мы добавим другую процедуру, называющуюся INIT_SEC_DISP, которая инициализирует изображение сектора, выводя на экран изображение его половины; затем модифицируем READ_SECTOR так, чтобы вызывать процедуру INIT_SEC_DISP.

Прежде всего поместите следующие данные сразу после SECTOR (в файле DISP_SEC.ASM) в сегмент данных:

Листинг 16.6. Дополнения к DISP_SEC.ASM

```

TOP_LINE_PATTERN      LABEL      BYTE
DB      ' ',7
DB      UPPER_LEFT, 1
DB      HORIZONTAL_BAR,12
DB      TOP_TICK, 1
DB      HORIZONTAL_BAR,11
DB      TOP_TICK,1
DB      HORIZONTAL_BAR,11
DB      TOP_TICK,1
DB      HORIZONTAL_BAR,12
DB      TOP_T_BAR,1
DB      HORIZONTAL_BAR,18
DB      UPPER_RIGHT,1
DB      0
BOTTOM_LINE_PATTERN   LABEL      BYTE
DB      ' ',7
DB      LOWER_LEFT,1
DB      HORIZONTAL_BAR,12
DB      BOTTOM_TICK,1

```

```
DB    HORIZONTAL_BAR,11
DB    BOTTOM_TICK,1
DB    HORIZONTAL_BAR,11
DB    BOTTOM_TICK,1
DB    HORIZONTAL_BAR,12
DB    BOTTOM_TICK,1
DB    HORIZONTAL_BAR,18
DB    LOWER_RIGHT,1
DB    0
```

Каждое выражение DB содержит часть данных для одной линии. Первый байт - это символ, который надо печатать; второй байт сообщает WRITE_PATTERN о числе повторов символа. Например, мы начинаем верхнюю строку с семи пробелов, затем идёт символ левого верхнего угла, затем двенадцать символов горизонтальных линий и так далее. Последнее DB содержит шестнадцатеричный ноль, который означает конец шаблона.

Давайте продолжим внесение изменений для того, чтобы прежде чем обсуждать работу WRITE_PATTERN, увидеть их результат. Ниже приводится пробная версия INIT_SEC_DISP. Эта процедура печатает верхнюю строку символов, изображение половины сектора и, наконец, нижнюю строку символов. Поместите её в файл DISP_SEC.ASM, перед DISP_HALF_SECTOR:

Листинг 16.7. Добавьте эту процедуру к DISP_SEC.ASM

```
PUBLIC    INIT_SEC_DISP
EXTRN    WRITE_PATTERN:NEAR, SEND_CRLP:NEAR
;
;Процедура инициализирует отображение половины сектора
;
;Используются:WRITE_PATTERN, SEND_CRLF, DISP_HALF_SECTOR
;Считываются: TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN
;
INIT_SEC_DISP    PROC    NEAR
    PUSH    DX
    LEA     DX,TOP_LINE_PATTERN
    CALL    WRITE_PATTERN
    CALL    SEND_CRLP
    XOR     DX,DX                ;Старт в начале сектора
    CALL    DISP_HALF_SECTOR
    LEA     DX,BOTTOM_LINE_PATTERN
    CALL    WRITE_PATTERN
    POP     DX
    RET
```

```
INIT_SEC_DISP    ENDP
```

Мы использовали инструкцию LEA для загрузки адреса в регистр DX для того, чтобы WRITE_PATTERN знала, где искать данные для записи последовательности символов.

Наконец, нам надо внести небольшое изменение в READ_SECTOR в файле DISK_IO.ASM для того, чтобы вызывалась процедура INIT_SECTOR_DISP, а не DISP_HALF_SECTOR и вокруг изображения половины сектора был построен полный прямоугольник:

Листинг 16.8. Изменения в процедуре READ_SECTOR в файле DISK_IO.ASM.

```
EXTRN    INIT_SEC_DISP:NEAR
;
;Процедура считывает с диска A первый сектор и
;дампирует его первую половину
;
READ_SECTOR    PROC    NEAR
    MOV     AL,0           ;Диск A (номер 0)
    MOV     CX,1           ;Учитывается только один сектор
    MOV     DX,0           ;Считывается сектор номер 0
    LEA     BX,SECTOR      ;Где сохранить этот сектор
    INT     25h            ;Считать сектор
    POPF                    ;Сбросить флаг стека, установленный
                          ;DOS
    XOR     DX,DX          ;Сбросить в 0 смещение внутри
                          ;SECTOR
    CALL    INIT_SEC_DISP  ;Дампировать первую половину
                          ;сектора
    INT     20h            ;Возврат в DOS
READ_SECTOR    ENDP
```

Вот и всё, что нужно для печати верхней и нижней строк изображения сектора. Ассемблируйте и скомпонуйте эти два файла (не забудьте ассемблировать три файла, которые мы изменили), пропустите результат через Exe2bin и испытайте его. Рис. 16.3 показывает, как будет теперь выглядеть наше изображение.

Давайте посмотрим, как работает WRITE_PATTERN. Как отмечалось выше, эта процедура использует две новые инструкции. LODSB означает "Загрузить Байт Строки" и она относится к строковым инструкциям, специально созданным для работы со строками символов. Микропроцессору 8088 всё равно, работаем ли

мы со строками символов или чисел, и поэтому LODSB прекрасно выполняет возложенную на неё задачу.

LODSB пересылает байт в регистр AL из ячейки памяти, указанной в регистровой паре DS:SI, которую мы до этого не использовали. Все регистры сегментов в .COM-файле установлены на начало одного сегмента CGROUP, поэтому DS также указывает на этот сегмент. Перед выполнением инструкции LODSB мы переслали смещение в регистр SI с помощью инструкции "MOV SI,DX".

Инструкция LODSB похожа на инструкцию MOV, но она более мощная. За одну инструкцию LODSB микропроцессор 8088 пересылает один байт в регистр AL и либо увеличивает, либо уменьшает регистр SI на единицу. После увеличения регистр SI будет указывать на следующий байт памяти, после уменьшения - на предыдущий.

```
A:\>
00|| EB 21 90 49'42 40 20 20'33 2E 31 00'02 02 01 00 || .....
10|| || [pause]
20|| ||
30|| ||
40|| ||
50|| ||
60|| ||
70|| ||
80|| ||
90|| ||
A0|| ||
B0|| ||
C0|| ||
D0|| ||
E0|| ||
F0|| ||
A:\>
```

Рис. 16.3. Изображение с прямоугольниками

Приращение на единицу - именно то, что нам нужно. Нам надо пройти по всему шаблону, байт за байтом, начиная с самого начала, и именно это делает инструкция LODSB, потому что мы применили ещё одну новую инструкцию CLD (от англ. "Clear Direction Flag"- сбросить флаг направления), сбрасывающую флаг направления - DF. Если мы установим флаг направления, то инструкция LODSB будет, наоборот, уменьшать регистр SI на единицу. Мы используем инструкцию LODSB ещё в нескольких местах Dskpatch,

всегда со сброшенным флагом направления, то есть для приращения.

Кроме LODSB и CLD мы также применили инструкции PUSHF и POPF для сохранения и восстановления регистра флагов, так как позже будем использовать флаг направления в процедуре, которая вызывает WRITE_PATTERN.

Добавление к изображению чисел

Теперь мы уже почти закончили Часть II этой книги. Мы создадим ещё одну процедуру и затем перейдём к Части III, к более сложным и интересным вопросам.

Обратите внимание, что нашему изображению не хватает ряда чисел вдоль верхней строчки. Эти числа - 00 01 02 03 и так далее - позволят нам по столбцу находить адрес каждого байта. Итак, давайте напишем процедуру для печати этого ряда чисел. Добавьте эту процедуру, WRITE_TOP_HEX_NUMBERS в DISP_SECMASM после процедуры INIT_SEC_DISP:

Листинг 16.9. Добавьте эту процедуру к DISP_SEC.ASM.

```
EXTRN    WRITE_CHAR_T_TIMES:NEAR, WRITE_HEX:NEAR,
WRITE_CHAR:NEAR
EXTRN    WRITE_HEX_DIGIT:NEAR, SEND_CRLF:NEAR
;
;Процедура номера (с 0 по F) вдоль верха
;изображения половины сектора
;
;Используются:  WRITE_CHAR_N_TIMES, WRITE_HEX, WRITE_CHAR
;               WRITE_HEX_DIGIT, SEND_CRLF
;
WRITE_TOP_HEX_NUMBERS    PROC    NEAR
    PUSH    CX
    PUSH    DX
    MOV     DL, ' '      ;Записать 9 пробелов перед строкой
    MOV     CX, 9
    CALL    WRITE_CHAR_N_TIMES
    XOR     DH, DH       ;Старт с 0
HEX_NUMBER_LOOP:
    MOV     DL, DH
    CALL    WRITE_HEX
    MOV     DL, ' '
    CALL    WRITE_CHAR
    INC     DH
    CMP     DH, 10h      ;Сделано?
    JB      HEX_NUMBER_LOOP
```

```
MOV     DL, ' '           ;Вывести шестнадцатеричные числа
                               ;над окном ASCII-
MOV     CX, 2
CALL    WRITE_CHAR_N_TIMES
XOR     DL, DL
HEX_DIGIT_LOOP
CALL    WRITE_HEX_DIGIT
INC     DL
CMP     DL, 10h
JB      HEX_DIGIT_LOOP
CALL    SEND_SRLF
POP     DX
POP     CX
RET
WRITE_TOP_HEX_NUMBERS     ENDP
```

Измените процедуру INIT_SEG_DISP (которая тоже находится в DISP_SEC.ASM), как указано ниже для того, чтобы она вызывала WRITE_TOP_HEX_NUMBERS перед тем, как напечатать остальную часть изображения половины сектора:

Листинг 16.10. Изменения в процедуре INIT_SEG_DISP в файле DISP_SEC.ASM.

```
;
;Используются:  WRITE_PATTERN, SEND_CRLF,
;              DISP_HALF_SECTOR
;              WRITE_TOP_HEX_NUMBERS
;Считываются:  TOP_LINE_PATTERN, BOTTOM_LINK_PATTERN
;
INIT_SEG_DISP  PROC      NEAR
    PUSH     DX
    CALL     WRITE_TOP_HEX_NUMBER
    LEA     DX, TOP_LINE_PATTERN
    CALL     WRITE_PATTERN
    CALL     SEND_CRLF
    XOR     DX, DX           ;Старт в начале сектора
    CALL     DISP_HALF_SECTOR
    LEA     DX, BOTTOM_LINE_PATTERN
    CALL     WRITE_PATTERN
    POP     DX
    RET
INIT_SEG_DISP  ENDP
```

Теперь у нас есть законченная процедура изображения половины сектора, результат действия которой вы видите на рис. 16.4.

[illegible]

Рис. 16.4. Законченное изображение половины сектора

Тем не менее между этой версией и окончательной версией остаются некоторые различия. Изменим WRITE_CHAR так, чтобы эта процедура могла печатать все 256 символов, которые может выводить на экран IBM PC, затем добавим процедуру очищения экрана и выполним центровку изображения по вертикали, используя подпрограммы ROM BIOS. Но этим мы займёмся позднее.

Итог

Мы выполнили существенную часть работы по созданию программы Dskpatch, добавив ряд новых процедур и изменив некоторые из написанных ранее. Если вам начинает казаться, что вы теряете нить рассуждений и перестаете понимать то, что мы делаем, обращайтесь к полному листингу Dskpatch в Приложении А. В листинге приведена окончательная версия, но мы надеемся, что вы сможете разобраться в происходящем.

Большая часть изменений, внесенных в этой главе, не содержат каких-либо трюков, напротив, они потребовали от нас скорее упорного труда. Однако мы узнали о двух новых инструкциях: LODSB и CLD. LODSB относится к строковым инструкциям, которые позволяют использовать одну инструкцию для выполнения нескольких операций. Мы применили LODSB в WRITE_PATTERN для считывания последовательности байтов из шаблона, последовательно загружая каждый новый байт в регистр AL. CLD сбрасывает флаг направления, устанавливающий направление приращения. Каждая последующая LODSB инструкция загружает следующий байт из памяти.

Часть III. ROM BIOS IBM PC

Глава 17. Подпрограммы ROM BIOS

Внутри каждого компьютера IBM PC имеются специальные микросхемы, называемые ROM (сокр. от англ. "Read Only Memory" - постоянное запоминающее устройство, ПЗУ). В одной из этих микросхем записаны подпрограммы, обеспечивающие обмен информацией между микропроцессором и остальными частями компьютера. Так как подпрограммы, содержащиеся в ROM, обеспечивают ввод и вывод, то их часто называют BIOS (от англ. "Basic Input Output System" - базовая система ввода-вывода). DOS использует подпрограммы ROM BIOS для вывода символа на экран и записи/считывания с диска, поэтому мы также будем применять их в создаваемых программах.

Мы сосредоточимся на тех подпрограммах BIOS, которые можно применять в Dskpatch. В этот набор входят подпрограммы, работающие с экраным изображением. Среди них имеются такие, которые мы не можем достигнуть без взаимодействия непосредственно с аппаратной частью компьютера, что всегда является достаточно трудным делом.

Video_io, подпрограммы ROM BIOS

Элементы ROM BIOS называются здесь подпрограммами для того, чтобы отличать их от процедур. Вызов процедур производится инструкцией CALL, в то время как подпрограммы вызываются различными инструкциями INT. Например, мы будем использовать инструкцию "INT 10h", чтобы вызывать подпрограммы для работы с экраном, аналогично тому, как мы применяли инструкцию "INT 21h" для вызова подпрограмм в DOS.

"INT 10h", в частности, вызывает подпрограмму Video_io в ROM BIOS. Другие значения вызывают другие подпрограммы, но мы с ними не будем работать, Video_io предоставляет все функции обмена данными с дисплеем, которые нам нужны вне DOS. (Для информации заметим, что DOS вызывает одну из

подпрограмм ROM BIOS при считывании сектора с диска.)

В этой главе мы используем подпрограммы ROM BIOS для того, чтобы добавить к Dskpatch две новые процедуры: очищения экрана и перемещения курсора. Функции, позволяющие выполнять эти действия, очень полезны, но ни одна из них не доступна из DOS. Поэтому мы задействуем подпрограммы ROM BIOS. Позднее мы увидим более интересные вещи, которые можем выполнять с помощью подпрограмм из ROM, но сначала начнём с использования "JNT 10h" для очищения экрана перед тем, как вывести на экран изображение половины сектора.

Инструкция "JNT 10h" является лишь точкой входа в набор различных функций. Помните, что когда мы применяли инструкцию DOS "INT 21h", мы ещё выбирали функцию, помещая её функции в регистр AH. Мы будем выбирать функцию VIDEO_IO тем же способом: помещая номер соответствующей функции в регистр AH (полный список этих инструкций приведен в табл. 17,1).

Таблица 17.1. Функции INT 10h

(AH) = 0	Установить режим экрана. Регистр AL содержит номер режима.
ТЕКСТОВЫЕ РЕЖИМЫ	
(AL) = 0	40 на 25, чёрно-белым.
(AL) = 1	40 на 25, цветной.
(AL) = 2	80 на 25, чёрно-белый.
(AL) = 3	80 на 25, цветной.
(AL) = 7	80 на 25, монохромный адаптер экрана.
ГРАФИЧЕСКИЕ РЕЖИМЫ	
(AL) = 4	320 на 200, цветной.
(AL) = 5	320 на 200, черно-белый.
(AL) = 6	640 на 200, черно-белый.
(AH) = 1	Установить размер курсора.
(CH)	Первая линии курсора. Верхняя линия имеет номер 0 на монохромном, и на цветном графическом мониторе, в то время как нижняя линия имеет номер 7 для цветного графического адаптера и 13 для монохромного адаптера.
(CL)	Допустимые значения: от 0 до 31.
	Последняя линия курсора.

ROM BIOS IBM PC

При загрузке для цветного графического адаптера устанавливается CH = 6 и CL = 7.
Для монохромного экрана: CH = 11 и CL = 12.

- (AH) = 2 Установить позицию курсора.
 (DH,DL) Номер строки и столбца новой позиции курсора; левый верхний угол имеет координаты (0,0).
 (BH) Номер страницы. Это номер страницы экрана. Цветной графический адаптер имеет место для нескольких страниц экрана, но большинство программ использует страницу 0.
- (AH) = 3 Считать позицию курсора.
 (BH) Номер страницы
 На выходе (DH,DL) Строка и столбец курсора
 (CH,CL) Размер курсора
- (AH) = 4 Считать позицию светового пера (смотрите "Техническое Руководство").
- (AH) = 5 Выбрать активную страницу экрана.
 (AL) Номер новой страницы (от 0 до 7 в режимах 0 и 1; от 0 до 3 в режимах 2 и 3).
- (AH) = 6 Прокрутка экрана вверх.
 (AL) Число строк, которые должны быть стёрты внизу окна. Обычная прокрутка стирает одну строку. Нуль означает, что нужно стереть всё окно.
 (CH,CL) Строка и столбец левого верхнего угла окна.
 (DH,DL) Строка и столбец правого нижнего угла окна.
 (BH) Атрибуты, используемые для этих строк.
- (AH) = 7 Прокрутка экрана вниз.
 То же, что и прокрутка вверх (функция 6), но строки вставляются сверху, а не внизу.
- (AH) = 8 Считать символ в позиции курсора и его атрибуты.
 (BH) Страница экрана (только для текстовых режимов)
 (AL) Считанный символ
 (AH) Атрибуты считанного символа (только для текстовых режимов)

(AH) = 9 Напечатать символ с указанными атрибутами в текущей позиции курсора.
(BH) Страница экрана (только для текстовых режимов).
(CX) Сколько раз надо вывести символ с указанными атрибутами на экран.
(AL) Символ, который надо напечатать.
(BL) Атрибуты символа
(AH) = 10 Напечатать символ в текущей позиции курсора (с обычными атрибутами).
(BH) Страница экрана.
(CX) Сколько раз надо напечатать символ.
(AL) Символ, который надо напечатать.
(AH) = 11... Различные графические функции (если вас интересуют детали, смотрите "Техническое Руководство").
13
(AH) = 14 Печать в режиме телетайпа. Напечатать символ и сдвинуть курсор в следующую позицию.
(AL) Печатаемый символ.
(BL) Цвет символа (только для графических режимов).
(BH) Страница экрана (текстовые режимы).
(AH) = 15 Получить текущий режим экрана.
(AL) Режим экрана, установленный в данный момент.
(AH) Число символов на одну строку
(BH) Активная в данный момент страница экрана.

Очищение экрана будет производить "INT 10h", функция номер 6, называемая "Прокрутить активную страницу вверх". Нам не нужно специально прокручивать изображение экрана, но эта функция очищает экран одновременно с прокруткой изображения. Введите в файл CURSOR.ASM ниже приведенную процедуру:

Листинг 17.1. Добавьте эту процедуру и CURSOR.ASM.

```

PUBLIC CLEAR_SCREEN
;
;Процедура очищает экран полностью
;
CLEAR_SCREEN PROC NEAR
    PUSH    AX
    PUSH    BX
```

```

PUSH    CX
PUSH    DX
XOR     AL,AL    ;Очистить все окно
XOR     CX,CX    ;Верхний левый угол в (0,0)
MOV     DH,24    ;Нижняя строка экрана - 24
MOV     DL,79    ;Правая граница в 79 столбце
MOV     BH,7     ;Применить нормальные атрибуты очистки
MOV     AH,6     ;Очистить окно
POP     DX
POP     CX
POP     BX
POP     AX
RET
CLEAR_SCREEN    ENDP

```

Как оказывается, функция номер 6 десятого прерывания ("INT 10h") требует довольно много входной информации, даже если все сводится только к очищению экрана. Отметим, что она обладает мощными способностями: может очистить любую прямоугольную часть экрана - окно (англ. "Window"). Мы установили размер окна во весь экран, задав первую и последнюю строки экрана 0 и 24 и установив значения столбцов 0 и 79. Подпрограмма, которую мы здесь используем, также может очищать экран, закрашивая его белым цветом (для вывода символов черного цвета) или черным (для вывода белых символов). Способ (атрибут) очистки задается инструкцией "MOV BH,7". Число в регистре AL сообщает подпрограмме о количестве строк, на которое надо переместить (прокрутить) изображение. Нуль, установленный в AL, сообщает о том, что надо не прокрутить окно, а только очистить его.

Теперь необходимо изменить тестовую процедуру, READ_SECTOR, так, чтобы она вызывала процедуру очищения экрана CLEAR_SCREEN перед выводом изображения половины сектора. Мы не поместили этот вызов в процедуру INIT_SEC_DISP, так как хотим использовать INIT_SEC_DISP для перепечатки изображения половины сектора, не затрагивая остальной части экрана.

Измените READ_SECTOR, добавив EXTRN для CLEAR_SCREEN и поместив обращение к CLEAR_SCREEN.

Листинг 17.2. Изменения в процедуре READ_SECTOR файле DISK_IO.ASM.

```
EXTRN    INIT_SEC_DISP:NEAR, CLEAR_SCREEN:NEAR
;
;Процедура считывает первый сектор диска A
; и дампирует первую половину сектора
;
READ_SECTOR    PROC NEAR
    MOV     AL,0           ;Диск A (номер 0)
    MOV     CX,1           ;Считать только 1 сектор
    MOV     DX,0           ;Считать сектор номер 0
    LEA     BX,SECTOR      ;Сохранить сектор здесь
    INT     25h            ;Считать сектор
    POPF                    ;Сбросить флаг стека, установленный
                          ;DOS
    XOR     DX,DX          ;Остановить нулевое смещение внутри
                          ;SECTOR
    CALL    CLEAR_SCREEN
    CALL    INIT_SEC_DISP  ;Дампировать первую половину
                          ;сектора
    INT     20h            ;Возврат в DOS
READ_SECTOR    ENDP
```

Перед тем, как запустить новую версию Disk_io, обратите внимание на место расположения курсора на экране. После запуска Disk_io экран очистится, и затем Disk_io начнет печать изображения половины сектора в том месте экрана, в котором находился курсор перед запуском программы - возможно даже в нижней части экрана. Это может произойти вследствие того, что после очистки экрана мы забыли передвинуть курсор обратно вверх. В Бейсике аналогичная команда CLS (очистка экрана) состоит из двух шагов: она очищает экран и затем передвигает курсор в левый верхний угол. Наша процедура этого не делает, поэтому нам придется перемещать курсор самим.

Движение курсора

Функция номер 2 прерывания "INT 10h" устанавливает позицию курсора методом, похожим на действия оператора Бейсика LOCATE. Мы можем использовать новую процедуру GOTO_XY для перемещения курсора в любую точку экрана (например в левый верхний угол после стирания экрана). Введите эту процедуру в файл CURSOR.ASM:

Листинг 17.3

```

PUBLIC      GOTO_XY
;
;Процедура перемещает курсор
;
; DH      Строка  (Y)
; DL      Колонка  (X)
;
GOTO_XY     PROC      NEAR
    PUSH     AX
    PUSH     BX
    MOV      BH,0      ;Вывести страницу 0
    MOV      AH,2      ;Вызвать SET_CURSOR_POSITION
    INT      10h
    POP      BX
    POP      AX
    RET
GOTO_XY     ENDP

```

Для перемещения курсора на строку ниже выведенного изображения мы будем использовать GOTO_XY в улучшенной версии INIT_SEC_DISP. Введите очередные изменения в процедуру INIT_SEG_DISP в файле DISP_SEC.ASM:

Листинг 17.4. Добавьте эту процедуру к CURSOR.ASM.

```

PUBLIC      INIT_SEC_DISP
EXTRN       WRITE_PATTERN:NEAR, SEND_CRLF:NEAR
EXTRN       GOTO_XY:NEAR
;
;Процедура инициализирует вывод изображения половины
;сектора.
;
;Используются:  WRITE_PATTERN, SEND_CRLF,
;               DISP_HALF_SECTOR
;               WRITE_TOP_HEX_NUMBERS, GOTO_XY
; Считываются:  TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN
;
INIT_SEC_DISP  PROC      NEAR
    PUSH      DX
    XOR       DL,DL      ;Сместить курсор в начало третьей строки
    MOV       DH,2
    CALL      GOTO_XY
    CALL      WRITE_TOP_HEX_NUMBERS
    LEA       DX,TOP_LINE_PATTERN

```


Опробуйте новую версию, изображение половины сектора теперь прекрасно центрировано.

После получения доступа к подпрограммам ROM BIOS, работать с экраном стало довольно просто. В следующей главе мы применим другую подпрограмму ROM BIOS, улучшающую WRITE_CHAR таким образом, чтобы эта процедура могла выводить на экран любой символ. Но прежде чем мы продолжим, давайте внесем в программу ещё ряд изменений -процедуру WRITE_HEADER, выводящую в верхней части экрана строку состояния, с указанием текущего дисковода и номера считываемого сектора.

Применение изменяемых переменных

Перед тем, как создать WRITE_HEADER, нам многое придётся изменить. Как правило, процедуры используют заранее установленные неизменяемые числа. READ_SECTOR, например, пока считывает только нулевой сектор с диска в дисковом A. Мы хотим поместить номер дисковода и номер сектора в переменные для того, чтобы этими значениями могли пользоваться другие процедуры.

Нам нужно изменить эти процедуры так, чтобы они применяли переменные, но начнём с того, что поместим все переменные в один файл DSKPATCH.ASM. Так как Dskpatch.asm размещён первым файлом в программе Dskpatch, то переменные легко в нём отыщутся. Ниже приведён текст DSKPATCH.ASM, завершающийся длинным списком переменных:

Листинг 17.5. Новый файл DSKPATCH.ASM.

```
CGROUP    GROUP    CODE_SEG, DATA_SEG
          ASSUME    CS:CGROUP, DS:CGROUP

CODE_SEG  SEGMENT  PUBLIC
          ORG       100h

          EXTRN     CLEAR_SCREEN:NEAR, READ_SECTOR:NEAR
          EXTRN     INIT_SEC_DISP:NEAR

DISK_PATCH PROCNEAR
          CALL      CLEAR_SCREEN
          CALL      READ_SECTOR
          CALL      INIT_SEC_DISP
          INT       20h
```

ROM BIOS IBM PC

```
DISK_PATCH      ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC

PUBLIC          SECTOR_OFFSET
;
;SECTOR_OFFSET - смещение для половины сектора в полном
;               секторе.
;Оно должно быть кратно 16 и не превышать 156
SECTOR_OFFSET    DW      0

PUBLIC  CURRENT_SECTOR_NO, DISK_DRIVE_NO

CURRENT_SECTOR_NO    DW      0      ;Начальный сектор 0

DISK_DRIVE_NO        DB      0      ;Начальный дисковод A:

PUBLIC  LINES_BEFORE_SECTOR, HEADER_LINE_NO

PUBLIC  HEADER_PART_1, HEADER_PART_2
;
;LINES_BEFORE_SECTOR количество линий вверху экрана
;перед изображением половины сектора
;
LINES_BEFORE_SECTOR  DB      2
HEADER_LINE_NO       DB      0
HEADER_PART_1        DB      'Disk ',0
HEADER_PART_2        DB      '      Sector ',0

PUBLIC  SECTOR
;
;Весь сектор до 8192 байт сохраняется в этой части памяти
;
SECTOR    DB      8192      DUP (0)

DATA_SEG  ENDS

END      DISK_PATCH
```

главная процедура, DISK_PATCH, обращается к трём другим процедурам. Все они встречались ранее, а в дальнейшем мы перепишем READ_SECTOR и INIT_SEC_DISP таким образом, чтобы они использовали переменные, помещённые в сегмент данных.

Перед тем как мы сможем использовать Dskpatch, нам надо модифицировать Disp_sec, заменив

определение SECTOR на EXTRN. Нам также потребуется изменить DISK_IO, заменив READ_SECTOR на обычную процедуру, которую можно вызывать из Dskpatch.

Давайте сначала займёмся SECTOR. Так как мы поместили эту переменную в DSKPATCH.ASM, нам надо заменить определение SECTOR в Disp_sec на EXTRN. Внесите эти изменения в DISP_SEC.ASM:

Листинг 17.6. Изменения в DISP_SEC.ASM.

```
DATA_SEG      SEGMENT      PUBLIC
    EXTRN      SECTOR:BYTE
    PUBLIC      SECTOR
SECTOR        DB          512 DUP(0)

TOP_LINE_PATTERN LABEL      BYTE
    DB          ' ',7
    DB          UPPER_LEFT,1
    .
    .
    .
```

Изменим файл DISK_IO.ASM. Это необходимо, так как он содержит только процедуры, а READ_SECTOR использует переменные (а не константы) в качестве номеров сектора и дисковода. Вот новая версия DISK_IO.ASM:

Листинг 17.7. Изменения в DISK_IO.ASM.

```
CGROUP        GROUP        CODE_SEG, DATA_SEG
                ASSUME      CS:CGROUP, DS:CGROUP

CODE_SEG      SEGMENT      PUBLIC
    ORG        100h
    PUBLIC      READ_SECTOR
DATA_SEG      SEGMENT PUBLIC
    EXTRN      SECTOR_BYTE
    EXTRN      DISK_DRIVE_NO:BYTE
    EXTRN      CURRENT-SECTOR-NO:WORD
DATA_SEG      ENDS
    EXTRN      INIT_SEC_DISP:NEAR, CLEAR_SCREEN:NEAR
;
;Процедура считывает один сектор (512 байт) в SECTOR.
;Считываются:      CURRENT_SECTOR_NO, DISK_DRIVE_NO
;Запись в:         SECTOR
;
READ_SECTOR    PROC        NEAR
```

ROM BIOS IBM PC

```
PUSH    AX
PUSH    BX
PUSH    CX
PUSH    DX
MOV     AL,DISK_DRIVE_NO      ;Номер диска
MOV     CX,1                  ;Считать 1 сектор
MOV     DX,CURRENT_SECTOR_NO  ;Логический номер
                                ;сектора
LEA     BX,SECTOR             ;Сохранить сектор в SECTOR
INT     25h                   ;Считать сектор
POPF    ;Сбросить флаг стека, установленный DOS
XOR     DX,DX                 ;Установить нулевое смещение в SECTOR
CALL    CLIR_SCREEN
CALL    INIT_SEC_DISP         ;Дампировать первую половину
INT     20h
POP     DX
POP     CX
POP     BX
POP     AX
RET
READ_SECTOR    ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
               EXTRN    SECTOR:BYTE
DATA_SEG      ENDS
               END
```

Эта версия Disk_io использует переменные DISK_DRIVE_NO и CURRENT_SECTOR_NO для номера дисковода и того сектора, который должен быть считан. Так как эти переменные уже определены в DSKPATCH.ASM, то нам не надо изменять Disk_io при считывании разных секторов и при смене дисковода.

Если вы используете программу Make для ретрансляции DSKPATCH.COM, то необходимо внести некоторые дополнения а Make-файл, называющийся Dskpatch:

Листинг 17.8. Новая версия DSKPATCH.

```
dskpatch.obj:  dskpatch.asm
               masm dskpatch;

disk_io.obj:   disk_io.asm
masm disk_io;
```

Построение заголовка

```
disp_sec.obj:    disp_sec.asm
                masm disp_sec;

Video_io.obj:    VIDEO_IO.ASM
                masm Video_io;

cursor.obj:     cursor.asm
                masm cursor;

dskpatch.com:    dskpatch.obj disk_io.obj disp.obj Video_io.obj
cursor.obj
                link dskpatch disk_io DISP_SEC Video_io cursor;
                exe2bin dskpatch dskpatch.com
```

Если вы не используете Make, то не забудьте переассемблировать все три измененных файла (Dskpatch, Disk_io и DISP_SEC) и опять скомпоновать файлы с Dskpatch.

```
LINK DSKPATCH disk_io DISP_SEC Video_io CURSOR;
EXE2BIN DSKPATCH DSKPATCH.COM
```

Итак, мы внесли некоторые изменения, поэтому, прежде чем двинуться дальше, протестируйте Dskpatch и убедитесь, что он работает, правильно.

Построение заголовка

Теперь, после замены констант на обращения к переменным, мы можем написать процедуру WRITE_HEADER, выводящую в верхнюю часть экрана строку состояния или заголовка. Заголовок будет выглядеть следующим образом:

```
Disk A      Sector 0
```

WRITE_HEADER будет использовать процедуру WRITE_DECIMAL, выводящую текущий номер сектора в десятичном виде. Эта процедура также будет печатать две строки символов "Disk" и "Sector" (каждая будет завершаться пробелом) и букву дисковода, например "A". Мы поместим процедуру в файл VIDEO_IO.ASM.

Начнём с указателя сегмента данных (DATA_SEG), для чего необходимо изменить первую строку

-- 188 --

(оператор GROUP) в VIDEO_IO.ASM таи, чтобы она читалась следующим образом:

```
CGROUP    GROUP    CODE_SEG, DATA_SEG
           Поместите следующую процедуру в Video_io .ASM:
```

Листинг 17.9. Добавьте эту процедуру к VIDEO_IO.ASM.

```
        PUBLIC      WRITE_HEADER
DATA_SEG SEGMENT PUBLIC
    EXTRN  HEADER_LINE_NO:BYTE
    EXTRN  HEADER_PART_1:BYTE
    EXTRN  HEADER_PART_2:BYTE
    EXTRN  DISK_DRIVE_NO:BYTE
    EXTRN  CURRENT_SECTOR_NO:WORD
DATA_SEG ENDS
    EXTRN  GOTO_XY:NEAR
;
; Процедура создаёт заголовок с номерами дисководов и
; сектора
;
; Используются:  GOTO_XY, WRITE_STRING, WRITE_CHAR,
;                WRITE_DECIMAL
;
;                HEADER_LINE_NO, HEADER_PART_1, HEADER_
;                PART_2, DISK_DRIVE_NO, CURRENT_SECTOR_NO
;
WRITE_HEADER PROC     NEAR
    PUSH    DX
    XOR     DL,DL      ;Сместить курсор в строку заголовка
    MOV     DH,HEADER_LINE_NO
    CALL    GOTO_XY
    LEA     DX,HEADER_PART_1
    CALL    WRITE_STRING
    MOV     DL,DISK_DRIVE_NO
    ADD     DL,'A'      ;Печатать дисковод А, В, ....
    CALL    WRITE_CHAR
    LEA     DX,HEADER_PART_2
    CALL    WRITE_STRING
    MOV     DX,CURRENT_SECTOR_NO
    CALL    WRITE_DECIMAL
    POP     DX
    RET
WRITE_HEADER ENDP
```

Процедура WRITE_STRING пока не существует. Она будет выводить на экран строку символов. Две строки HEADER_PART_1 и HEADER_PART_2, уже определены в DSKPATCH.ASM. WRITE_STRING будет использовать DS:DX в качестве адреса строки.

Напишем собственную процедуру вывода на экран строк. Выводимые строки смогут содержать любой символ, включая "\$", который невозможно напечатать с помощью функции DOS номер 9. Там, где DOS использует "\$" для регистрации конца строки, мы будем применять шестнадцатеричный "0". Ниже приведён текст процедуры. Введите её в VIDEO_IO.ASM:

Листинг 17.10. Добавьте эту процедуру к VIDEO_IO.ASM.

```
        PUBLIC      WRITE_STRING
;
; Процедура выводит на экран строку символов
; Строка должна заканчиваться      DB0
;
;      DS:DX      Адрес строки
;
;      Используется:      WRITE_CHAR
;
WRITE_STRING      PROC NEAR
        PUSH      AX
        PUSH      DX
        PUSH      SI
        PUSHF                    ;Сохранить флаг направления
        CLD                    ;Установить направление на увеличение (вперёд)
        MOV       SI,DX         ;Поместить адрес в SI для LODSB STRING_LOOP:
        LODSB                    ;Ввести символ в регистр AL
        OR        AL,AL         ;Уже найден 0?
        JZ        AND_OF_STRING ;Да, строка создана
        MOV       DL,AL         ;Нет, записать символ
        CALL      WRITE_CHAR
        JMP       STRINGL_OOP
AND_OF_STRING:
        POPF                    ;Восстановить флаг направления
        POP       SI
        POP       DX
        POP       AX
RET WRITE_STRING      ENDP
```


Прежде всего, мы узнали об "INT 10h", функции номер 6, используемой для очищения экрана. Мы также узнали (хотя и очень кратко), что эта функция имеет гораздо больше применений, чем те, которые мы использовали в этой книге. Например, вы найдете их полезными при прокрутке частей экрана - в Dskpatch или в собственных программах.

Затем мы использовали функцию 2 прерывания "INT 10h" для перемещения курсора на третью строку экрана (строка номер 2), с которой начали печатать дамп сектора.

Для упрощения работы с программой, мы переписали несколько процедур так, чтобы они могли использовать переменные вместо констант. Теперь можно считывать любые сектора и изменять стиль работы программы, изменив всего несколько основных чисел в DSKPATCH.ASM.

Наконец, мы написали процедуры WRITE_HEADER и WRITE_STRING, выводящие заголовок в верхней части экрана. В следующей главе мы напишем улучшенную версию WRITE_CHAR, которая вместо точек, выводимых для ASCII-символов с кодом, меньшим 32, будет печатать соответствующие им графические символы. Освоив навыки модульного конструирования, мы осуществим все это, не изменяя процедур, использующих WRITE_CHAR.

Глава 18. Окончательный вариант WRITE_CHAR

В последней главе мы нашли хорошее применение подпрограммам ROM BIOS, использовав их для очищения экрана и перемещения курсора. Однако существует ещё много областей, где можно применять ROM BIOS, и с некоторыми из них мы познакомимся в этой главе.

Применяя только DOS, мы не могли выводить на экран все 256 символов символьного набора IBM PC. Поэтому в этой главе мы представим новую версию WRITE_CHAR, печатающую на экране любой ASCII-символ благодаря ещё одной функции Video_io .

Затем мы напишем очередную полезную процедуру, называющуюся CLEAR_TO_END_OF_LINE, очищающую строку символов от точки расположения курсора до правой границы экрана. Она будет

применяться в WRITE_HEADER. Представьте, что мы переходим от сектора номер 10 (две цифры) к сектору номер 9. После того как мы вызовем WRITE_HEADER, чтобы напечатать "9", справа от этой цифры по-прежнему будет показан "0", оставшийся от числа 10. CLEAR_TO_END_OF_UNE сотрет этот ноль, как и любой другой остаток строки.

Новая процедура WRITE_CHAR

Функция 9 ROM BIOS для INT 10h печатает символ с указанными атрибутами в текущей позиции курсора. Атрибуты управляют такими возможностями, как подчеркивание, мерцание и цвет (смотрите описание различных кодов цветов в руководстве по Бейсику в разделе COLOR). В Dskpatch будут использоваться только два атрибута: атрибут 7, который означает нормальный цвет символа и атрибут 70h, в котором ноль означает цвет символа, а 7- цвет фона, то есть в результате образуется инверсное изображение (черные символы на белом фоне). Мы можем устанавливать атрибуты индивидуально для каждого символа, и в дальнейшем создадим инверсный курсор, называющийся призрачным (англ. "phantom"), или псевдокурсором. Однако сейчас при печати символа мы будем использовать только нормальный атрибут.

INT 10h, функция 9 печатает символ с указанным атрибутом в текущей позиции курсора. В отличие от DOS она не перемещает курсор на следующую позицию (если только не печатается несколько копий символа одновременно). Позднее в другой процедуре мы найдем применение этому факту, но сейчас нам нужна только одна копия каждого символа, поэтому мы будем перемещать курсор сами.

Ниже приведен текст очередной версии WRITE_CHAR, печатающей один символ и затем смещающий курсор на одну позицию вправо. Введите её в файл VIDEO_IO.ASM:

Листинг 18.1. Изменения в процедуре WRITE_CHAR в файле VIDEO_IO.ASM.

```
PUBLIC      WRITE_CHAR
EXTRN      CURSOR_RIGHT:NEAR
;
; Процедура выводит символ на экран, применяя подпрограмму
; ROM BIOS,
```

7 6588

-- 193 --

Новая процедура WRITE_CHAR

```
; причём отображает все 256 символов
; Процедура производит подготовительную работу по установке
; позиции курсора;
; DL Байт, выводимый на экран;
; Используется: CURSOR_RIGHT
WRITE_CHAR PROC NEAR
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     AH,9           ;Вызвать подпрограмму вывода символов-
                           ;атрибутов
    MOV     BH,0           ;Задать вывод нулевой страницы
    MOV     CX,1           ;Записать только один символ
    MOV     AL,DL          ;Выводимый символ
    MOV     BL,7           ;Нормальный атрибут
    INT     10h            ;Записать символ и атрибут
    CALL    CURSOR_RIGHT   ;Перейти к следующей позиции
                           ;курсора
    POP     DX
    POP     CX
    POP     BX
    POP     AX
    RET
WRITE_CHAR ENDP
```

Читая эту процедуру, вы, возможно, удивились, зачем мы включили в неё инструкцию "MOV BH,0". Если у вас графический адаптер дисплея, то он в обычном текстовом режиме имеет четыре страницы текста. Мы используем только первую страницу, номер которой 0, поэтому эта инструкция и присутствует здесь.

Что касается курсора, то WRITE_CHAR применяет процедуру CURSOR_RIGHT для его перемещения на одну позицию вправо или на начало следующей строки, если курсор оказался в столбце, номер которого больше 79. Поместите следующую процедуру в файл CURSOR.ASM:

Листинг 18.2. Добавьте эту процедуру к файлу CURSOR.ASM.

```
PUBLIC CURSOR_RIGHT
;
; Процедура сдвигает курсор на одну позицию вправо
; или на следующую строку, если он оказался в конце строки
```

```

; Используется:   SEND_CRLF
;
CURSOR_RIGHT      PROC      NEAR
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     AH,3      ;Считать текущую позицию курсора
    MOV     BH,0      ;на странице 0
    INT     10h       ;Считать позицию курсора
    MOV     AH,2      ;Установить новую позицию курсора
    INC     DL        ;Установить колонку в новое положение
    CMP     DL,79     ;Проверить номер колонки на = 79
    JBE     OK
    CALL    SEND_CRLF ;Переход к следующей строке
    JMP     DONE
OK:      INT     10h
DONE:    POP     DX
    POP     CX
    POP     BX
    POP     AX
    RET
CURSOR_RIGHT      ENDP

```

CURSOR_RIGHT использует две новые функции прерывания INT 10h. Функция 3 считывает, а функция 2 изменяет позицию курсора. Процедура сначала использует функцию 3, чтобы определить позицию курсора, значение которой возвращается в двух байтах, номер столбца в DL и номер строки в DH. Затем CURSOR_RIGHT увеличивает номер столбца (в DL) и сдвигает курсор. Если DL указывал на последний столбец (79), процедура посылает пару символов "возврат каретки"/"перевод строки" для перемещения курсора на следующую строку. Отметим, что в Dskpatch нам не нужно проверять номер столбца (79), но эта проверка делает CURSOR_RIGHT процедурой общего назначения, которую вы сможете использовать в ваших программах.

После этих изменений Dskpatch должен печатать все 256 символов, как показано на рис. 18.1.

Вы можете проверить это, найдя байт со значением, меньшим 20h и увидев в окне ASCII вместо точки, печатавшейся ранее, какой-то странный символ.

7*

-- 195 --

Теперь мы напишем процедуру, стирающую символы строки от текущей позиции курсора до конца строки.

Стирание до конца строки

В предыдущей главе в процедуре CLEAR_SCREEN, для очищения экрана мы применили функцию 6 прерывания INT 21h. Тогда мы отметили, что функция 6 может быть использована для очищения любого прямоугольного окна. Окно при этом может иметь произвольные размеры, начиная от одного символа.

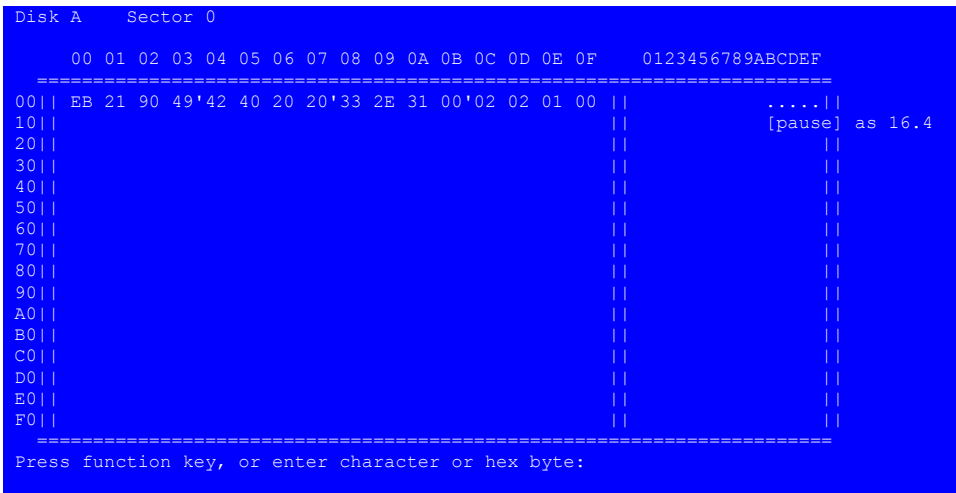


Рис. 18.1. Dskpatch с новой процедурой WRITE_CHAR

Левая граница окна представляет собой столбец, в котором находится курсор. Номер этого столбца мы получили, используя обращение к функции 3 (также применяется процедурой CURSOR_RIGHT). Правая граница окна всегда соответствует столбцу с номером 79. Детали вы можете увидеть в CLEAR_TO_END_OF_LINE. Поместите эту процедуру в файл CURSOR.ASM:

Листинг 18.3. Добавьте эту процедуру к CURSOR.ASM.

```
PUBLIC CLEAR_TO_END_OF_LINE
;
; Процедура очищает строку, начиная с текущего положения
; курсора и до конца строки
;
CLEAR_TO_END_OF_LINE PROC NEAR
```

```
PUSH    AX
PUSH    BX
PUSH    CX
PUSH    DX
MOV     AH,3      ;Считать текущую позицию курсора
XOR     BH,0      ;Страница 0
INT     10h       ;Координаты (X,Y) в DL, DH
MOV     AH,6      ;Подготовиться к очистке до конца строки
XOR     AL,AL     ;Очистить окно
MOV     CH,DH     ;Всё той же строке
MOV     CL,DL     ;Начать в позиции курсора
MOV     DL,79     ;Закончить в конце строки
MOV     BH,7      ;Применить нормальный атрибут
INT     10h
POP     DX
POP     CX
POP     BX
POP     AX
RET
CLEAR_TO_END_OF_LINE    ENDP
```

Мы будем применять эту процедуру в WRITE_HEADER для очищения остатка строки, во время считывания других секторов (этим мы займёмся очень скоро). Пока мы не напишем процедуру, считывающую различные сектора и обновляющую изображение, вы не сможете увидеть, как работает CLEAR_TO_END_OF_LINE совместно с WRITE_HEADER, но давайте исправим процедуру WRITE_HEADER сейчас, чтобы потом к ней не возвращаться. Внесите следующие изменения в процедуру WRITE_HEADER в файле Video_io, чтобы в конце процедуры было обращение к CLEAR_TO_END_OF_LINE:

Листинг 18.4. Изменения в процедуре WRITE_HEADER в файле VIDEO_IO.ASM.

```
        PUBLIC      WRITE_HEADER
DATA_SEG SEGMENT PUBLIC
    EXTRN  HEADER_LINE_NO:BYTE
    EXTRN  HEADER_PART_1 :BYTE
    EXTRN  HEADER_PART_2:BYTE
    EXTRN  DISK_DRIVE_NO:BYTE
    EXTRN  CURRENT_SECTOR_NO:WORD
DATA_SEG ENDS
    EXTRN  GOTO_XY:NEAR, CLEAR_TO_END_OF_LINE: NEAR
```

```
; Процедура записывает заголовок с номером дисководов и
; номером
; считываемого сектора
;
; Используются:      GOTO_XY, WRITE_STRING, WRITE_CHAR,
;                   WRITE_DECIMAL, CLEAR_TO_END_OF_LINE
; Считываются:      HEADER_LINE_NO, HEADER_PART_1, HEADER_
;                   PART_2, DISK_DRIVE_NO,
;                   CURRENT_SECTOR_NO
;
;
WRITE_HEADER      PROC      NEAR
    PUSH        DX
    XOR         DL, DL                ;Переместить курсор в строку заголовка
    MOV         DH, HEADER_LINE_NO
    CALL        GOTO_XY
    LEA         DX, HEADER_PART_1
    CALL        WRITE_STRING
    MOV         DL, DISK_DRIVE_NO
    ADD         DL, 'A'              ;Печатать дисковод A, B, ...
    call        WRITE_CHAR
    lea         dx, HEADER_PART_2
    call        WRITE_STRING
    MOV         DX, CURRENT_SECTOR_NO
    CALL        WRITE_DECIMAL
    CALL        CLEAR_TO_END_OF_LINE ;Очистить предыдущий
                                   ;номер сектора

    POP         DX
    RET
WRITE_HEADER      ENDP
```

Это исправление знаменует не только появление окончательной версии WRITE_HEADER, но и завершение файла CURSOR.ASM. Однако нам ещё не хватает нескольких важных частей Dskpatch. В следующей главе мы напишем центральный диспетчер команд клавиатуры, после чего получим возможность нажимать F1 и F2, чтобы считывать другие сектора диска.

Итог

Эта глава была относительно простой, в ней не было информации о новых инструкциях или трюках. Мы узнали, как использовать INT 10h, функцию номер 9 ROM BIOS, чтобы напечатать на экране любой символ.

Мы также узнали, как считывать позицию курсора с помощью INT 10h функции 3, благодаря чему смогли перемещать курсор вправо после того, как напечатали символ. Проблема заключалась в том, что INT 10h функция 9 не перемещает курсор после того, как напечатает символ. Наконец, мы нашли работу для INT 10h функции 6 - она стирает часть строки, начиная с текущей позиции курсора.

В следующей главе мы опять вернёмся к тому, что мы делали, и займемся созданием программы центрального диспетчера.

Глава 19. Диспетчер

Всегда приятно иметь хорошо написанную программу, делающую что-то полезное; но для того, чтобы эта программа была полноценной, желательно сделать её интерактивной, то есть работающей в режиме диалога. Мы напишем простую процедуру ввода команд с клавиатуры и особую программу - центральный диспетчер, управляющий процедурами. Работа диспетчера будет заключаться в том, чтобы вызывать определённую процедуру при нажатии клавиши. Например, для считывания и вывода на экран предыдущего сектора достаточно нажать клавишу F1, после чего диспетчер вызовет процедуру PREVIOUS_SECTOR ("Предыдущий-сектор"). Чтобы сделать это, нам придётся внести в Dskpatch много изменений. Мы начнём с создания DISPATCHER, центрального диспетчера и ряда процедур форматирования изображения. Затем мы добавим две новые процедуры, PREVIOUS_SECTOR и NEXT_SECTOR которые будут вызываться из DISPATCHER.

Dispatcher

Dispatcher является центральным контролером Dskpatch, потому что и ввод с клавиатуры, и редактирование будут проводиться через него. Задачей DISPATCHER является считывание вводимых символов и вызов соответствующих процедур для выполнения работы. Вскоре вы увидите, каким образом диспетчер все это делает, но сначала давайте посмотрим, где и как он будет располагаться в Dskpatch.

DISPATCHER будет иметь свою собственную строку для ввода команд с клавиатуры, расположенную под

изображением половины сектора. В первой версии процедуры ввода с клавиатуры вы не сможете вводить шестнадцатеричные числа, но позже мы исправим этот недостаток. Введите первые изменения в DSKPATCH.ASM; они добавляют данные для командной строки:

Листинг 19.1. Дополнения к DATA_SEG в DSKPATCH.ASM.

```
HEADER_LINE_NO    DB      0
HEADER_PART_1     DB      ' Диск ',0
HEADER_PART_2     DB      ' Сектор',0
PUBLIC            PROMT_LINE_NO, EDITOR_PROMPT
PROMT_LINE_NO     DB      21
EDITOR_PROMPT     DB      'Нажмите функциональную клавишу
                        или введите'
                        DB      ' введите символ или
                        шестнадцатеричный байт:',0
```

В дальнейшем мы добавим ещё несколько приглашений, например, для ввода номера сектора, и поэтому будем использовать общую процедуру построения строк приглашения - WRITE_PROMPT_LINE. Каждая процедура, применяющая WRITE_PROMPT_LINE, будет снабжать её адресом строки приглашения (в данном случае адресом EDITOR_PROMPT), после чего эта процедура напечатает приглашение на строке 21 (так как в PROMPT_LINE_NO - определён номер строки приглашения - 21). Например, эта новая версия DISK_PATCH (в файле DSKPATCH.ASM) использует WRITE_PROMPT_LINE перед тем как вызвать DISPATCHER:

Листинг 19.2. Дополнения к DISK_PATCH в файле DSKPATCH.ASM.

```
EXTRN    CLEAR_SCREEN:NEAR, READ_SECTOR:NEAR
EXTRN    INIT_SEC_DISP:NEAR, WRITE_HEADER:NEAR
EXTRN    WRITE_PROMPTLIN:NEAR, DISPATCHER:NEAR
DISK_PATCH PROC    NEAR
CALL     CLEAR_SCREEN
CALL     WRITE_HEADER
CALL     READ_SECTOR
CALL     INIT_SEC_DISP
LEA      DX,EDITOR_PROMPT
CALL     WRITE_PROMPT_UN
CALL     DISPATCHER
INT      20h
DISK_PATCH ENDP
```

Диспетчер сам по себе довольно прост, но мы применили в нём несколько новых трюков. Ниже приводится листинг первой версии файла DISPATCH.ASM:

Листинг 19.3. Новый файл DISPATCH.ASM.

```
CGROUP    GROUP    CODE_SEG, DATA_SEG
          ASSUME    CS:CGROUP, DS:CGROUP

CODE_SEG    SEGMENT PUBLIC

          PUBLIC    DISPATCHER
          EXTRN     READ_BYTE:NEAR
;
; Это центральный диспетчер. Во время обычного просмотра и
; редактирования он считывает символы, вводимые с
; клавиатуры, и если символ является командной клавишей
; (например управляет курсором), DISPATCHER вызывает
; соответствующую процедуру для совершения необходимых
; действий. Такое управление введено для специальных ключей,
; представленных в таблице DISPATCH_TABLE, где записаны
; адреса процедур и имена ключей. Если символ не является
; специальной клавишей, то он помещается в буфер сектора
; включается режим редактирования
;
;   Используется:    READ_BYTE
;
DISPATCHER    PROC    NEAR
    PUSH    AX
    PUSH    BX
DISPATCH_LOOP:
    CALL    READ_BYTE    ;Считать символ в AX
    OR      AH,AH        ;AX = 0, если символ не считан,-1
                        ;для расширенного кода.
    JZ      DISPATCH_LOOP ;Символ не считан, считать ещё раз
    JS      SPECIAL_KEY  ;Считать расширенный код
    JMP     DISPATCH_LOOP ;Считать другой символ
SPECIAL_KEY:
    CMP     AL,68        ;F10-выход?
    JE      END_DISPATCH ;Да, выйти
                        ;Использовать BX для просмотра таблицы
;
    LEA     BX,DISPATCH_TABLE
SPECIAL_LOOP:
    CMP     BYTE PTR [BX],0 ;Конец таблицы?
    JE      NOT_IN_TABLE   ;Да, ключа нет в таблице
    CMP     AL,[BX]        ;Это вход в таблицу?
    JE      DISPATCH      ;Да, тогда отправить
    ADD     BX,3           ;Нет, ещё одна попытка
```

__Dispatcher__

```
JMP      SPECIAL_LOOP      ;Проверить следующий вход

DISPATCH:
    INC    BX                ;Отметить адрес процедуры
    CALL   WORD PTR[BX]     ;Вызвать процедуру
    JMP    DISPATCH_LOOP    ;Ждать следующий ключ

NOT_IN_TABLE:                ;Ничего не делать, считать
                              ;следующий символ
    JMP    DISPATCH_LOOP

END_DISPATCH:
    POP    BX
    POP    AX
    RET

DISPATCHER    ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC

CODE_SEG      SEGMENT      PUBLIC
    EXTRN    NEXT_SECTOR:NEAR      ;B DISK_IO.ASM
    EXTRN    PREVIOUS_SECTOR_NEAR  ;B DISK_IO.ASM
CODE_SEG      ENDS
;
;Таблица содержит разрешенные расширенные ASCII-коды и
;адреса процедур, вызываемых нажатием клавиш.
; Формат таблицы
;    DB      72
;расширенный код перемещения курсора вверх
;    DW      OFFSET CGROUP:PHANTOM_UP
;
DISPATCH_TABLE    LABEL      BYTE
    DB      59                ;F1
    DW      OFFSET CGROUP:PREVIOUS_SECTOR
    DB      60                ;F2
    DW      OFFSET CGROUP:NEXT_SECTOR
    DB      0                 ;Конец таблицы
DATA_SEG      ENDS

END
```

DISPATCH_TABLE содержит расширенные ASCII-коды для клавиш F1 и F2. После каждого кода в таблице записан адрес соответствующей процедуры, вызываемой DISPATCHER после считывания этого кода. Например, когда процедура READ_BYTE считывает значение кода клавиши F1 (расширенный

код 59), DISPATCHER вызывает процедуру PREVIOUS_SECTOR.

Для того, чтобы получить адреса процедур, вызываемых диспетчером и содержащихся в таблице, мы применили новый псевдооператор OFFSET. Строка

```
DW OFFSET CGROUP:PREVIOUS_SECTOR
```

сообщает ассемблеру о необходимости применить смещение процедуры PREVIOUS_SECTOR. Это смещение подсчитывается относительно начала группы CGROUP, и именно поэтому необходимо поставить "CGROUP:" перед именем процедуры. Если бы мы не поместили там CGROUP, то ассемблер подсчитывал бы адрес PREVIOUS_SECTOR относительно начала сегмента кода, а это не совсем то, что нужно. (Однако в данном конкретном случае этот CGROUP не так уж необходим, так как в нашей программе сегмент кода загружается первым. Тем не менее, в интересах ясности мы будем везде писать "CGROUP:".)

Обратите внимание на то, что DISPATCH_TABLE содержит и байты, и слова. Это требует некоторых разъяснений. Ранее мы имели дело с таблицами, содержащими данные только одного типа: либо слова, либо байты. Теперь же мы работаем с двумя типами данных, поэтому нам надо указать ассемблеру, какой тип данных используется при применении инструкций CMP или CALL. В случае, если инструкция будет написана следующим образом:

```
CMP [BX],0
```

то ассемблер не поймет, что мы хотим сравнивать слова или байты. Но если мы напишем инструкцию в виде:

```
CMP BYTE PTR [BX],0
```

то ассемблеру станет ясно, что BX указывает на байт, и что мы хотим сравнивать байты. Аналогично, инструкция "CMP WORD PTR [BX],0" будет сравнивать слова. С другой стороны, инструкция "CMP AL,[BX]" проблем не вызывает, так как AL - это байтовый регистр, и ассемблер понимает без разъяснений, что мы хотим сравнивать байты.

Кроме того, как вы помните, инструкция CALL может быть либо типа NEAR, либо типа FAR.

адреса "близкого" вызова ("NEAR CALL") требуется одно слово, в то время как для "дального" вызова требуется два слова. Например, инструкция:

CALL WORD PTR [BX]

посредством "WORD PTR" говорит ассемблеру, что [BX] указывает на одно слово, поэтому ассемблер будет генерировать близкий вызов и использовать то слово, на которое указывает [BX], как адрес, который мы записали в DISPATCH_TABLE. (Для дальнего вызова, который использует адрес, состоящий из двух слов, мы должны были бы использовать инструкцию "CALL DWORD PTR [BX]". DWORD означает "Double Word" -двойное слово.)

Как вы увидите в главе 22, мы легко можем добавить к Dskpatch новые команды клавиатуры, добавив процедуры обработки этих клавиш и новые точки входа в DSKPATCH_TABLE. Однако сейчас, перед тем как мы опробуем новую версию Dskpatch, нам надо написать четыре процедуры. Они называются READ_BYTE, WRITE_PROMPT_LINE, PREVIOUS_SECTOR и NEXT_SECTOR.

READ_BYTE - процедура для считывания с клавиатуры символов и расширенных ASCII-кодов. Окончательная версия этой процедуры сможет считывать специальные клавиши (такие как функциональные клавиши и клавиши курсора), ASCII-символы и двузначные шестнадцатеричные числа. Но в данный момент мы напишем простую версию READ_BYTE для считывания только символа или специальной клавиши. Ниже приведена первая версия KBD_IO.ASM, файла, в котором мы будем хранить все наши процедуры для ввода с клавиатуры:

Листинг 19.4. Новый файл KBD_IO.ASM.

```
CGROUP      GROUP      CODE_SEG, DATA_SEG
              ASSUME     CS:CGROUP, DS:CGROUP

CODE_SEG     SEGMENT PUBLIC

PUBLIC       READ_BYTE

; Процедура считывает ASCII-символ. Это только тестовая
; версия READ_BYTE.
; Возвращает байт в AL код символа (если AH не 0)
;              AH 1, если считан ASCII-символ
```

___ROM BIOS IBM PC___

```
;                                     -1, если считан специальный ключ
;
READ_BYTE    PROC    NEAR
    MOV     AH,7           ;Считать символ без вывода на экран
    INT     21h           ; и поместить в AL
    OR      AL,AL         ;Это расширенный код?
    JZ      EXTENDED_CODE ;Да
NOT_EXT      ENDED:
    MOV     AH,1          ;Считан нормальный ASCII-символ
DONE_READING:
    RET
EXTENDED_CODE:
    INT     21h           ;Считать расширенный код
    MOV     AH,0FFh       ;Считан расширенный код
    JMP     DONE_READING
READ_BYTE    ENDP
CODE_SEG     ENDS
END
```

К VIDEO_IO.ASM мы добавим, как показано ниже, процедуру
WRITE_PROMPT_LINE:

Листинг 19.5. Добавьте эту процедуру к VIDEO_IO.ASM.

```
    PUBLIC    WRITE_PROMT_LINE
    EXTRN    CLEAR_TO_END_OF_LINE:NEAR
    EXTRN    GOTO_XY:NEAR
DATA_SEG     SEGMENT PUBLIC
    EXTRN    PROMPT_LINE_NO:BYTE
DATA_SEG     ENDS
;
; Процедура выводит на экран линию запроса и очищает её до
; конца.
;     DS:DX          Адрес сообщения строки запроса
;
;     Используются: WRITE_STRING, CLEAR_TO_END_OF_LINE,
;                  GOTO_XY
;     Считывается: PROMPT_LINE_NO
;
WRITE_PROMT_LINE    PROCNEAR
    PUSH     DX
    XOR      DL,DL        ;Записать линию запроса
    MOV      DH,PROMPT_LINE_NO ; переместить в неё курсор
    CALL     GOTO_XY
    POP      DX
    CALL     WRITE_STRING
    CALL     CLEAR_TO_END_OF_LINE
    RET
```

```
WRITE_PROMT_LINE    ENDP
```

Эта процедура не очень сложна. Она передвигает курсор на начало строки приглашения (ее номер 21), а затем выводит строку приглашения и стирает её остаток. Когда WRITE_PROMT_LINE выполнена, то курсор находится в конце строки приглашения, неиспользуемый остаток строки стирается процедурой CLEAR_TO_END_OF_LINE.

Считывание других секторов

Наконец мы дошли до этапа, на котором нам понадобились две процедуры - PREVIOUS_SECTOR и NEXT_SECTOR, считывающие и выводящие на экран предыдущий или следующий сектора диска. Добавьте две процедуры к файлу DISK_IO.ASM:

Листинг 19.6. Добавьте эти процедуры к DISK_IO.ASM.

```
    PUBLIC      PREVIOUS_SECTOR
    EXTRN      INIT_SEC_DISP:NEAR, WRITE_HEADER:NEAR
    EXTRN      WRITE_PROMPT_LINE:NEAR
DATA_SEG      SEGMENT PUBLIC
    EXTRN      CURRENT_SECTOR_NO:WORD, EDITOR_PROMPT:BYTE
DATA_SEG      ENDS
;
; Процедура считывает предыдущий сектор, если это возможно
;   Используется: WRITE_HEADER, READ_SECTOR,
;               INIT_SEC_DISP WRITE_PROMPT_LINE
;   Считываются: CURRENT_SECTOR_NO, EDITOR_PROMPT
;   Записывается: CURRENT_SECTOR_NO
;
PREVIOUS_SECTOR    PROC    NEAR
    PUSH    AX
    PUSH    DX
    MOV     AX,CURRENT_SECTOR_NO    ;Получить номер
                                         ;текущего сектора
    OR      AX,AX                    ;Не уменьшать, если уже 0
    JZ      DONT_DECREMENT_SECTOR
    DEC     AX
    MOV     CURRENT_SECTOR_NO    ;Сохранить новый номер
                                         ;сектора
    CALL    WRITE_HEADER
    CALL    READ_SECTOR
    CALL    INIT_SEC_DISP    ;Вывести новый сектор
    LEA     DX,EDITOR_PROMPT
```

```

        CALL    WRITE_PROMPT_LINE
DONT_DECREMENT_SECTOR:
        POP     DX
        POP     AX
        RET
PREVIOUS_SECTOR      ENDP

        PUBLIC  NEXT_SECTOR
        EXTRN   INIT_SEC_DISP:NEAR, WRITE_HEADER:NEAR
        EXTRN   WRITE_PROMPT_LINE:NEAR
DATA_SEG      SEGMENT      PUBLIC
        EXTRN   CURRENT_SECTOR_NO:WORD, EDITOR_PROMPT:BYTE
DATA_SEG      ENDS
;
; Процедура считывает следующий сектор
; Используются: WRITE_HEADER, READ_SECTOR,
;              INIT_SEC_DISP, WRITE_PROMPT LINE
; Считываются: CURRENT_SECTOR_NO, EDITOR_PROMPT
; Записывается: CURRENT_SECTOR_NO
;
NEXT_SECTOR      PROC      NEAR
        PUSH    AX
        PUSH    DX
        MOV     AX,CURRENT_SECTOR_NO
        INC     AX                      ;Переход к следующему сектору

        MOV     CURRENT_SECTOR_NO,AX
        CALL    WRITE_HEADER
        CALL    READ_SECTOR
        CALL    INIT_SEC_DISP          ;Вывести новый сектор
        LEA     DX,EDITOR_PROMPT
        CALL    WRITE_PROMPT_LINE
        POP     DX
        POP     AX
        RET
NEXT_SECTOR      ENDP

```

Теперь мы готовы ассемблировать все созданные или измененные файлы: Dskpatch, Video_io, Kbd_io, Dispatch и disk_io. Когда вы будете проводить компоновку файлов, составляющих Dskpatch, помните, что их теперь семь: Dskpatch, Disp_sec, Disk_io, Video_io, Kbd_io, Dispatch и Cursor.

В случае применения Make, в Dskpatch необходимо внести следующие изменения (обратная косая черта в конце четвёртой строки сообщает Make о том, что мы продолжаем список файлов на следующей строке):

Листинг 19.7. Изменения в Make-файле DSKPATCH.

```
cursor-sec.obj:    cursor.asm
    masm cursor;

dispatch.obj:    dispatch.asm
    masm dispatch;

kbd_io.obj:    kbd_io.asm masm kbd_io;

dskpatch.com:    dskpatch.obj Disk_io.obj disp.obj Video_io.obj
cursor.obj\
    dispatch.obj kbd_io.obj
    link dskpatch disk.io DISP_SEC video.io cursor dispatch kbd_io;
    exe2bin dskpatch dskpatch.com
```

При отсутствии Make, можно воспользоваться следующим небольшим бэтч-файлом, komponующим и создающим .COM-файл:

```
LINK DSKPATCH DISK_IO DISP_SEC Video_io CURSOR DISPATCH KBD_IO;  
EXE2BIN DSKPATCH DSKPATCH.COM
```

По мере того, как мы будем добавлять новые файлы, вам понадобится изменять только этот бэтч-файл, а не печатать весь длинный список компонуемых файлов каждый раз, когда создаёте .COM-файл.

Эта версия Dskpatch имеет три активные клавиши:

F1 - считывает и печатает предыдущий сектор, устанавливаясь на секторе 0;

F2 - считывает следующий сектор:

F10 - клавиша выхода из Dskpatch.

Опробуйте эти клавиши. Изображение на вашем экране будет выглядеть похожим на рис. 19.1.



Рис. 19.1. Dskpatch со строкой приглашения

Философия следующих глав

В этой главе мы прошли гораздо большее расстояние, чем обычно, и в этом отношении вы наверное почувствовали вкус философии, которой мы будем следовать в главах 20...27. Начиная с этой точки, мы будем двигаться вперёд с довольно большой скоростью, так что сможем рассмотреть большее количество примеров, как рекомендуется писать большие программы. Вы также найдёте много процедур, которые можно применять при написании собственных программ.

Эти главы призваны послужить вам в качестве учебного примера, следовательно, в них довольно много новых процедур. Но в последних двух главах книги мы вернёмся к изучению новых вопросов, поэтому вы можете либо продолжать чтение, либо (если хотите) сразу перейти к знакомству с новым материалом.

Если вы стремитесь быстрее начать писать свои собственные процедуры, то вам есть смысл прочесть следующую главу. Вы найдёте в ней множество подсказок, что позволит успешно создавать собственные процедуры. Начиная с главы 21, мы представим различные процедуры и дадим возможность самим разобраться в том, как они работают. Почему? На это есть две причины: обе они касаются вашего становления как программиста, использующего ассемблер. Во-первых, мы хотим, чтобы у вас была библиотека процедур, которыми вы можете воспользоваться при написании собственных программ; но для того, чтобы правильно использовать их, вам надо усовершенствовать свое умение. Во-вторых, представляя здесь большой пример программирования, мы хотим показать вам не только как писать большую программу, но также дать вам ощущение того, как это делается.

Итак, поступите с остальной частью этой книги так, как вам больше нравится. Глава 20 предназначена для тех, кто рвётся писать собственные программы. В главе 21 мы вернёмся к Dskpatch и создадим процедуру печати и передвижения того, что назовём псевдокурсором - инверсным курсором для перемещения по дампу сектора.

Глава 20. Вызов программистам

Книга содержит ещё шесть глав процедур. Если вы хотите попробовать написать их самостоятельно, прочитайте эту главу. Мы здесь обозначим курс, и нанесём его на карту для движения по главам 21 и 22. Затем вы можете попробовать написать процедуры из этих глав, перед тем как прочитать их. Если у вас нет желания испытать свои силы, пропустите эту главу. Она очень коротка, но даёт много пищи воображению.

Если вы решили прочитать эту главу, то вам лучше всего действовать следующим образом: прочитайте один раздел, и затем попытайтесь внести соответствующие изменения в Dskpatch. Когда вы почувствуете, что достигли значительного прогресса, то прочитайте главу с таким же названием, как и название раздела. После того, как вы прочитаете соответствующую главу, вы можете переходить к чтению следующего раздела.

Примечание: Перед тем как вы начнёте вносить изменения, скопируйте созданные файлы. Тогда при прочтении главы 21, вы сможете выбрать, следовать ли приведённым в ней изменениям, либо пользоваться собственной версией программы.

Псевдокурсор

В главе 21 мы поместим на экран два псевдокурсора: один в шестнадцатеричном окне, другой в ASCII-окне. Псевдокурсор похож на обычный, но он не мерцает, а цвет самого курсора и цвет закрываемых им символов меняются на противоположные. Такие курсоры показаны на рис. 20.1.

Псевдокурсор в шестнадцатеричном окне имеет четыре символа в ширину, а курсор в ASCII-окне - один символ.

Как мы создадим псевдокурсор? Каждый символ на экране имеет байт атрибута. Этот байт сообщает IBM PC о том, как высвечивать каждый символ. Код атрибута 7h означает печать нормального символа, в то время как 70h означает печать символа в инверсном изображении. Последнее - именно то, что нам нужно для псевдокурсора, поэтому вопрос состоит в том, как мы изменим атрибут наших символов на 70h?

INT 10h функция 9 выводит на экран символ с указанным атрибутом, а INT 10h функция 8 считывает

код символа в текущей позиции курсора. Мы можем создать псевдокурсор в шестнадцатеричном окне с помощью следующих шагов:



Рис. 20.1. Изображение с псевдокурсорами

- сохранить позицию настоящего курсора (используя INT 10h функцию 3 для считывания позиции курсора и записывая её в переменные).
- передвинуть настоящий курсор в точку размещения псевдокурсора в шестнадцатеричном окне.
- считать коды следующих четырёх символов (функция 8), и напечатать символ с его атрибутом (установленным в 70h).
- восстановить старую позицию курсора.

Псевдокурсор в ASCII-окне создаётся таким же образом. Если у вас есть псевдокурсор в шестнадцатеричном окне, вам останется только добавить дополнительный код для ASCII-окна.

Помните, что эта ваша первая попытка - временная. Когда у вас будет работающая программа с псевдокурсорами, вы можете вернуться назад и переписать её, поэтому вам надо работать с несколькими небольшими процедурами. После того, как вы закончите работу, взгляните на процедуры в главе 21, там можете найти одно из возможных решений.

Простейшее редактирование

Так как у нас теперь есть псевдокурсоры, мы хотим, чтобы они двигались по экрану. Необходимо обратить внимание на граничные условия, то есть предусмотреть, чтобы курсоры не выходили за пределы своих окон. Также необходимо предусмотреть, чтобы курсоры двигались синхронно, ведь они представляют шестнадцатеричное и ASCII-представления одного и того же объекта.

Как мы можем перемещать курсоры? Каждая из клавиш курсора (клавиши со стрелками) посылает особый номер функции: 72 для "курсор вверх", 80 для "курсор вниз", 75 для "курсор влево" и 77 для "курсор вправо". Эти числа надо добавить в DISPATCH_TABLE вместе с адресами четырех процедур перемещения курсора вверх, вниз, влево, вправо.

Чтобы переместить псевдокурсор, необходимо предварительно очистить его изображение, затем изменить его координаты и напечатать изображение курсора снова. Если вы будете аккуратны при выведении псевдокурсоров на экран, то четыре процедуры для их перемещения должны быть довольно простыми.

Всякий раз, когда вводится символ с клавиатуры, Dskpatch должен считать его и заменить байт в позиции курсора на только что считанный символ. Вот шаги, которые надо предпринять для получения простейшего редактирования:

- считать символ с клавиатуры;
- сменить шестнадцатеричное число в шестнадцатеричном окне и символ в ASCII-окне на только что считанный;
- изменить байт в буфере сектора SECTOR.

Небольшая подсказка: не нужно вносить изменения, чтобы получить процедуру редактирования. Для Dispatch требуется всего лишь обращение к новой процедуре (мы назовем её EDIT_BYTE), которая выполняет большую часть работы. EDIT_BYTE предназначена для изменения экрана и SECTOR.

Другие изменения и дополнения к Dskpatch

В главах 23 - 27 изменения будут более сложными. Если вы по-прежнему хотите писать свою собственную

версию, подойдите к этому вопросу так: что ещё, по вашему мнению, должен был бы делать Dskpatch? В оставшихся главах мы использовали следующие идеи.

Мы хотим написать новую версию READ_BYTE, которая будет считывать либо символ, либо двузначное шестнадцатеричное число и будет ожидать, пока мы нажмём на клавишу "Ввод", перед тем, как передать символ Dskpatch. Эта часть нашего "списка желаний" не так проста, как это звучит, и мы посвятим две главы (главы 23 и 24) работе над этой проблемой.

В главе 25 мы займемся охотой за ошибками, а затем, в главе 26, узнаем, как записывать модифицированные сектора обратно на диск, используя функцию DOS INT 26h, очень похожую на INT 25h, применяемую для считывания сектора с диска. (В главе 26 мы не будем проверять наличие ошибок считывания, но проверка представлена в версии Dskpatch, поставляемой на диске. Диск можно приобрести за дополнительную плату.)

И наконец, в главе 27 мы внесём в Dskpatch некоторые изменения, позволяющие просмотреть вторую половину изображения сектора. Однако они не позволят свободно прокручивать изображение сектора, как нам хотелось бы, но, опять-таки, эта возможность представлена в дисковой версии Dskpatch.

Глава 21. Псевдокурсоры

В этой главе мы создадим процедуры записи и стирания курсоров в шестнадцатеричном и в ASCII-окнах. Такой курсор называется псевдокурсором, потому что он не является стандартным курсором PC, физически - это курсор - призрак..., чисто дизайнерское решение оформления экрана, инвертирующее занимаемую курсором область экрана, изменяя цвета фона и самого символа на противоположные. В шестнадцатеричном окне достаточно места, чтобы создать курсор хорошо заметным, шириной в четыре символа. В ASCII-окне псевдокурсор будет иметь в ширину только один символ, так как в этом окне нет промежутков между символами.

В этой главе много процедур и кодов, поэтому мы будем описывать эти процедуры кратко.

INIT_SEC_DISP - это единственная имеющаяся у нас процедура, которая изменяет изображение сектора.

Новое изображение появляется после запуска Dskpatch, при считывании нового сектора. Так как псевдокурсоры будут находиться в изображении сектора, то мы начнём работу с помещения вызова WRITE_PHANTOM в INIT_SEC_DISP. Таким образом, мы будем печатать псевдокурсоры каждый раз, когда печатаем изображение нового сектора.

Ниже приведён текст пересмотренной - и окончательной - версии INIT_SEC_DISP в DISP_SEC.ASM:

Листинг 21.1. Изменения в процедуре INIT_SEC_DISP в файле DISP_SEC.ASM.

```
PUBLIC      INIT_SEC_DISP
    EXTRN   WRITE_PATTERN:NEAR, SEND_CRLF:NEAR
    EXTRN   GOTO_XY:NEAR, WRITE_PHANTOM:NEAR

DATA_SEG    SEGMENT PUBLIC
    EXTRN    LINES_BEFORE_SECTOR:BYTE
    EXTRN    SECTOR_OFFSET:WORD
DATA_SEG    ENDS

;
; Процедура инициализирует вывод на экран половины сектора
;
; Используются: WRITE_PATTERN, SEND_CRLF,
;               DISP_HALF_SECTOR
;               WRITE_TOP_HEX_NUMBERS, GOTO_XY,
;               WRITE_PAHERN
; Читаются: TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN
;           LINES_BEFORE_SECTOR
; Записывается: SECTOR_OFFSET
;
INIT_SEC_DISP    PROC    NEAR
    PUSH    DX
    XOR     DL,DL                ;Установить курсор
    MOV     DH,LINES_BEFORE_SECTOR
    CALL    GOTO_XY
    CALL    WRITE_TOP_HEX_NUMBERS
    LEA     DX,TOP_LINE_PATTERN
    CALL    WRITE_PAHERN
    CALL    SEND_CRLF
    XOR     DX,DX                ;Старт в начале сектора
    MOV     SECTOR_OFFSET,DX     ;Установить смещение сектора 0
    CALL    DISP_HALF_SECTOR
    LEA     DX,BOTTOM_LINE_PATTERN
    CALL    WRITE_PATTERN
    CALL    WRITE_PHANTOM        ;Записать псевдокурсор
    POP     DX
```

```
RET
INIT_SEG_DISP      ENDP
```

Обратите внимание, что мы также добавили в INIT_SEG_DISP использование и инициализацию переменных. Теперь эта процедура устанавливает значение SECTOR_OFFSET в нуль, чтобы изображалась первая половина сектора.

Перейдём к самой процедуре WRITE_PHANTOM. Здесь потребуется слегка поработать. Всего предстоит написать шесть процедур, включая WRITE_PHANTOM. Идея, однако, довольно проста. Прежде всего, мы перемещаем реальный курсор к позиции псевдокурсора в шестнадцатеричном окне и изменяем атрибут следующих четырёх символов на инверсный (атрибут 70h). Это создаст блок белого цвета, имеющий четыре символа в ширину и содержащий шестнадцатеричное число чёрного цвета. Затем мы делаем то же самое с ASCII-окном, но для одного символа. Наконец, мы перемещаем реальный курсор обратно к тому месту, где он был, когда мы начинали. Все процедуры для псевдокурсоров будут расположены в PHANTOM.ASM, за исключением WRITE_ATTRIBUTE_N_TIMES - процедуры, которая устанавливает атрибут символов.

Введите следующие процедуры в файл PHANTOM.ASM:

Листинг 21.2. Новый файл PHANTOM.ASM.

```
CGROUP      GROUP      CODE_SEG, DATA_SEG
            ASSUME      CS:CGROUP, DS:CGROUP

CODE_SEG     SEGMENT PUBLIC

PUBLIC      MOV_TO_HEX_POSITION
EXTRN      GOTO_XY:NEAR
DATA_SEG     SEGMENT PUBLIC
EXTRN      LINES_BEFORE_SECTOR:BYTE
DATA_SEG     ENDS

;
; Процедура перемещает реальный курсор в точку
; расположения псевдокурсора в шестнадцатеричном окне.
;   Используется :      GOTO_XY
;   Считываются  :      LINES_BEFORE_SECTOR, PHANTOM_CURSOR_X,
;                   PHANTOM_CURSOR_Y
;
MOV_TO_HEX_POSITION  PROC    NEAR
```


___Псевдокурсоры___

```
PUSH    AX
PUSH    CX
PUSH    DX
MOV      DH,LINES_BEFORE_SECTOR      ;Строка
                                           ;псевдокурсора (0,0)
ADD      DH,2                        ;Плюс шестнадцатеричная строка и
                                           ;горизонтальная линия
ADD      DH,PHANTOM_CURSOR_Y        ;DH = строка псевдокурсора

MOV      DL,8                        ;Отступ с левой стороны
MOV      CL,3                        ;Каждый столбец шириной 3 символа, поэтому
MOV      AL,PHANTOM_CURSOR_X        ;надо умножить
                                           ;CURSOR_X на 3
MUL      CL
ADD      DL,AL      ;И добавить к отступу, чтобы получить столбец
CALL     GOTO_XY   ; для псевдокурсора
POP      DX
POP      CX
POP      AX
RET

MOV_TO_HEX_POSITION      ENDP

PUBLIC   MOV_TO_ASCII_POSITION
EXTRN    GOTO_XY:NEAR
DATA_SEG SEGMENT PUBLIC
EXTRN    LINES_BEFORE_SECTOR:BYTE
DATA_SEG ENDS
;
; Процедура перемещает реальный курсор в начало
; псевдокурсора в окне ASCII.
;   Используется:      GOTO_XY
;   Считываются:      LINES_BEFORE_SECTOR,PHANTOM_CURSOR_X,
;                     PHANTOM_CURSOR_Y
;
MOV_TO_ASCII_POSITION      PROC      NEAR
PUSH     AX
PUSH     DX
MOV      DH,LINES_BEFORE_SECTOR      ;Строка
;                                           псевдокурсора (0,0)
ADD      DH,2                        ;Плюс шестнадцатеричная строка и
;                                           горизонтальная линия
ADD      DH,PHANTOM_CURSOR_Y        ;DH= строка
;                                           псевдокурсора
MOV      DL,59      ;Отступ с левой стороны
ADD      DL,PHANTOM_CURSOR_X        ;Прибавить CURSOR_X для
;                                           того, чтобы получить координату X
```

```

CALL    GOTO_XY      ; для псевдокурсора
POP     DX
POP     AX
RET
MOV_TO_ASCII_POSITION    ENDP

PUBLIC   SAVE_REAL_CURSOR
;
; Процедура записывает положение реального курсора в двух
; переменных REAL_CURSOR_X и REAL_CURSOR_Y.
;
;   Записываются:    REAL_CURSOR_X, REAL_CURSOR_Y
;
SAVE_REAL_CURSOR    PROC    NEAR
PUSH     AX
PUSH     BX
PUSH     CX
PUSH     DX
MOV      AH,3        ;Считать координаты курсора
XOR      BH,BH        ; на странице 0
INT      10h          ;и поместить в DL,DH
MOV      REAL_CURSOR_Y,DL ;Записать координаты
MOV      REAL_CURSOR_X,DH
POP      DX
POP      CX
POP      BX
POP      AX
RET
SAVE_REAL_CURSOR    ENDP

PUBLIC   RESTORE_REAL_CURSOR
EXTRN    GOTO_XY:NEAR
;
; Процедура восстанавливает реальный курсор на его прежнем
;месте, координаты которого хранятся в REAL_CURSOR_X и
;REAL_CURSOR_Y.
;
;   Используются:    GOTO_XY
;   Считываются:    REAL_CURSOR_X, REAL_CURSOR_Y
;
RESTORE_REAL_CURSOR    PROC    NEAR
PUSH     DX
MOV      DL,REAL_CURSOR_Y
MOV      DH,REAL_CURSOR_X
CALL     GOTO_XY
POP      DX

```

Псевдокурсоры

```
RET

RESTORE_REAL_CURSOR          ENDP

PUBLIC      WRITE_PHANTOM
EXTRN      WRITE_ATTRIBUTE_N_TIMES:NEAR
;
; Процедура применяет CURSOR_X и CURSOR_Y, используя
; MOV_TO_..., в качестве координат псевдокурсора.
; WRITE_PHANTOM записывает псевдокурсор.
;
;   Используются:   WRITE_ATTRIBUTE_N_TIMES,
;                   SAVE-REAL_CURSOR
;                   RESTORE_REAL_CURSOR,MOV_TO_HEX_POSITION
;                   MOV_TO_ASCII_POSITION
;
WRITE_PHANTOM      PROC      NEAR
    PUSH      CX
    PUSH      DX
    CALL      SAVE-REAL_CURSOR
    CALL      MOV_TO_HEX_POSITION      ;Координаты курсора в
;                                     шестнадцатеричном окне
    MOV       CX,4      ;Сделать псевдокурсор шириной в 4 символа
    MOV       DL,70h
    CALL      WRITE_ATTRIBUTE_N_TIMES
    CALL      MOV_TO_ASCII_POSITION    ;Координаты курсора в
;                                     ASCII-окне
    MOV       CX,1      ;Сделать псевдокурсор шириной 1 символ
    CALL      WRITE_ATTRIBUTE_N_TIMES
    CALL      RESTORE_REAL_CURSOR
    POP       DX
    POP       CX
    RET
WRITE_PHANTOM      ENDP
PUBLIC      ERASE_PHANTOM
EXTRN      WRITE_ATTRIBUTE_N_TIMES:NEAR
;
; Процедура стирает псевдокурсор в противоположность
; WRITE_PHANTOM.
;
;   Используются:   WRITE_ATTRIBUTE_N_TIMES,
;                   SAVE_REAL_CURSOR
;                   RESTORE_REAL_CURSOR,
;                   MOV_TO_HEX_POSITION
;                   MOV_TO_ASCII_POSITION
```

```

;
ERASE_PHANTOM      PROC      NEAR
    PUSH    CX
    PUSH    DX
    CALL    SAVE_REAL_CURSOR
    CALL    MOV_TO_HEX_POSITION      ; Коорд. курсора в
;                                   ; шестн. окне
    MOV     CX, 4                    ; Поменять белый на чёрный
    MOV     DL, 7
    CALL    WRITE_ATTRIBUTE_N_TIMES
    CALL    MOV_TO_ASCII_POSITION
    MOV     CX, 1
    CALL    WRITE_ATTRIBUTE_N_TIMES
    CALL    RESTORE_REAL_CURSOR
    POP     DX
    POP     CX
RET
ERASE_PHANTOM      ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT      PUBLIC
REAL_CURSOR_X  DB      0
REAL_CURSOR_Y  DB      0
    PUBLIC    PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
PHANTOM_CURSOR_X  DB      0
PHANTOM_CURSOR_Y  DB      0
DATA_SEG      ENDS

END

```

WRITE_PHANTOM и ERASE_PHANTOM очень похожи. Фактически единственное отличие между ними заключается в используемом атрибуте: WRITE_PHANTOM устанавливает атрибут 70h для получения инверсного изображения, в то время как ERASE_PHANTOM изменяет атрибут обратно на нормальный (7).

Обе эти процедуры сохраняют предыдущее положение реального курсора с помощью процедуры SAVE_REAL_CURSOR. Эта процедура использует прерывание INT 10h, функцию номер 3, для считывания позиции курсора и затем записывает её в двух байтах REAL_CURSOR_X и REAL_CURSOR_Y.

После записи реальной позиции курсора и WRITE_PHANTOM, и ERASE_PHANTOM вызывают процедуру MOV_TO_HEX_POSITION, которая пере-

щает курсор к начальной точке размещения псевдокурсора в шестнадцатеричном окне. Затем процедура `WRITE_ATTRIBUTE_N_TIMES` записывает атрибут инверсного изображения четырех символов, начиная с начальной точки размещения курсора и двигаясь вправо. Таким образом в шестнадцатеричном окне появляется псевдокурсор. Затем, аналогичным образом, `WRITE_PHANTOM` печатает псевдокурсор шириной в один символ в ASCII-окне. Наконец, `RESTORE_REAL_CURSOR` восстанавливает значение реальной позиции курсора, в которой он находился перед обращением к `WRITE_PHANTOM`.

Единственной ненаписанной процедурой осталась процедура `WRITE_ATTRIBUTE_N_TIMES`.

Изменение атрибута символа

Процедура `WRITE_ATTRIBUTE_N_TIMES` делает следующее. Прежде всего она считывает символ, находящийся в месте размещения курсора. Это делается потому, что прерывание `INT 10h`, которую мы используем для установки атрибута символа, функция номер 9 записывает в позиции курсора символ с указанным атрибутом. Таким образом, `WRITE_ATTRIBUTE_N_TIMES` изменяет атрибут, записывая новый атрибут вместе с только что считанным символом. Наконец, процедура перемещает курсор вправо на одну позицию, чтобы мы могли повторить весь описанный процесс `N` раз. Детали вы можете увидеть в самой процедуре; поместите `WRITE_ATTRIBUTE_N_TIMES` в файл `VIDEO_IO.ASM`:

Листинг 21.3. Добавьте эту процедуру к `VIDEO_IO.ASM`.

```
PUBLIC      WRITE_ATTRIBUTE_N_TIMES
EXTRN      CURSOR_RIGHT:NEAR
;
; Процедура устанавливает атрибуты для N символов, начиная с
; текущей позиции курсора.
; CX      Число символов для установки атрибутов
; DL      Новый атрибут
;      Используется:      CURSOR_RIGHT
;
WRITE_ATTRIBUTE_N_TIMES      PROC      NEAR
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     DL,DL      ;Установить новый атрибут
    XOR     BH,BH      ;Страница 0
```

```
MOV     DX,CX           ;CX используется подпрограммой BIOS
MOV     CX,1           ;Установить атрибут для одного символа

ATTR_LOOP:
MOV     AH,8           ;Считать символ под курсором
INT     10h
CALL    CURSOR_RIGHT
DEC     DX             ;Установлены атрибуты для N символов?
JNZ     ATTR_LOOP      ;Нет, продолжать
POP     DX
POP     CX
POP     BX
POP     AX
RET     WRITE_ATTRIBUTE_N_TIMES    ENDP
```

Это первая и последняя версия WRITE_ATTRIBUTE_N_TIMES. Создав её, мы получили окончательную версию файла VIDEO_IO.ASM, так что вам больше не надо изменять его и ассемблировать.

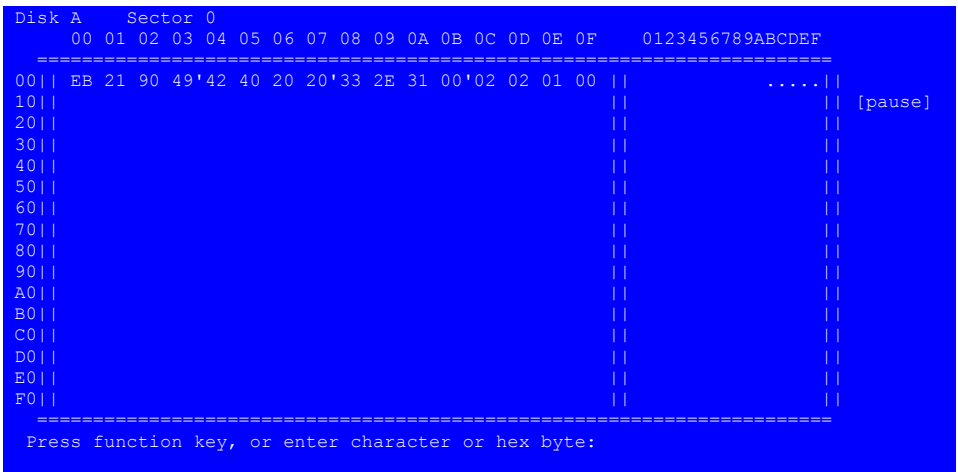


Рис. 21.1. Изображение на экране с псевдокурсорами

Итог

Теперь нам надо компоновать восемь файлов с главной процедурой, по-прежнему находящейся в Dskpatch. В этой главе мы изменили два файла -DISP_SEC и Video_io, и создали один Phantom. Если вы используете Make или бэтч-файл, предложенный в главе 20, то не забудьте добавить к списку новый файл Phantom.

После запуска Dskpatch вы увидите, что он, как и прежде, печатает изображение сектора, но при этом на изображении представлены два псевдокурсора (см. рис. 21.1), Обратите внимание, что настоящий курсор находится именно там, где и должен быть - внизу экрана.

В следующей главе мы добавим процедуры перемещения псевдокурсоров, а также напишем небольшую процедуру для редактирования, позволяющую изменять байт в позиции курсора.

Глава 22. Простейшее редактирование

Мы почти достигли точки, с которой можем начать редактировать изображение сектора - изменять в нем значения чисел. Предварительно необходимо создать процедуру перемещения псевдокурсоров к байтам внутри изображения половины сектора. Оказывается, так как у нас уже есть процедуры ERASE_PHANTOM и WRITE_PHANTOM, эту задачу решить довольно просто.

Перемещение псевдокурсоров

Перемещение псевдокурсоров в любом направлении состоит из трёх основных шагов: стирания курсора в текущей позиции; изменения позиции курсора с помощью изменения значения одной из переменных, PHANTOM_CURSOR_X или PHANTOM_CURSOR_Y; запись псевдокурсора процедурой WRITE_PHANTOM в новой позиции. В процессе перемещения, однако, необходимо следить за тем, чтобы курсор не вышел за границы окна, имеющего 16 байт в ширину и 16 байт в высоту.

Для перемещения псевдокурсоров надо создать четыре новые процедуры, по одной для каждой клавиши со стрелкой. Изменения в DISPATCHER незначительны, необходимо только добавить расширенные ASCII-коды и адреса процедур в таблицу DISPATCH_TABLE для этих клавиш. Вот дополнения, которые надо внести в DISPATCH.ASM, чтобы пробудить к жизни клавиши курсора:

Листинг 22.1. Изменения в DISPATCH.ASM.

```
DATA_SEG      SEGMENT PUBLIC
CODE_SEG      SEGMENT PUBLIC
```

```
EXTRN    NEXT_SECTOR:NEAR                                ;B DISK_IO.ASM
EXTRN    PREVIOUS_SECTOR:NEAR                            ;B DISK_IO.ASM
EXTRN    PHANTOM_UP:NEAR, PHANTOM_DOWN: NEAR             ;B PHANTOM.ASM
EXTRN    PHANTOM_LEFT:NEAR, PHANTOM_RIGHT:NEAR
CODE_SEG    ENDS
;
; Таблица содержит расширенные ASCII-коды и адреса
;процедур, которые вызываются после нажатия
;соответствующей клавиши.
; Формат таблицы
;    DB 72
;Расширенный код перемещения курсора вверх
;    DW OFFSET CGROUP:PHANTOM_UP
;
DISPATCH_TABLE    LABEL    BYTE
    DB    59    ;F1
    DW    OFFSET CGROUP:PREVIOUS_SECTOR
    DB    60    ;F2
    DW    OFFSET CGROUP:NEXT_SECTOR
    DB    72    ;Курсор вверх
    DW    OFFSET CGROUP:PHANTOM_UP
    DB    80    ;Курсор вниз
    DW    OFFSET CGROUP:PHANTOM_DOWN
    DB    75    ;Курсор влево
    DW    OFFSET CGROUP:PHANTOM_LEFT
    DB    77    ;Курсор вправо
    DW    OFFSET CGROUP:PHANTOM_RIGHT
    DB    0     ;Конец таблицы
DATA_SEG    ENDS
```

Как видите, добавить команды перемещения курсора к Dskpatch очень просто: достаточно написать соответствующие процедуры и поместить их имена в DISPATCH_TABLE.

При написании процедур необходимо отметить, что процедуры PHANTOM_UP, PHANTOM_DOWN и т.д. довольно просты. Они похожи одна на другую, а отличаются только обусловленными для каждой из них граничными условиями. Мы уже рассмотрели принципы их работы, и если хотите, то можете попробовать написать их сами, поместив в файл PHANTOM.ASM. Ниже приведены работающие версии процедур.

Листинг 22.2. Добавьте эти процедуры к PHANTOM.ASM.

Перемещение псевдокурсоров

```
; Четыре процедуры перемещения псевдокурсора
;
;   Используются: ERASE_PHANTOM, WRITE_PHANTOM
;   Читаются:     PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
;   Записываются: PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
;
PUBLIC   PHANTOM_UP
PHANTOM_UP    PROC    NEAR
    CALL    ERASE_PHANTOM    ;Очищает текущую позицию курсора
    DEC     PHANTOM_CURSOR_Y ;Перемещает курсор на строку
;                                     вверх
    JNS     WASNT_AT_TOP     ;Если не вверх, то записывает курсор
    MOV     PHANTOM_CURSOR_Y,0    ;Если вверх, оставить там
WASNT_AT_TOP:
    CALL    WRITE_PHANTOM     ;Записать псевдокурсор на новом
;                                     месте
    RET
PHANTOM_UP    ENDP

    PUBLIC   PHANTOM_DOWN
PHANTOM_DOWN  PROC    NEAR
    CALL    ERASE_PHANTOM     ;Очищает текущую позицию курсора
    INC     PHANTOM_CURSOR_Y  ;Перемещает курсор на строку
;                                     вниз
    CMP     PHANTOM_CURSOR_Y, 16    ;Он внизу?
    JB      WASNT_AT_BOTTOM ;Нет, записать псевдокурсор
    MOV     PHANTOM_CURSOR_Y,15    ;Да, оставить здесь
WASNT_AT_BOTTOM:
    CALL    WRITE_PHANTOM     ;Записать псевдокурсор
    RET
PHANTOM_DOWN  ENDP

    PUBLIC   PHANTOM_LEFT
PHANTOM_LEFT  PROC    NEAR
    CALL    ERASE_PHANTOM     ;Очищает текущую позицию курсора
    DEC     PHANTOM_CURSOR_X    ;Перемещает курсор на один
;                                     столбец влево
    JNS     WASNT_AT_LEFT     ;Если не у левого края,
;                                     писать курсор
    MOV     PHANTOM_CURSOR_X,0    ;Если на краю, оставить там
WASNT_AT_LEFT:
    CALL    WRITE_PHANTOM     ;Записать позицию псевдокурсора
    RET
PHANTOM_LEFT  ENDP

    PUBLIC   PHANTOM_RIGHT
```

```
PHANTOM_RIGHT    PROC    NEAR
    CALL    ERASE_PHANTOM    ;Очищает текущую позицию курсора
    INC     PHANTOM_CURSOR_X    ;Перемещает курсор на один
;                                     столбец вправо
    CMP     PHANTOM_CURSOR_X, 16    ;Он уже на правом краю?
    JB      WASNT_AT_RIGHT
    MOV     PHANTOM_CURSOR_Y,15    ;Если на правом краю,
;                                     оставить там
WASNT_AT_RIGHT:
    CALL    WRITE_PHANTOM    ;Записать псевдокурсор
    RET
PHANTOM_RIGHT    ENDP
```

PHANTOM_LEFT и PHANTOM_RIGHT - окончательные версии, но в дальнейшем понадобится изменить PHANTOM_UP и PHANTOM_DOWN. Это произойдет после того, как мы начнём прокручивать изображение (делать скроллинг).

В том состоянии, в котором находится Dskpatch, мы можем просмотреть только первую половину сектора. В главе 27 мы внесём в Dskpatch некоторые изменения и дополнения, которые позволят прокручивать изображение и просматривать остальные части сектора. К тому времени мы изменим и PHANTOM_UP, и PHANTOM_DOWN так, чтобы изображение прокручивалось при перемещении курсора за верхнюю или нижнюю границы экрана. Например, в том случае, если курсор находится в нижней части изображения половины сектора, то нажатие клавиши "курсор вниз" должно сдвинуть изображение вверх на одну строку, добавив снизу ещё одну строку изображения. Прокрутка, однако, довольно трудное занятие, поэтому мы вернёмся к ней только в конце книги. До главы 26 в книге будут рассмотрены вопросы редактирования и ввода с клавиатуры только для первой половины сектора.

Опробуйте новую версию Dskpatch, чтобы увидеть возможность перемещения курсоров по экрану. Они должны двигаться синхронно, каждый в своем окне.

Сейчас мы займёмся редактированием, чтобы иметь возможность изменять байты в секторе, изображение которого выводится на экран.

Простейшее редактирование

У нас уже есть простая процедура ввода с клавиатуры, READ_BYTE, считывающая один символ, введенный с клавиатуры, не дожидаясь нажатия клавиши "Enter". Мы используем старую версию READ_BYTE для введения в Dskpatch возможности редактирования. Затем в следующей главе мы напишем более сложную версию процедуры, которая будет ждать, пока мы нажмём либо "Enter", либо специальную клавишу, например, одну из функциональных клавиш или клавишу перемещения курсора.

Процедура редактирования называется EDIT_BYTE, она будет изменять один байт как на экране, так и в памяти (SECTOR). EDIT_BYTE будет брать символ из регистра DL и записывать его в ту ячейку памяти внутри SECTOR, на которую указывает псевдокурсор в изображении сектора, изменяя при этом само изображение.

В DISPATCHER уже есть соответствующее место, в которое мы можем поместить обращение к EDIT_BYTE. Ниже приводится новая версия DISPATCHER в файле DISPATCH.ASM, в которой есть CALL (обращение) к EDIT_BYTE и изменения, с этим связанные:

Листинг 22.3. Изменения в DISPATCHER в файле DISPATCH.ASM.

```
    PUBLIC    DISPATCHER
    EXTRN    READ_BYTE:NEAR, EDIT_BYTE:NEAR
;
; Это центральный диспетчер, во время обычного
; редактирования эта процедура считывает символ с
; клавиатуры и, если командный ключ (например, управление
; курсором) DISPATCHER вызывает соответствующую процедуру,
; которая выполняет действия. Такое управление построено для
; специальных ключей, список которых хранится в таблице
; DISPATCH_TABLE. В таблице записаны имена процедур и
; адреса, по которым они хранятся. Если символ не является
; специальным ключом, то он будет помещён в буфер сектора в
; режиме редактирования.
;
;    Используется:    READ_BYTE, EDIT_BYTE
;
DISPATCHER    PROC    NEAR
    PUSH    AX
    PUSH    BX
    PUSH    DX
```

```
DISPATCH_LOOP:
    CALL    READ_BYTE      ;Считать символ в AL
    OR      AH,AH          ;AH =0 если символ не считай, -1
;                               для расширенного кода.
    JZ      DISPATCH_LOOP ;Символ не считан, считать ещё
;                               раз.
    JZ      SPECIAL_KEY    ; Считать расширенный код
;                               с символом ничего не делать
    MOV     DL,AL
    CALL    EDIT_BYTE      ;Нормальный символ,
;                               редактировать байт
    JMP     DISPATCH_LOOP  ;Считать другой символ
SPECIAL_KEY:
    CMP     AL, 68          ;F10 - выход?
    JE      END_DISPATCH   ;Да, выйти
;                               Использовать BX для просмотра
;                               таблицы
    LEA     BX,DISPATCH_TABLE
SPECIAL_LOOP:
    CMP     BYTE PTR [BX],0 ;Конец таблицы?
    JE      NOT_IN_TABLE    ;Да, ключа в таблице нет
    CMP     AL,[BX]         ;Это вход в таблицу?
    JE      DISPATCH       ;Да, затем диспетчер
    ADD     BX,3            ;Нет, попробовать следующий
    JMP     SPECIAL_LOOP    ;Проверить следующий вход в
;                               таблицу
DISPATCH:
    INC     BX              ;Отметить адрес процедуры
    CALL    WORD PTR [BX]   ;Вызвать процедуру
    JMP     DISPATCH_LOOP   ;Ждать другой ключ
NOT_IN_TABLE:
;                               Ничего не делать, считать
;                               следующий символ
    JMP     DISPATCH_LOOP

END_DISPATCH:
    POP     DX
    POP     BX
    POP     AX
    RET
DISPATCHER    ENDP
```

Сама процедура EDIT_BYTE выполняет большую часть работы, вызывая другие процедуры, и это одна из возможностей модульного конструирования программ. Используя его принципы, мы сможем писать довольно длинные и сложные процедуры, составляя

список обращений (CALL) к другим процедурам, выполняющим основную работу. Многие из процедур в EDIT_BYTE работают с символом в регистре DL, но этот регистр уже задействован вызовом EDIT_BYTE. Единственная инструкция, кроме CALL (или PUSH и POP), которая нам понадобится в EDIT_BYTE, - это инструкция LEA, устанавливающая адреса приглашения для процедуры WRITE_PROMPT_LINE. Большая часть процедур, вызываемых из EDIT_BYTE, служит для обновления изображения во время редактирования байта. Остальные детали EDIT_BYTE вы увидите, когда мы дойдём до листинга этой процедуры.

Так как EDIT_BYTE изменяет байт на экране, то нам нужна ещё одна процедура, WRITE_TO_MEMORY, изменяющая байт в SECTOR. Процедура WRITE_TO_MEMORY использует координаты PHANTOM_CURSOR_X и PHANTOM_CURSOR_Y для вычисления смещения псевдокурсора относительно SECTOR, затем эта процедура записывает символ (байт) из регистра DL в SECTOR.

Вот новый файл, EDITOR.ASM, который содержит окончательные версии EDIT_BYTE и WRITE_TO_MEMORY:

Листинг 22.4. Новый файл EDITOR.ASM.

```
CGROUP      GROUP      CODE_SEG, DATA_SEG
            ASSUME      CS:CGROUP , DS:CGROUP

CODE_SEG     SEGMENT PUBLIC

DATA_SEG     SEGMENT PUBLIC
    EXTRN    SECTOR:BYTE
    EXTRN    SECTOR_OFFSET:WORD
    EXTRN    PHANTOM_CURSOR_X:BYTE
    EXTRN    PHANTOM_CURSOR_Y:BYTE
DATA_SEG     ENDS
;
; Процедура записывает один байт в SECTOR, в соответствии
; с расположением псевдокурсора.
; DL Байт, записываемый в SECTOR
; Смещение вычисляется как
; OFFSETS = SECTOR_OFFSET = (16*PHANTOM_CURSOR_Y) +
; PHANTOM_CURSOR_X
;
; Считываются:    PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y,
;                SECTOR_OFFSET
```

```

; Записываются:      SECTOR
;
WRITE_TO_MEMORY      PROC      NEAR
    PUSH      AX
    PUSH      BX
    PUSH      CX
    MOV       BX, SECTOR_OFFSET
    MOV       AL, PHANTOM_CURSOR_Y
    XOR       AH, AH
    MOV       CL, 4          ;Умножить PHANTOM_CURSOR_Y на 16
    SHL       AX, CL
    ADD       BX, AX          ;BX = SECTOR_OFFSET + (16*Y)
    MOV       AL, PHANTOM_CURSOR_X
    XOR       AH, AH
    ADD       BX, AX          ;Это адрес!
    MOV       SECTOR[BX], DL  ;Запомнить байт
    POP       CX
    POP       BX
    POP       AX
    RET
WRITE_TO_MEMORY      ENDP

PUBLIC      EDIT_BYTE
EXTRN      SAVE_REAL_CURSOR:NEAR
EXTRN      RESTORE_REAL_CURSOR:NEAR
EXTRN      MOV_TO_HEX_POSITION:NEAR,
EXTRN      MOV_TO_ASCII_POSITION:NEAR
EXTRN      WRITE_PHANTOM:NEAR, WRITE_PROMPT_LINE:NEAR
EXTRN      CURSOR_RIGHT:NEAR, WRITE_HEX:NEAR,
EXTRN      WRITE_CHAR:NEAR
DATA_SEG    SEGMENT      PUBLIC
    EXTRN      EDITOR_PROMPT:BYTE
DATA_SEG    ENDS
;
; Процедура изменяет байт в памяти и на экране;
;   DL Байт записываемый в SECTOR и изменяемый на экране
;
;   Используются:      SAVE_REAL_CURSOR,
;   RESTORE_REAL_CURSOR MOV_TO_HEX_POSITION,
;   MOV_TO_ASCII_POSITION , WRITE_PHANTOM,
;   WRITE_PROMPT_LINE, CURSOR_RIGHT
;   WRITE_HEX, WRITE_CHAR, WRITE_TO_MEMORY
;   Считывается:      EDITOR_PROMPT
;
EDIT_BYTE      PROC      NEAR
    PUSH      DX

```

```

CALL    SAVE_REAL_CURSOR      ;Перейти к шестн. числу
CALL    MOV_TO_HEX_POSITION   ; в шестн. окне
CALL    CURSOR_RIGHT          ;Записать новое число
CALL    MOV_TO_ASCII_POSITION ;Перейти к символу в окне
;                               ASCII
CALL    WRITE_CHAR            ;Записать новый символ
CALL    RESTORE_REAL_CURSOR   ;Вернуть курсор на место
CALL    WRITE_PHANTOM         ;Переписать псевдокурсор
CALL    WRITE_TO_MEMORY       ;Сохранить новый байт в
                               SECTOR
;
    LEA    DX, EDITOR_PROMT
    CALL   WRITE_PROMPT_LINE
    POP    DX
    RET
EDIT_BYTE    ENDP
CODE_SEG     ENDS
END

```

Итог

Dskpatch теперь содержит девять файлов: Dskpatch, Dispatch, Disp_sec, Dis-io, Video_io, Kbd-io, Phantom, Cursor и Editor. В этой главе мы изменили Dispatch, добавив Editor. Все эти файлы небольшие, следовательно, они быстро ассемблируются. Более того, мы можем довольно быстро внести изменения, отредактировав и ассемблировав один из этих файлов и затем опять скомпоновав все файлы вместе.

В текущей версии Dskpatch после нажатия любой клавиши вы увидите, что изменятся число и символ под псевдокурсором. Редактирование работает, но оно пока не совсем безопасно, так как мы можем изменить байт, нажав любую клавишу. Необходимо создать какое-нибудь средство защиты от подобных случайностей, например такое, как нажатие "Ввод" для изменения байта. Это предотвратит от случайных изменений, которые мы можем внести, недостаточно освоив клавиатуру.

Кроме того, текущая версия READ_BYTE не позволяет вводить шестнадцатеричное число для изменения байта. В главе 24 мы перепишем READ_BYTE таким образом, чтобы нам надо было нажимать "Ввод" перед тем, как процедура воспримет новый символ, и чтобы мы могли вводить шестнадцате-

ричное число. Но прежде всего необходимо написать процедуру ввода шестнадцатеричных чисел. В следующей главе мы напишем процедуру, с помощью которой можно вводить шестнадцатеричные, и десятичные числа.

Глава 23. Шестнадцатеричный и десятичный ввод

В этой главе мы познакомимся с двумя новыми процедурами ввода с клавиатуры: первая - для считывания байта, представленного либо двузначным шестнадцатеричным числом, либо одним символом, и вторая - для считывания слова через считывание символов десятичного числа. Это будут процедуры шестнадцатеричного и десятичного ввода.

Обе процедуры достаточно сложны, так что нам придётся использовать для них тестовую программу, прежде чем присоединить их к Dskpatch. Мы будем работать с READ_BYTE, и тестовая процедура будет здесь особенно важна, потому что READ_BYTE потеряет (временно) способность считывать специальные функциональные клавиши. В связи с тем, что Dskpatch также применяет функциональные клавиши, мы не сможем использовать в ней новую версию READ_BYTE. Мы также разберёмся, почему не сможем считывать функциональные клавиши с помощью версии READ_BYTE, приведённой здесь. В следующей главе мы изменим файл, чтобы избавиться от этой проблемы.

Шестнадцатеричный ввод

Давайте начнём с того, что перепишем READ_BYTE. В последней главе READ_BYTE считывала либо обычный символ, либо функциональную клавишу, и затем возвращала байт Dispatch. Если READ_BYTE считывала обычный символ, Dispatch обращался к Editor (редактору) и EDIT_BYTE изменяла байт, на который указывал псевдокурсор. В противном случае Dispatch просматривал специальные функциональные клавиши в DISPATCH_TABLE, чтобы проверить, есть ли там введенный байт; если да, то Dispatch вызывал указанную в таблице процедуру.

Но, как отмечалось в предыдущей главе, старая версия READ_BYTE не предусматривала защиты от случайного изменения байта. Если вы случайно наж-

мёте клавишу на клавиатуре (любую, кроме функциональных), EDIT_BYTE изменит байт под псевдокурсором. Все мы иногда бываем невнимательны, и такие нежелательные изменения сектора могут привести к неприятностям.

Мы изменим READ_BYTE так, чтобы она не воспринимала символ, клавишу которого мы нажали до тех пор, пока не нажмём клавишу "Enter". Мы обеспечим эту возможность, используя прерывание DOS INT 21h, функцию 0Ah, считывающую строку символов. DOS возвращает эту строку только тогда, когда мы нажмём "Enter", так что мы получим защиту от невнимательности. Но по причинам, которые вы увидите позже, READ_BYTE временно лишится возможности воспринимать функциональные клавиши.

Чтобы увидеть, как наши изменения повлияют на READ_BYTE, нам надо написать программу для отдельного тестирования READ_BYTE. Таким образом, если случится что-либо странное, мы будем знать, что это вина READ_BYTE, а не иной части Dskpatch. Работа по написанию тестовой процедуры будет проще, если мы будем использовать некоторые процедуры из Kbd_io, Video_io и Cursor для печати текущей информации во время работы READ_BYTE. Мы будем также использовать процедуры WRITE_HEX и WRITE_DECIMAL для печати возвращаемого кода символа и числа считанных символов. Детали здесь, в TEST.ASM:

Листинг 23.1. Тестовая программа TEST.ASM.

```
CGROUP      GROUP      CODE_SEG, DATA_SEG
      ASSUME      CS:CGROUP, DS:CGROUP

CODE_SEG    SEGMENT      PUBLIC
      ORG      100h

      EXTRN      WRITE_HEX:NEAR, WRITE_DECIMAL:NEAR
      EXTRN      WRITE_STRING:NEAR, SEND_CRLF:NEAR
      EXTRN      READ_BYTE:NEAR

TEST        PROC      NEAR
      LEA      DX, ENTER_PROMPT
      CALL     WRITE_STRING
      CALL     READ_BYTE
      CALL     SEND_CRLF
      LEA      DX, CHARACTER_PROMPT
      CALL     WRITE_STRING
```

```
MOV     DL,AL
CALL    WRITE_HEX
CALL    SEND_CRLF
LEA     DX, CHARACTER_READ_PROMPT
CALL    WRITE_STRING
MOV     DL,AH
XOR     DH,DH
CALL    WRITE_DECIMAL
CALL    SEND_CRLF
INT     20h
TEST    ENDP

CODE_SEG    ENDS

DATA_SEG    SEGMENT PUBLIC
ENTER_PROMPT      DB    'Введите символы: ',0
CHARACTER_PROMPT  DB    'Код символа: ',0
CHARACTER_READ_PROMPT DB 'Число символов: ',0
; Набор переменных: PUBLIC  HEADER_LINE_NO,
; DISK_DRIVE_NO, HEADER_PART_1,  HEADER_PART_2
; PUBLIC
; PROMPT_LINE_NO,
; CURRENT_SECTOR_NO
HEADER_LINE_NO    DB    0
DISK_DRIVE_NO     DB    0
HEADER_PART_1     DB    0
HEADER_PART_2     DB    0
PROMPT_LINE_NO    DB    0
CURRENT_SECTOR_NO DB    0
DATA_SEG    ENDS

END      TEST
```

Попробуйте скомпоновать этот файл с последними версиями Kbd_io, Video.io и Cursor (поместите Test первым в списке LINK). Если вы нажмете любую из функциональных клавиш, Test сообщит, что считано 255 символов. Почему? Мы поместили -1 из AH в DL и установили старший байт DX в нуль, так что значение DX равно 255 (FFh), а не -1 (FFFFh).

Мы не будем так беспечны, когда станем использовать READ_BYTE в Dskpatch. Это тестовая программа, и так как мы знаем, что можно от нее ожидать, мы можем протестировать процедуру READ_BYTE и все её граничные условия. Однако перед тем, как мы перепишем READ_BYTE, нам надо разобраться с одним

свойством TEST.ASM, которое вы возможно заметили:
определение переменных.

Набор инструкций в TEST.ASM для вывода изображения на экран и его форматирования выглядит замечательно. Объявления переменных в конце Test включены только для того, чтобы удовлетворить редактор связей. Когда мы компонуем Test с Kbd_io, Video_io и Cursor, редактор связей ищет объявления переменных, используемых в Kbd_io, Video_io и Cursor. Мы объявили переменные в Dskpatch, но, так как мы не компонуем Dskpatch с этими файлами, необходимо переопределить эти переменные в TEST.ASM. Мы не будем их применять, так как не будем обращаться к процедурам из Video_io и Cursor, которые используют эти переменные. Но в любом случае необходимо их определить для того, чтобы удовлетворить требованиям редактора связей избежать оборванных концов.

Сейчас мы перейдём к изменению процедуры READ_BYTE таким образом, чтобы она воспринимала строку символов. Это не только обезопасит нас от последствий возможной невнимательности при работе с Dskpatch, но и позволит задействовать клавишу "Backspace" (русский аналог "Забой") для удаления символов в случае необходимости, например при неправильном вводе. READ_BYTE будет использовать процедуру READ_STRING для считывания строки символов.

Процедура READ_STRING очень проста, почти что тривиальна, но мы выделили её в отдельную процедуру для того, чтобы иметь возможность изменить её в следующей главе так, чтобы она считывала специальные функциональные клавиши без ожидания нажатия клавиши "Enter". В целях экономии времени мы также добавили три новые процедуры, которые использует READ_BYTE: STRING_TO_UPPER, CONVERT_HEX_DIGIT и HEX_TO_BYTE.

Процедуры STRING_TO_UPPER и HEX_TO_BYTE работают со строками. STRING_TO_UPPER заменяет все строчные буквы в строках на заглавные. Это означает, что вы можете печатать f3 или F3 для ввода шестнадцатеричного числа F3h. Позволяя производить ввод шестнадцатеричных чисел с помощью и строчных, и заглавных букв, мы сделали Dskpatch более дружелюбной по отношению к пользователю.

После вызова STRING_TO_HEX процедура HEX_TO_BYTE переводит строку двузначных шестнадцатеричных чисел, считанную DOS, в строку однобайтных чисел. HEX_TO_BYTE применяет процедуру CONVERT_HEX_DIGIT для перевода каждой шестнадцатеричной цифры в четырёхбитное число.

Почему мы уверены, что DOS не будет считывать больше двух символов? Функция DOS 0Ah считывает строку символов в область памяти, определенную следующим образом:

```
CHAR_NUM_LIMIT    DB    0
NUM_CHARS_READ     DB    0
STRING             DB    80 DUP (0)
```

Первый байт ограничивает число считываемых символов. CHAR_NUM_LIMIT сообщает DOS о том, сколько символов необходимо считать. Если мы установим это число равным трём, DOS будет считывать два символа плюс символ возврата каретки (DOS всегда учитывает этот символ). Все символы, которые мы напечатаем после этих двух символов, будут игнорироваться, и на каждый лишний символ DOS будет выдавать звуковой сигнал, предупреждая о том, что перейдён заданный предел.

Когда мы нажимаем клавишу "Enter", DOS записывает во второй байт, NUM_CHARS_READ, число считанных символов, исключая символ возврата каретки.

NUM_CHARS_READ применяется в процедуре READ_BYTE для определения того, что напечатано: один символ или двузначное шестнадцатеричное число. Если NUM_CHARS_READ был установлен в единицу, то READ_BYTE возвращает в регистр AL символ. Если NUM_CHARS_READ равен двум, то READ_BYTE использует процедуру HEX_TO_BYTE для перевода двузначной шестнадцатеричной строки в байт.

Без дальнейших хлопот представим новую версию файла KBD_IO.ASM со всеми четырьмя новыми процедурами:

Листинг 23.2. Новая версия KBD_IO.ASM.

```
CGROUP    GROUP    CODE_SEG, DATA_SEG
          ASSUME    CS:CGROUP, DS:CGROUP

CODE_SEG  SEGMENT  PUBLIC
```

```
PUBLIC      SRTING_TO_UPPER
;
;Процедура преобразует строку, применяя строчный формат DOS
;для всех заглавных букв.
;
;      DS:DX      Адрес буфера строки
;
STRING_TO_UPPER  PROC      NEAR
    PUSH     AX
    PUSH     BX
    PUSH     CX
    MOV      BX,DX
    INC      BX      ;Установить счёт символов
    MOV      CL,[BX]  ;Счёт символов во втором байте буфера
    XOR      CH,CH    ;Очистить старший байт счетчика
UPPER_LOOP:
    INC      BX      ;Установить на следующий символ в буфере
    MOV      AL,[BX]
    CMP      AL,V      ;Буква заглавная?
    JB       NOT_LOWER
    CMP      AL,'z'
    JA       NOT_LOWER
    ADD      AL,'A'-'a' ;Преобразовать в заглавную букву
    MOV      [BX],AL
NOT_LOWER:
    LOOP     UPPER_LOOP
    POP      CX
    POP      BX
    POP      AX
    RET
STRING_TO_UPPER  ENDP
;
;Процедура преобразует символ из ASCII-(шестнадц.) в 4 бита
;
;      AL      Преобразуемый символ
; Возвращаются      AL      4 бита
;      CF      Устанавливается для ошибок, в другом случае пуст
;
CONVERT_HEX_DIGIT  PROC      NEAR
    CMP      AL,'0'      ;Это цифра?
    JB       BAD_DIGIT    ;нет оператора
    CMP      AL,'9'      ;Пока не уверен
    JA       TRY_HEX      ;Может быть шестн. цифра
    SUB      AL,'0'      ;Если десятичная цифра,
;                          преобразовать в 4 бита
;
```

```

    CLC                ;Очистить флаг переноса, ошибки нет
    RET
TRY_HEX:
    CMP     AL,'A'      ;ещё не уверен
    JB      BAD_DIGIT   ;Не шестнадцатеричное число
    CMP     AL,'A'      ;ещё не уверен
    JA      BAD_DIGIT   ;Не шестнадцатеричное число
    SUB     AL,'A'-10    ;Шестнадцатеричное число,
                        ;преобразовать в 4 бита
    CLC                ;Очистить флаг переноса, ошибки нет
    RET
BAD_DIGIT:
    STC                ;Установить флаг переноса, ошибка
    RET
CONVERT_HEX_DIGIT     ENDP

    PUBLIC     HEX_TO_BYTE
;
; Процедура преобразует два символа в DS:DX из шести, формы
; в байт.
;           DS:DX      Адрес двух символов
; Возвращаются:
;   AL      Байт
;   CF      Устанавливается для ошибок, в другом случае пуст
;
;   Используется:   CONVERT_HEX_DIGIT
;
HEX_TO_BYTE  PROC    NEAR
    PUSH     BX
    PUSH     CX
    MOV      BX,DX      ;Установить адрес в BX для не прямой
;адресации
    MOV      AL,[BX]    ;Взять первую цифру
    CALL     CONVERT_HX_DIGIT
    JC      BAD_HEX     ;Если установлен перенос, то плохая
;шестн. цифра
    MOV      CX,4        ;Умножить на 16
    SHL      AL,CX      ;                               [pause]умножаем на CX ?
    MOV      AH,AL      ;Сохранить копию
    INC      BX          ;Взять вторую цифру
    MOV      AL,[BX]
    CALL     CONVERT_HEX_DIGIT
    JC      BAD_HEX     ;Если установлен перенос, то плохая
;шестн. цифра
    OR       AL,AH      ;Объединить две 4- битовые части

```

```

        CLC                                ;Очистить флаг переноса, ошибок нет
DONE_HEX:
        POP     CX
        POP     BX
        RET
BAD_HEX:
        STC                                ;Установить перенос для ошибок
        JMP     DONE_HEX
HEX_TO_BYTE    ENDP
;
; Это простая версия READ_STRING.
;
;      DS:DX      Адрес строки
;
READ_STRING    PROC    NEAR
        PUSH    AX
        MOV     AH,0Ah                    ;Вызов для буфера клавиатурного ввода
        INT     21h                      ;Вызов функции DOS для буферного ввода
        POP     AX
        RET
READ_STRING    ENDP

PUBLIC    READ_BYTE
;
; Процедура считывает либо ASCII-символ, либо две цифры
;шестнадцатеричного числа. Это только тестовая версия
;READ_BYTE.
;
; Возвращает байт в      AL      Код символа (пока AH = 0)
;                        AH      1, если считан ASCII-символ
;                        0, если символ не считан
;                        -1, если считан спец. ключ
;
;      Используются:  HEX_TO_BYTE_STRING_TO_UPPER,
;                      READ_STRING
;      Считываются:   KEYBOARD_INPUT, etc.
;      Записываются:  KEYBOARD_INPUT, etc.
;
READ_BYTE    PROC    NEAR
        PUSH    DX
        MOV     CHAR_NUM_LIMIT,3          ;Ограничивается число вводимых
;                                           символов (2 символа плюс Enter)
        LEA     DX,KEYBOARD_INPUT
        CALL    READ_STRING
        CMP     NUM_CHARS_READ,1          ;Сколько символов
        JE      ASCII_INPUT              ;Только один, обрабатывается

```

```
;как ASCII-символ
JB     NO_CHARACTERS      ;Нажата только Enter
CALL   STRING_TO_UPPER    ;Нет, преобразовать строку в
;                               заглавные буквы
LEA     DX, CHARS          ;Адрес строки, которую надо
;                               преобразовать
CALL    HEX_TO_BYTE        ;Преобразовать строку из шестн.
;                               в байт
JC      NO_CHARACTERS      ;Ошибка, возврат 'Символы не
;                               считаны'
MOV     AH, 1              ;Считан один символ
DONE_READ:
POP     DX
RET
NO_CHARACTERS:
XOR     AH, AH              ;Установить 'Символы не считаны'
JMP     DONE_READ
ASCII_INPUT:
MOV     AL, CHARS           ;Загрузить считанный символ
MOV     AH, 1              ;Считай один символ
JMP     DONE_READ
READ_BYTE   ENDP
CODE_SEG    ENDS
DATA_SEG    SEGMENT        PUBLIC
KEYBOARD_INPUT LABEL BYTE
CHAR_NUM_LIMIT DB 0        ;Длина буфера ввода
NUM_CHARS_READ DB 0        ; Число считанных символов
CHARS          DB 80 DUP (0) ;Буфер для клавиатурного
;                               ввода
DATA_SEG    ENDS
END
```

Ассемблируйте Kbd_io, скомпонуйте четыре файла Test, Kbd_io, Video_io и Cursor и опробуйте новую версию READ_BYTE.

К этому моменту у нас есть две проблемы с READ_BYTE. Помните специальные функциональные клавиши? Мы не можем считывать их с помощью функции DOS 0Ah. Здесь она не срабатывает. Попробуйте нажать функциональную клавишу, когда вы запустите Test. DOS не вернёт два байта с первым байтом, установленным в нуль, как вы, возможно, ожидали.

Мы не можем считывать расширенные коды с помощью буферного ввода DOS, используя функцию 0Ah. Положительным результатом применения этой функции явилось то, что мы можем использовать клавишу "Backspace" для удаления символов перед тем, как мы нажмём "Enter". Но теперь, так как мы не можем считывать специальные функциональные клавиши, необходимо написать свою собственную процедуру READ_STRING. Заменяем функцию 0Ah, чтобы обеспечить возможность нажатия специальных функциональных клавиш без нажатия "Enter".

Следующая проблема состоит в том, как функция DOS 0Ah обрабатывает символ перевода строки. Нажмите Control-Enter (перевод строки) после того, как вы напечатали один символ, и затем клавишу Backspace. Вы увидите, что оказались на следующей строке, и обратно вернуться не можете. Наша новая версия Kbd_io в следующей главе будет воспринимать перевод строки (Control-Enter) как обычный символ, то есть ввод этого символа не переведёт курсор на следующую строку.

Перед тем как перейти к решению проблем, связанных с READ_BYTE и READ_STRING, давайте напишем процедуру для считывания десятичного числа без знака. Мы не будем использовать эту процедуру в книге, но дисковая версия Dskpatch пользуется ей, так что мы можем, например, сообщить Dskpatch вывести содержимое сектора номер 567.

Десятичный ввод

Напоминаем, что наибольшим десятичным числом без знака, которое мы можем поместить в одно слово, является 65536. Когда мы применяем READ_STRING для считывания строки десятичных цифр, то сообщаем DOS о необходимости считать не более чем шесть символов (пять цифр и символ возврата каретки в конце). Это означает, что READ_DECIMAL сможет считывать и числа от 65536 до 99999, но эти числа не поместятся в одно слово. Мы будем отслеживать такие числа и выдавать в этом случае код ошибки. Код ошибки будет выдаваться также в случае попытки считывания символа, не относящегося к десятичным цифрам.

Перевод пятизначной строки в слово производится с применением умножения, которое мы выполняли в

главе 1: берётся первая (крайняя слева) цифра, умножается на десять, переходим ко второй цифре, умножаем её на десять, и т.д. Используя этот метод, мы можем, например, записать число 49856 как

$$4*10^4 + 9*10^3 + 8*10^2 + 5*10^1 + 6*10^0$$

или, если произвести преобразование:

$$10*(10*(10*(10*4 + 9) + 8) + 5) + 6$$

Конечно, мы должны следить за ошибками, которые могут происходить при умножении, и возвращать флаг переноса в том случае, если такая ошибка происходит. Как мы узнаем о том, что пытаемся считать число, превышающее 65535? Для чисел, превышающих 65535, последняя инструкция MUL даст переполнение регистра DX. При этом устанавливается флаг CF. CF устанавливается всегда, если DX не равен нулю после операции MUL, поэтому для определения ошибки возможно применение инструкции JC ("Jump if Carry set" - перейти, если установлен флаг переноса). Ниже приведён текст процедуры READ_DECIMAL, которая также проверяет каждую цифру на ошибку (находится ли цифра в промежутке от 0 до 9). Поместите эту процедуру в файл KBD_IO.ASM:

Листинг 23.3. Добавьте эту процедуру к KBD_IO.ASM.

```
    PUBLIC    READ_DECIMAL
;
; Эта процедура извлекает содержимое буфера READ_STRING и
; преобразует строку десятичных цифр в слово.
;
; AX    Слово, преобразованное из десятичных цифр
; CF    Установлен при ошибке, сброшен при отсутствии ошибки
;
; Используется:    READ_STRING
; Считывается:    KEYBOARD_INPUT, etc.
; Записывается:    KEYBOARD_INPUT, etc.
;
READ_DECIMAL    PROC    NEAR
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     CHAR_NUM_LIMIT,6    ;Максимум 5 цифр (65535)
```

```
LEA     DX,KEYBOARD_INPUT
CALL    READ_STRING
MOV     CL,NUM_CHARS_READ      ;Число считанных символов
XOR     CH,CH                  ;Сброс в 0 старшего байта счётчика
CMP     CL,0                   ;Возврат ошибки, если нет считанных символов
JLE     BAD_DECIMAL_DIGIT      ;Символы не считаны, ошибка
XOR     AX,AX                  ;Старт с 0
XOR     BX,BX                  ;Старт с начала строки
CONVERT_DIGIT:
MOV     DX,10                  ;Умножить число на 10
MUL     DX                     ;Умножить AX на 10
JC      BAD_DECIMAL_DIGIT      ;CF установлен, если MUL
;переполняет слово
MOV     DL,CHARS[BX]           ;Взять следующую цифру
SUB     DL,'0'                 ;И преобразовать в 4 байта
JC      BAD_DECIMAL_DIGIT      ;Плохая цифра, если не 0
CMP     DL,9                   ;Это плохая цифра?
JC      BAD_DECIMAL_DIGIT      ;Да
ADD     AX,DX                  ;Нет, добавить к числу
INC     BX                     ;Перейти к следующему символу
LOOP    CONVERT_DIGIT          ;Взять следующую цифру
DONE_DECIMAL:
POP     DX
POP     CX
POP     BX
RET
BAD_DECIMAL_DIGIT:
STC                      ;Установить перенос для сообщения об ошибке
JMP     DONE_DECIMAL
READ_DECIMAL    ENDP
```

Для того, чтобы убедиться в том, что эта процедура работает правильно, необходимо протестировать её для всех граничных условий. Вот простая тестовая программа для READ_DECIMAL, использующая метод, применяемый для тестирования READ_BYTE:

Листинг 23.4. Изменения в TEST.ASM.

```
CGROUP   GROUP    CODE_SEG, DATA_SEG
ASSUME    CS:CGROUP, DS:CGROUP

CODE_SEG   SEGMENT PUBLIC
ORG        100h

EXTRN     WRITE_HEX:NEAR, WRITE_DECIMAL:NEAR
EXTRN     WRITE_STRING:NEAR, SEND_CRLF:NEAR
```

```

EXTRN    READ_DECIMAL: NEAR

TEST     PROC     NEAR
    LEA     DX,ENTER_PROMPT
    CALL    WRITE_STRING
    CALL    READ_DECIMAL
    JC      ERROR
    CALL    SEND_CRLF
    LEA     DX, NUMBER_READ_PROMPT
    CALL    WRITE_STRING
    MOV     DX,AX
    CALL    WRITE_DECIMAL
ERROR:
    CALL    SEND_CRLF
    INT     20h
TEST     ENDP
CODE_SEG     ENDS

DATA_SEG     SEGMENT     PUBLIC

ENTER_PROMPT     DB     'Введите десятичное число: ',0
NUMBER_READ_PROMPT     DB     'Число считано: ',0
    PUBLIC  HEADER_LINE_NO, DISK_DRIVE_NO, KEADER_PART_1,
    PUBLIC  HEADER_PART_2,  PROMPT_LINE_NO, CURRENT_SECTOR_NO
HEADER_LINE_NO     DB     0
DISK_DRIVE_NO      DB     0
HEADER_PART_1      DB     0
HEADER_PART_2      DB     0
PROMPT_LINE_NO     DB     0
CURRENT_SECTOR_NO  DB     0

DATA_SEG     ENDS

    END      TEST

```

Опять-таки необходимо скомпоновать пять файлов: Test (предыдущий файл), Kbd_io, Video_io и Cursor. Опробуйте граничные условия, используя как правильные цифры, так и неправильные (такие как А, которая не является десятичной цифрой), и такие числа, как 0, 65535 и 65536.

Итог

Мы вернёмся к двум простым тестовым процедурам позже, когда будем обсуждать способы, применяя которые вы можете писать собственные программы. Мы рассмотрели применение усложнённой версии

TEST.ASM для программы, переводящей числа из шестнадцатеричных в десятичные.

Но теперь мы готовы перейти к следующей главе, в которой напомним улучшенную версию READ_BYTE и READ_STRING.

Глава 24. Улучшенный ввод с клавиатуры

Мы уже отмечали, что представим развитие Dskpatch именно так, как мы его создавали, включая ошибки и не совсем удачно составленные процедуры, некоторые вы уже видели. В этой главе мы напишем новую версию READ_BYTE, и она внесёт в Dskpatch хитрую ошибку, а в следующей главе проведём облаву, чтобы её обнаружить. Попробуйте обнаружить её самостоятельно. (Подсказка: аккуратно проверьте все граничные условия для процедуры READ_BYTE, после того, как она будет присоединена к Dispatch.)

Новая процедура READ_STRING

Наша философия модульного конструирования требует написания коротких процедур, так как каждую процедуру в отдельности легче понять. Новая версия READ_STRING будет примером неудачно составленной процедуры: она слишком длинна. Она должна быть разбита на несколько процедур, но это дело мы оставляем для вас. Эта книга быстро приближается к концу, а нам осталось написать ещё несколько процедур перед тем, как Dskpatch станет полезной программой. Например, мы до сих пор можем редактировать только первую половину сектора и не можем записать этот сектор обратно на диск.

В этой главе мы придадим READ_STRING новую процедуру BACK_SPACE для моделирования функции клавиши Backspace, как это происходит в функции DOS 0Ah. Когда мы нажимаем клавишу Backspace, процедура BACK_SPACE будет удалять последний напечатанный символ, как с экрана, так и из памяти.

На экране BACK_SPACE будет удалять символ, сдвигая курсор на одну позицию влево и печатая там пробел. Эта последовательность действий будет приводить к такому же результату, что и удаление Backspace, предусмотренное DOS.

В буфере BACK_SPACE будет удалять символ, изменяя указатель буфера DS:SI+ BX таким образом, чтобы он указывал на предыдущий младший байт памяти. Другими словами, BACK_SPACE будет просто уменьшать BX:(BX = BX-1). Символ по-прежнему будет находиться в буфере, но наша программа его не увидит. Почему? READ_STRING сообщает нам о том количестве символов, сколько она сосчитала. Если мы попытаемся считать из буфера количество символов, большее этого числа, то мы увидим символы, которые мы уже удалили.

Необходимо быть аккуратными, чтобы не стереть символ при пустом буфере. Помните, что наши данные о строке выглядят так:

CHAR_NUM_LIMIT	DB	0
NUM_CHARS_READ	DB	0
STRING	DB	80 DUP (0)

Буфер строки начинается со второго байта этой области данных или на смещении 2 от её начала. Таким образом, BACK_SPACE не сотрёт символ, если BX равен 2 (то есть указывает на начало буфера строки), потому что буфер в таком состоянии пуст.

Поместите текст процедуры BACK_SPACE в KBD_IO.ASM.

Листинг 24.1. Добавьте эту процедуру к KBD_IO.ASM.

```
    PUBLIC      BACK_SPACE
    EXTRN      WRITE_CHAR:NEAR
;
; Процедура уничтожает символы, по одному при каждом
; нажатии, из буфера и с экрана при не пустом буфере.
; При пустом буфере курсор только перемещается
;
; DS:SI + BX          Символы всё ещё в буфере
;
; Используется:      WRITE_CHAR
;
BACK_SPACE    PROC    NEAR    ;Уничтожает один символ
    PUSH      AX
    PUSH      DX
    CMP       BX,2          ;Буфер пуст?
    JE        END_BS        ;Да, считать следующий символ
    DEC       BX            ;Удалить один символ из буфера
    MOV       AH,2          ;Удалить символ с экрана
```

```
MOV     DL,BS
INT     21h
MOV     DL,20h           ;Записать пробел
CALL    WRITE_CHAR
MOV     DL,BS           ;Вернуться назад
INT     21h
END_BS:
POP     DX
POP     AX
RET
BACK_SPACE    ENDP
```

Перейдём к новой версии READ_STRING. Она будет достаточно длинной. В листинге, который вы увидите, содержится всего одна процедура. Из всех процедур, которые мы написали, READ_STRING является самой длинной и, как мы говорили, слишком большой. Это потому, что она осложнена выполнением возможно большего количества условий.

Для чего READ_STRING делает так много вещей? Мы добавили ещё несколько возможностей. Если вы нажмёте клавишу Escape, то READ_STRING сотрёт буфер строки и удалит все символы с экрана. DOS также удаляет все символы из буфера строки, когда вы нажимаете Escape, но не стирает символы с экрана. Она записывает в конце строки символ обратной косой черты ("\") и переходит на следующую строку. Наша версия READ_STRING будет более многосторонней, чем функция DOS READ_STRING.

READ_STRING использует три специальные клавиши: Backspace, Escape и Enter. Мы могли бы написать ASCII-коды для каждой из этих клавиш в READ_STRING там, где они нам нужны, но вместо этого добавим в начало KBD_IO.ASM несколько макроопределений, которые делают READ_STRING более читаемой. Вот эти определения:

Листинг 24.2. Дополнения к KBD_IO.ASM.

```
CGROUP   GROUP   CODE_SEG,DATA_SEG
        ASSUME   CS:CGROUP, DS:CGROUP
BS       EQU     8       ;Символ "Backspace"
CR       EQU     13      ;Символ возврата каретки
ESC      EQU     27      ;Символ "Escape"

CODE_SEG   SEGMENT      PUBLIC
```

Ниже приведён текст самой процедуры READ_STRING.
Несмотря на её длину, она не очень сложна. Замените старую версию READ_STRING в файле KBD_IO.ASM на эту новую версию:

Листинг 24.3. Новая процедура READ_STRING в KBD_IO.ASM.

```

PUBLIC      READ_STRING
EXTRN      WRITE_CHAR:NEAR
;
;Процедура преобразует функцию почти аналогично DOS 0Ah.
;Отличие в том, что при нажатии функциональной клавиши будет
;возвращаться специальный символ. Esc - уничтожает сделанный
;ввод.
;
;  DS:DX Адрес буфера клавиатуры. Первый байт должен
;          содержать максимальное число считываемых символов
;          (плюс один для возврата). Второй байт будет
;          использоваться этой процедурой для возврата
;          действительного числа считанных символов.
;          0   нет считанных символов
;          -1  считан специальный символ, в другом случае число
;          считанных символов (без ENTER)
;
;          Используется:      BACK_SPACE, WRITE_CHAR
;
READ_STRING  PROC      NEAR
    PUSH     AX
    PUSH     BX
    PUSH     SI
    MOV      SI,DX      ;SI применяется в качестве регистра индекса
START_OVER:
    MOV      BX,2        ;BX для ввода без проверки
    MOV      AH,7        ;Вызов для ввода без проверки
    INT      21h         ;для CTRL-BREAK и без вывода
    OR       AL,AL       ;Символ расширенный ASCII?
    JZ       EXTENDED    ;Да, считать расширенный символ
NOT_EXTENDED:
    ;Расширенный символ ошибка пока буфер пуст
    CMP      AL,CR       ;Это возврат каретки?
    JE       END_INPUT   ;Да, мы сделали ввод
    CMP      AL,BS       ;Это символ пробела?
    JNE      NOT_BS      ;Нет оператора
    CALL     BACK_SPACE   ;Да, уничтожить символ
    CMP      BL,2        ;Буфер пуст?

```


___Новая процедура READ_STRING___

```
    JE      START_OVER      ;Да, можем считать расширенный
;                               символ снова
    JMP     SHORT READ_NEXT_CHAR ;Нет, продолжаем
;                               нормальное считывание
NOT_BS:
    CMP     AL,ESC           ;Это очистка буфера через ESC?
    JE      PURGE_BUFFER     ;Да, очистить буфер
    CMP     BL,[SI]          ;Проверить, полон ли буфер
    JA      BUFFER_FULL      ;Буфер попон
    MOV     [SI + BX],AL      ;Записать ещё символ в буфер
    INC     BX               ;Отметить следующий свободный символ в буфере
    PUSH    DX
    MOV     DL,AL            ;Отразить символ на экран
    CALL    WRITE_CHAR
    POP     DX
READ_NEXT_CHAR:
    MOV     AH,7
    INT     21h
    OR      AL,AL            ;Все расширенные ASCII-символы
;                               неправильные, если буфер не пуст
    JNE     NOT_EXTENDED     ;Символ правильный
    MOV     AH,7
    INT     21h              ;Вывести расширенный символ
; Сообщает о наличие ошибки звуковым сигналом, посылая
; символ chr$(7)
;
SIGNAL_ERROR:
    PUSH    DX
    MOV     DL,7             ;Включить сигнал chr$(7)
    MOV     AH,2
    INT     21h
    POP     DX
    JMP     SHORT READ_NEXT_CHAR ;Считать следующий символ
;
; Очистить буфер строки и удалить все символы, выведенные на
; экран
;
PURGE_BUFFER:
    PUSH    CX
    MOV     CL,[SI]          ;Очистить число считанных символов
    XOR     CH,CH
PURGE_LOOP:
    CALL    BACK_SPACE
    LOOP    PURGE_LOOP
    POP     CX
    JMP     START_OVER      ;Сейчас, пока буфер пуст, можно считать
```

```
;                                     расширенный ASCII-символ
;
;Буфер был полон, поэтому не можем считать символ.
;Сигнализировать
;   пользователю звуковым сигналом о полном
;буфере.
;
BUFFER_FULL:
    JMP     SHORT SIGNAL_ERROR        ;Если буфер полон - сигнал
;
;Считать расширенный ASCII-код и поместить его в буфер как
;символ, затем вернуть -1 как номер считанного символа.
;
EXTENDED:                                ;Считать расширенный ASCII-код
    MOV     AH,7
    INT     21h
    MOV     [SI+2],AL      ;Поместить символ в буфер
    MOV     BL,0FFh        ;Число считанных символов = -1 для специальных
    JMP     SHORT END_STRING
;
; Сохранить число считанных символов и вернуть.
;
END_INPUT:                                ;Сделано с вводом
    SUB     BL,2           ;Счётчик считанных символов
END_STRING:
    MOV     [SI+1],BL      ;Вернуть число считанных символов
    POP     SI
    POP     BX
    POP     AX
    RET
READ_STRING    ENDP
```

Двигаясь по тексту процедуры, мы видим, что READ_STRING сначала проверяет, нажаты или нет специальные функциональные клавиши. Она позволяет это сделать только в том случае, если строка пуста. Например, при нажатии F1 после нажатия клавиши "A" READ_STRING проигнорирует нажатие F1 и выдаст звуковой сигнал, означающий, что функциональная клавиша нажата несвоевременно. Мы можем, однако, нажать Escape, а затем F1, так как клавиша Escape заставляет READ_STRING очистить буфер строки.

Если READ_KEY считывает символ возврата каретки, то она помещает число считанных символов во второй байт области строки и возвращает управление вызвавшей её процедуре. Новая версия READ_BYTE

просматривает этот байт, чтобы увидеть, сколько символов в действительности считала READ_STRING.

Затем READ_STRING проверяет, не нажата ли Backspace. Если да, то вызывается BACK_SPACE, стирающая один символ. Если буфер строки полностью очищается (BX равен 2, началу буфера строки), то READ_STRING возвращается к началу процедуры, где она может считывать специальные клавиши. Если буфер заполнен, то она считывает следующий символ.

Наконец, READ_STRING проверяет наличие символа ESC. BACK_SPACE стирает символы только тогда, когда в буфере ещё есть символы, поэтому очистить буфер строки можно вызовом процедуры BACK_SPACE CHAR_NUM_LIMIT раз, потому что READ_STRING не может считать более чем CHAR_NUM_LIMIT символов. Все оставшиеся символы, хранящиеся в буфере строки, выводятся на экран с помощью WRITE_CHAR.

В последней главе мы изменили процедуру READ_BYTE так, что она не могла считывать специальные функциональные клавиши. Необходимо добавить всего несколько строк для того, чтобы READ_BYTE смогла работать с новой версией процедуры READ_STRING, которая уже может считывать специальные функциональные клавиши. Вот изменения, которые надо внести в процедуру READ_BYTE в файле KBD_IO.ASM:

Листинг 24.4. Изменения в READ_STRING в файле KBD_IO.ASM.

```
PUBLIC      READ_BYTE
;
; Процедура считывает ASCII-символ шестнадцатеричного числа.
; Возвращает байт в AL код символа (пока AH = 0)
;   AH      1, если считан ASCII-символ или шестн. число
;           0, если символ не считан
;           -1, если считана спец. клавиша
;   Используются:
;HEX_TO_BYTE_STRING TO UPPER,READ_STRING
;   Читаются:      KEYBOARD_INPUT, и т.д.
;   Записываются: KEYBOARD_INPUT, и т.д.
;
READ_BYTE   PROC      NEAR
    PUSH    DX
    MOV     CHAR_NUM_LIMIT,3      ; Только два символа (плюс Enter)
    LEA     DX,KEYBOARD_INPUT
    CALL    READ_STRING
    CMP     NUM_CHARS_READ,1      ;Много символов
```

```
JE      ASCII_INPUT      ;Только один ASCII-символ
JB      NO_CHARACTERS     ;Нажата только Enter
CMP     BYTE PTR NUM_CHARS_READ,0FFh      ;Функциональная
;                                           клавиша?

JE      SPECIAL_KEY       ;Да
CALL    STRING_TO_UPPER   ;Нет, преобразовать в заглавные
LEA     DX, CHARS          ;Адрес строки для преобразования
CALL    HEX_TO_BYTE       ;Преобразовать строку из
;                           шестнадц. в байт

JC      NO_CHARACTERS     ;Ошибка, вернуть 'символ не считан'
MOV     AH,1              ;Сигнал, считан один символ
DONE_READ:
POP     DX
RET

NO_CHARACTERS:
XOR     AH,AH             ;Установить 'символ не считан'
JMP     DONE_READ

ASCII_INPUT:
MOV     AL,CHARS          ;Загрузить считанный символ
MOV     AH,1              ;Сигнал, считан один символ
JMP     DONE_READ

SPECIAL_KEY:
MOV     AL,CHARS[0]       ;Вернуть скан-код
MOV     AH,0FFh           ;Спец. клавиша, -1
JMP     DONE_READ

READ_BYTE ENDP
```

Dskpatch с новой версией READ_BYTE и READ_STRING становится гораздо удобнее в работе. Но в нём, как мы предупреждали ранее, содержится ошибка. Чтобы попытаться найти её, запустите Dskpatch и опробуйте все граничные условия для READ_BYTE и HEX_TO_BYTE.

Глава 25. В поисках ошибок

Если в новой версии Dskpatch попытаться ввести "ag", не являющееся шестнадцатеричным числом, то после нажатия клавиши Enter ничего не произойдет. Никакого нарушения нет, так как строка "ag" не является шестнадцатеричным числом, и нет ничего неправильного в том, что Dskpatch игнорирует её. Но программа должна, по крайней мере, удалить такую строку с экрана.

Эта ошибка принадлежит к погрешностям того типа, которые мы можем обнаружить, только проверив граничные условия программы. Ошибка эта относится не к READ_BYTE, несмотря на то, что она появилась после того, как мы переписали эту процедуру. Скорее проблема состоит в том, как мы написали DISPATCHER и EDIT_BYTE.

Процедура EDIT_BYTE написана таким образом, что она вызывает WRITE_PROMPT_STRING для перепечатки строки приглашения редактора и стирания остатка строки. Это удаляет все введенные до этого символы. Но если мы напечатаем такую строку, как "ag", READ_BYTE сообщит, что она считала строку нулевой длины, и DISPATCH, естественно, не будет вызывать EDIT_BYTE. Каково же решение?

Исправление DISPATCHER

Существуют два способа решения этой проблемы. Лучшее из них - переписать Dskpatch таким образом, чтобы он более полно отвечал требованиям модульного конструирования, и затем воссоздать DISPATCHER заново. Но мы не будем этого делать. Помните: любая программа никогда не достигнет абсолютного совершенства, поэтому необходимо где-то остановиться. Мы добавим к DISPATCHER небольшое исправление для того, чтобы он перепечатывал строку приглашения в том случае, если READ_BYTE считывает строку нулевой длины.

Ниже приведены изменения, которые надо внести в DISPATCHER (в файле DISPATCH.ASM), чтобы исправить ошибку:

Листинг 25.1. Изменения в процедуре DISPATCHER в файле DISPATCH.ASM.

```
        PUBLIC      DISPATCHER
        EXTRN       READ_BYTE:NEAR, EDIT_BYTE:NEAR
        EXTRN       WRITE_PROMPT_LINE:NEAR
DATA SEG      SEGMENT      PUBLIC
        EXTRN       EDITOR_PROMPT:BYTE
DATA_SEG      ENDS
;
; Это центральный диспетчер. В режиме обычного
; редактирования и
; просмотра эта процедура считывает символ
; с клавиатуры и, если он является командой (например, клавиша
; управления курсором), DISPATCHER вызывает соответствующую
```

```
;процедуру для выполнения действия. Такое управление сделано
;для командных клавиш, список которых представлен в таблице
;DISPATCH_TABLE с именами процедур и их адресами
;расположения в памяти. Если символ не является командой, то
;он помещается непосредственно в буфер сектора - включается
;режим редактирования.
;Используются: READ_BYTE, EDIT_BYTE, WRITE_PROMPT_LINE
;          Считываются: EDITOR_PROMPT
;
DISPATCHER      PROC      NEAR
    PUSH        AX
    PUSH        BX
    PUSH        DX
DISPATCH_LOOP:
    CALL        READ_BYTE      ;Считать символ в AX
    OR          AH,AH          ;AH =0 если символа нет
;                               AH = -1 для расширенного кода.
    JZ          NO_CHARS.READ  ;Символ не считан, считать ещё раз
    JS          SPECIAL_KEY    ;Считан расширенный код
    MOV         DL,AL
    CALL        EDIT_BYTE      ;Считан обычный символ,
;                               редактировать байт
    JMP         DISPATCH_LOOP  ;Считать другой символ
SPECIAL_KEY:
    CMP         AL,68           ;F10-Выход?
    JE          END_DISPATCH   ;Да, выйти
;                               Применить BX для просмотра
    LEA         BX,DISPATCH_TABLE
SPECIAL_LOOP:
    CMP         BYTE PTR [BX],0 ;Конец таблицы?
    JE          NOT_IN_TABLE    ;Да, команды в таблице нет
    CMP         AL,[BX]         ;Это вход таблицы?
    JE          DISPATCH       ;Да, управлять
    ADD         BX,3            ;Нет, ещё раз
    JMP         SPECIAL_LOOP    ;Проверить следующий вход
DISPATCH:
    INC         BX              ;Отметить для адреса процедуры
    CALL        WORD PTR[BX]    ;Вызвать процедуру
    JMP         DISPATCH_LOOP   ;Ждать другой клавиши
NOT_IN_TABLE:
;                               Ничего не делать, только считать
;                               следующий символ
    JMP         DISPATCH_LOOP
NO_CHARS_READ:
    LEA         DX,EDITOR_PROMPT
    CALL        WRITE_PROMPT_LINE ;Очистить любой
```

```
                                ; напечатанный неправильный
                                ; символ
                                ; Повторить
    JMP     DISPATCH_LOOP
END_DISPATCH:
    POP     DX
    POP     BX
    POP     AX
    RET
DISPATCHER     ENDP
```

Это исправление ошибки не создаёт больших проблем, но оно делает DISPATCHER менее элегантным. Элегантность - это достоинство, за которое надо бороться. Элегантность и ясность часто идут рука об руку, и наши правила модульного конструирования позволяют создавать более элегантные программы.

Итог

DISPATCHER элегантен, так как он является простым решением проблемы. Вместо того, чтобы использовать множество сравнений для каждого специального символа, который мы можем напечатать, мы создали таблицу поиска. Это упростило DISPATCHER, и следовательно сделало его более надёжным по сравнению с программой, содержащей различные инструкции для каждого возникающего состояния. Добавив незначительное исправление, мы усложнили DISPATCHER, не намного в данном случае, но ошибки, возможные в дальнейшем, могут заставить действительно усложнить процедуру.

Если вы чувствуете, что исправления, добавляемые в программу, делают её излишне сложной, то попробуйте переписать и упростить процедуры. Проверяйте граничные условия и до, и после добавления изменённой процедуры к основной программе. Это поможет локализовать ошибки и сэкономит массу времени и труда по их устранению.

Тестирование процедур на работоспособность в области граничных условий также важно, как и следование правилам модульного конструирования. Эти чисто технические приёмы позволяют создавать более совершенные и лучше читаемые программы. В следующей главе мы познакомимся с ещё одним методом отладки программ.

Глава 26. Запись модифицированных секторов на диск

Программа Dskpatch уже почти готова и использованию. В этой главе мы создадим процедуру записи изменённого сектора обратно на диск, а в следующей главе напишем процедуру вывода на экран второй половины сектора. Dskpatch ещё не закончен окончательно, но в этой книге его рассмотрение будет завершено. Более совершенная версия Dskpatch, поставляемая на отдельном диске, обладает множеством дополнений.

Запись на диск

Запись модифицированного сектора на диск, если она делается неумышленно, может привести к неприятностям. Все функции Dskpatch до этого зависели от функциональных клавиш F1, F2, F10 и клавиш курсора. Но любая из этих клавиш может быть нажата случайно. Вероятность случайного одновременного нажатия двух клавиш существенно ниже. Поэтому для записи сектора на диск будет применяться комбинация Shift + F5. Внесите в DISPATCH.ASM, добавив в таблицу процедуру WRITE_SECTOR:

Листинг 26.1. Изменения в DISPATCH.ASM.

```
DATA_SEG SEGMENT PUBLIC
    EXTRN NEXT_SECTOR:NEAR ;В файле DISK_IO.ASM
    EXTRN PREVIOUS_SECTOR:NEAR ;В файле DISK_IO.ASM
    EXTRN PHANTOM_UP:NEAR, PHANTOM_DOWN:NEAR ;В файле
; PHANTOM.ASM
EXTRN PHANTOM_LEFT:NEAR, PHANTOM_RIGHT:NEAR
EXTRN WRITE_SECTOR:NEAR ;В файле DISK_IO.ASM
; Эта таблица содержит разрешенные расширенные ASCII-коды
; и адреса процедур, выполняемых при нажатии соответствующих
; клавиш.
; Формат таблицы:
; DB 72 ;Расширенный код перемещения курсора вверх
; OFFSET CGROUP:PHANTOM_UP
;
DISPATCH_TABLE LABEL BYTE
    DB 59 ;F1
```


___Запись на диск___

```
DW    OFFSET CGROUP:PREVIOUS_SECTOR
DB    60
DW    OFFSET CGROUP:NEXT_SECTOR
DB    72
DW    OFFSET CGROUP:PHANTOM_UP
DB    80
DW    OFFSET CGROUP:PHANTOM_DOWN
DB    75
DW    OFFSET CGROUP:PHANTOM_LEFT
DB    77
DW    OFFSET CGROUP:PHANTOM_RIGHT
DB    88
DW    OFFSET CGROUP:WRITE_SECTOR
DB    0
DATA_SEG    ENDS
```

WRITE_SECTOR почти идентична READ_SECTOR. Единственное отличие состоит в том, что мы хотим записать, а не считать сектор. В то время как INT 25h сообщает DOS о необходимости считать сектор, её функция-спутник, INT 26h, сообщает о записи сектора.

Вот текст процедуры WRITE_SECTOR, поместите его в файл DISK_IO.ASM:

Листинг 26.2. Добавьте эту процедуру к файлу DISK_IO.ASM.

```
PUBLIC WRITE_SECTOR
;      Процедура записывает сектор обратно на диск
;      Считываются: DISK_DRIVE_NO, CURRENT_SECTOR_NO, SECTOR
;
WRITE_SECTOR    PROC    NEAR
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     AL,DISK_DRIVE_NO    ;Номер дисководов
    MOV     CX,1                ;Записать один сектор
    MOV     DX,CURRENT_SECTOR_NO ;Логический сектор
    LEA     BX,SECTOR
    INT     26h                ;Записать сектор на диск
    POPF                    ;Сбросить флаг
    POP     DX
    POP     CX
    POP     BX
    POP     AX
    RET
WRITE_SECTOR    ENDP
```

Ассемблируйте Dispatch и Disk_io, но не пытайтесь сразу же опробовать функцию Dskpatch для записи. Найдите диск, содержимое которого вам не очень нужно, и поместите его в дисковод "A", а диск с программой - в другой дисковод, например "B". Запустите Dskpatch с дисковода "B" (или того дисковода, который выбрали) так, чтобы Dskpatch считал первый сектор с диска в дисковде "A". На всякий случай проверьте всё ещё раз, потому что данные на диске в дисковде "A" могут быть потеряны.

Измените один байт в изображении сектора и запомните это изменение. Затем нажмите Shift + F5. Вы увидите, как на дисковде загорится красная лампочка: происходит запись модифицированного сектора обратно на диск.

Затем нажмите F2, чтобы считать следующий сектор (сектор 1), и затем F1, чтобы опять считать предыдущий сектор (тот, который вы изменяли, номер 0). Вы должны опять увидеть изменённый сектор. Восстановите значение изменённого байта и запишите его обратно на дисковод "A", чтобы восстановить предыдущее состояние данных.

Некоторые приёмы отладки

Что произойдет, если в программе допущена некоторая ошибка? Dskpatch достаточно велик по объёму, поэтому нас ожидают определённые проблемы при использовании Debug для поиска ошибок. Кроме того, Dskpatch состоит из нескольких файлов, которые мы компонуем при формировании DSKPATCH.COM. Как можно выделить в этой большой программе одну процедуру, не трассируя всю программу? В этой главе рассмотрены два способа поиска процедур: используя карту загрузки, получаемую с помощью LINK, или применяя вместо DEBUG программу фирмы Microsoft SYMDEB.

Когда Dskpatch создавался впервые при написании книги, то после добавления WRITE_SECTOR что-то пошло не так - при нажатии клавиш Shift + F5 наш компьютер "зависал". Ошибок в самой процедуре WRITE_SECTOR и в тех изменениях, которые мы внесли в DISPATCH_TABLE, после самой тщательной проверки найдено не было. Казалось, что всё правильно.

Ошибка была найдена в неправильном определении в диспетчере. Заключалась она в том, что была неправильно определена точка входа в DISPATCH_TABLE для процедуры WRITE_SECTOR. Опечатка "DW" вместо "DB", приводила к тому, что адрес WRITE_SECTOR хранился в памяти со сдвигом на один байт.

Вы можете увидеть эту ошибку в приведённом ниже тексте, выделенную наклонным шрифтом:

```
DISPATCH_TABLE LABEL BYTE
.
.
.
DB 77 ;Курсор вправо
DW OFFSET CGROUP: PHANTOM_RIGHT
DW 88 ;Shift F5### (выделить курсивом)
DW OFFSET CGROUP:WRITE_SECTOR
DB 0 ;Конец таблицы
DATA_SEG ENDS
```

Используйте эту ошибку как упражнение по отладке: внесите её в ваш файл DISPATCH.ASM и следуйте инструкциям, изложенным в следующем разделе.

Построение карты загрузки

Сейчас мы узнаем, как использовать LINK для того, чтобы построить карту Dskpatch. Эта карта поможет найти в памяти процедуры и переменные.

Команда LINK стала очень громоздкой:

```
LINK DSKPATCH disk_io DISP_SEC Video_io CURSOR DISPATCH
KBD_IO PHANTOM EDITOR;
```

а мы хотим добавить к ней ещё кое-что. Для того чтобы избежать дальнейшего усложнения командной строки, воспользуемся способностью LINK применять файл автоматического ответа, содержащий всю необходимую информацию. Имея этот файл, названный "LINKINFO", мы можем просто напечатать:

```
LINK @LINKINFO
```

и LINK считает необходимую информацию из этого файла.

С именами файлов, которые компоновали до этого, файл LINKINFO выглядит так:

```
DSKPATCH DISK_IO DISP_SEC Video_io CURSOR +  
DSKPATCH KBD_IO PHANTOM EDITOR
```

Плюс (" + ") в конце первой строки сообщает LINK о необходимости продолжения считывания имен файлов из следующей строки.

LINK позволяет также создать карту процедур и переменных программы.

```
DSKPATCH DISK_IO DISP_SEC Video_io CURSOR +  
DSKPATCH KBD_IO PHANTOM EDITOR  
DSKPATCH  
DSKPATCH /MAP;
```

Последние две строки являются новыми параметрами. Первая dskpatch сообщает LINK о том, что .EXE, файл должен называться DSKPATCH.EXE, а второй - о необходимости создать файл листинга DSKPATCH.MAP или карту загрузки. Переключатель /map обеспечивает вывод списка имён всех процедур и переменных, объявленных как PUBLIC.

Создайте мэп-файл, ещё раз производя компоновку Dskpatch с файлом информации для LINK. Мэп-файл, построенный компоновщиком, будет иметь около 120 строк в длину. Это слишком много для воспроизведения его здесь полностью, поэтому мы приведём здесь только части, представляющие особенный интерес. Частичный листинг мэп-файла DSKPATCH.MAP приведён ниже:

```
Warning: no stack segment
```

Start	Stop	Length	Name	Class
00000H	007E5H	007E6H	CODE_SEG	
007F0H	0291FH	02130H	DATA_SEG	
Origin	Group			
0000:0	CGROUP			
Address	Publics by Name			
0000:0677	BACK_SPACE			
9*				

```
0000:048F      CLEAR_SCREEN
0000:04D1      CLEAR_TO_END_OF_LINE
0000:07F2      CURRENT_SECTOR_NO
0000:04B1      CURSOR_RIGHT
0000:07F4      DISK_DRIVE_NO
0000:04F0      DISPATCHER
0000:01F3      DISP_HALF_SECTOR
.
.
.
0000:0370      WRITE_HEX_DIGIT
0000:03DB      WRITE_PATTERN
0000:06FE      WRITE_PHANTOM
0000:0440      WRITE_PROMPT_LINE
0000:013A      WRITE_SECTOR
0000:0428      WRITE_STRING

Address      Publics by Value

0000:0120      READ_SECTOR
0000:013A      WRITE_SECTOR
0000:0154      PREVIOS_SECTOR
0000:0174      NEXT_SECTOR
0000:0190      INIT_SEC_DISP
0000:01F3      DISP_HALF_SECTOR
.
.
.
0000:07F5      LINES_BEFORE_SECTOP
0000:07F6      HEADER_LINE_NO
0000:07F7      HEADER_PART_1
0000:07FD      HEADER_PART_2
0000:080E      PROMPT_LINE_NO
0000:080F      EDITOR_PROMPT
0000:0844      SECTOR
0000:2912      PHANTOM_CURSOR_X
0000:2913      PHANTOM_CURSOR_Y

Program entry point at    0000:0100
```

Карта загрузки (называется так потому, что предоставляет информацию о расположении загруженной программы в памяти) состоит из трёх основных частей. Первая содержит список сегментов программы. Dskpatch имеет всего два сегмента CODE_SEG и DATA_SEG, объединённые вместе.

Следующая часть карты загрузки показывает в алфавитном порядке процедуры и переменные PUBLIC. LINK включает в список процедуры и переменные, объявленные как PUBLIC, то есть те, которые могут использоваться всеми файлами программы. Если вы отлаживаете длинную программу, то можете объявить все переменные и процедуры как PUBLIC -это позволит отыскать их на этой карте.

Последняя часть карты также содержит все процедуры и переменные, но теперь в том порядке, в котором они появляются в памяти.

Оба этих списка включают адреса для каждой PUBLIC процедуры и переменной. Если вы проверите этот список, то найдете, что процедура DISPATCHER начинается с адреса 4F0h. Мы сейчас используем этот адрес для того, чтобы протрассировать ошибку в Dskpatch.

Трассировка ошибок

При попытке запуска версии Dskpatch с имеющейся ошибкой, вы увидите, что все работает нормально, за исключением комбинации клавиш Shift + F5, которая на нашей машине заставляла Dskpatch зависнуть. Вы, пожалуй, не захотите пробовать Shift + F5, так как никто не может предсказать, что произойдет с вашим компьютером.

Так как всё, кроме Shift + F5, работало (и работает до сих пор), первым предположением было то, что допущена ошибка в процедуре WRITE_SECTOR. Поиск ошибки можно было бы начать, как и раньше, с трассировки WRITE_SECTOR, но мы возьмём немного другой курс.

Мы знаем, что DISPATCHER работает правильно (клавиши F1, F2 и F10 работают нормально) и поэтому он может быть использован в качестве стартовой точки поиска ошибки в Dskpatch.

Если вы посмотрите на листинг DISPATCHER (в главе 25), то увидите, что инструкция

```
CALL    WORD    PTR    [BX]
```

является ядром DISPATCHER, так как она вызывает все остальные процедуры. В частности, эта инструкция CALL будет вызывать процедуру WRITE_SECTOR при нажатии Shift + F5. начнём поиски здесь.

Мы будем использовать Debug для запуска Dskpatch с точкой останова, установленной на эту инструкцию. Конечно, это означает, что нам нужен адрес этой инструкции, и мы можем найти его, разассемблировав DISPATCHER с адреса 4F0h. После "U 4F0" вы должны увидеть команду CALL:

```
      .
      .
      .
2C14:0517  EBF2    JMP     050B
2C14:0519    43    INC     BX
2C14:051A  FF17    CALL    [BX]
2C14:051C  EBD5    JMP     04F3
      .
      .
      .
```

Теперь, когда мы знаем, что инструкция CALL находится по адресу 51Ah, можем установить точку останова на этот адрес и затем пройти в пошаговом режиме через WRITE_SECTOR.

Во-первых, используйте команду "G 51A" для выполнения Dskpatch до адреса этой инструкции. Вы увидите, как Dskpatch начнет работу и затем остановится, ожидая ввода команды. Нажмите Shift + F5, ведь именно эта комбинация является причиной проблем. Вы увидите следующее:

```
-G51A
AX=FF58 BX=28A3 CX=2820 DX=080F SP=FFF6 BP=419A SI=03CC DI=0001
DS=2C14 ES=2C14 SS=2C14 CS=2C14 IP=051A NV UP DI PL NZ NA PE NC
2C14:051A  FF17    CALL [BX]    DS:28A3=3A00
```

В этой точке регистр BX указывает на то слово, которое должно содержать адрес WRITE-SECTOR. Посмотрим, так ли это:

```
-D 28A3 L2
2C14:28A0  003A
-
```

Другими словами, мы пытаемся вызвать процедуру, размещённую по адресу 3A00h (помните, что младший байт высвечивается первым). Но если мы

взглянем на карту памяти, то увидим, что WRITE_SECTOR должна находиться по адресу 13Ah. Из этой карты загрузки мы можем также понять, что у нас нет НИКАКОЙ процедуры по адресу 3A00h. Адрес абсолютно неверен!

В этой настоящей охоте за ошибкой, после того, как мы определили, что адрес неверен, найти ошибку не составило большого труда. Мы знали, что DISPATCHER и DISPATCH_TABLE не содержат ошибки, так как все остальные клавиши работают, поэтому мы внимательно рассмотрели данные для Shift + F5 и нашли DW там, где должно было быть DB. Карта загрузки существенно упрощает процесс отладки, предоставляя необходимую информацию об адресах расположения процедур. Теперь давайте рассмотрим Symdeb.

Symdeb

Symdeb ("Symbolic Debugging" - символьная отладка) - это программа, которую фирма Microsoft включает в пакет макроассемблера, начиная с версии 3.00 и выше. Как вы увидите в этом разделе, Symdeb довольно полезен, так что если у вас его нет, вам, возможно, стоит приобрести более позднюю версию макроассемблера.

И Debug, и Symdeb написаны фирмой Microsoft, поэтому Symdeb повторяет многие, если не все команды Debug. Он также включает большое число очень полезных команд, не представленных в Debug, причём некоторые обладают совершенно замечательными возможностями. Две из них: символьную отладку и экранный свопинг мы будем использовать в этой главе.

Символьная отладка

Символьная отладка, которая и дала название Symdeb, позволяет видеть в разассемблированном листинге не адреса, а имена процедур и переменных. Например, если мы используем Debug для того, чтобы разассемблировать первую строку Dskpatch, то увидим:

```
2C14:0100E  88C03      CALL 048F
```

Если же мы используем Symdeb, то мы увидим следующее:

3245:0100 E88C03 CALL CLEAR_SCREEN

Какую из строк легче прочесть? С нашей точки зрения, выбор однозначен

Экранный свопинг

Вторая возможность, экранный свопинг, пригодится для отладки Dskpatch. Dskpatch выводит информацию в различные части экрана. В последнем разделе, когда мы применяли Debug, он начинал выводить новую информацию поверх предыдущей, и поэтому можно было потерять изображение, ранее выведенное Dskpatch.

Symdeb, однако, поддерживает две отдельные области экрана: одну для Dskpatch и одну для себя. Если в данный момент активен Dskpatch, мы видим его экран, если активен Debug, мы видим его экран. Мы лучше усвоим идею экранного свопинга, если мы поработаем с примерами.

Перед тем, как мы сможем использовать возможность Symdeb производить символьную отладку, нам надо с помощью программы, называющейся Mapsym, создать символьный файл. Mapsym берёт .MAP-файл, созданный ранее в этой главе, и переводит его в символьный файл:

```
A:\>MAPSYM DSKPATCH
Microsoft (R) Symbol File Generator Version 4.00
Copyright (C) Microsoft Corp 1984, 1985. All rights reserved.
```

В данном случае программа Mapsym создала символьный файл, называющийся DSKPATCH.SYM. Затем мы запускаем Symdeb, указав в командной строке и символьный, и .COM-файл:

```
A:\>SYMDEB /S DSKPATCH.SYM DSKPATCH.COM
Microsoft (R) Symbolic Debug Utility Version 4.00
Copyright (C) Microsoft Corp 1984, 1985. All rights reserved.

Processor is [8086]
-
```

Переключатель /S в командной строке сообщает Symdeb о необходимости включения экранного свопинга. Программа не использует эту возможность по умолчанию, так как свопинг значительно влияет на быстродействие Symdeb.

Перед тем как повторить предыдущую отладку, посмотрим на начало Dskpatch:

```
-U
330E:0100    E88C03    CALL    CLEAR_SCREEN
330E:0103    E8F402    CALL    WRITE_HEADER
330E:0106    E81700    CALL    READ_SECTOR
330E:0109    E88400    CALL    INIT_SEC_DISP
330E:010C    8D160F08  LEA     DX, [EDITOR_PROMPT]
330E:0110    E82D03    CALL    WRITE_PROMPT_LINE
330E:0113    E8DA03    CALL    DISPATCHER
330E:0116    CD20     INT     20
-
```

Вы видите, как удобно, что Symdeb высвечивает все имена, а не адреса.

Когда мы последний раз разасемблировали DISPATCHER для того, чтобы найти адрес инструкции "CALL WORD [BX]", мы сначала просматривали тар-файл, чтобы найти адрес процедуры, а затем печатали "U 4F0", чтобы разасемблировать её. С приходом Symdeb жизнь стала значительно легче: мы просто печатаем "U DISPATCHER", чтобы разасемблировать нужную процедуру.

```
-U DISPATCHER
CGROUP:DISPATCHER:
330E:04F0     50          PUSH    AX
330E:04F1     53          PUSH    BX
330E:04F2     52          PUSH    DX
330E:04F3    E80401    CALL    READ_BYTE
330E:04F6    0AE4     OR      AH, AH
330E:04F8    7426     JZ      DISPATCHER + 30 (0520)
330E:04FA    7807     JS      DISPATCHER + 13 (0503)
-
```

После ещё двух команд "U" мы находим инструкцию CALL:

```
330E:0514    83C3003    ADD     BX, + 03
330E:0517    EBF2     JMP     DISPATCHER + 1B (050B)
330E:0519     43        INC     BX
330E:051A    FF17     CALL    [BX]
330E:051C    EBD5     JMP     DISPATCHER + 03 (04F3)
-
```

Напечатайте "G 51A", как и раньше, и затем нажмите Shift + F5. Если вы используете Symdeb, то увидите, что Dskpatch обновит экран. После того как

нажмите Shift + F5, вы вернётесь в Symdeb. Однако в этот момент вы уже не увидите экрана Dskpatch, так как Symdeb опять поменяет экраны. Чтобы вернуться обратно к экрану Dskpatch, нажмите клавишу обратной косой черты ("\") и затем "Enter".

Возможно, вы заметили в применении Symdeb одну тонкость. Если мы посмотрим на разассемблированные листинги, то увидим, что инструкции выглядят следующим образом:

```
330E:051C  EBD5  JMP  DISPATCHER + 03 (04F3)
```

а не таким:

```
330E:051C  EBD5  JMP  DISPATCHER_LOOP
```

Почему Symdeb не использовал метку DISPATCH_LOOP? Мы не объявили, что метки в этой процедуре являются PUBLIC. Если мы вернёмся назад и объявим PUBLIC для всех меток DISPATCHER, то увидим эти метки в разассемблированном листинге. (Если вы сделаете это, то не забудьте воссоздать с помощью Mapsym символный файл).

Итог

Итак, наша дискуссия о методах отладки закончена. Вам осталось прочесть всего три главы этой книги. В следующей главе мы добавим процедуры скроллинга (прокрутки) экрана между двумя половинами сектора. Затем в последних двух главах мы подробнее рассмотрим разницу между .COM и .EXE-файлами, особенности оператора ASSUME и вопросы работы с сегментами.

Кстати, не забудьте удалить ту ошибку, которую мы поместили в DISPATCH_TABLE.

Глава 27. Другая половина сектора

В идеале Dskpatch должен вести себя так же, как текстовый редактор. Если при работе в текстовом редакторе курсор достиг нижней строки текста, выведенного на экран, то при попытке перемещения курсора вниз текст перемещается на одну строку вверх. Версия Dskpatch, содержащаяся на диске,

поставляемом отдельно, обладает этой возможностью, но здесь мы не будем вдаваться в такие сложности.

В этой главе мы добавим упрощённые версии двух процедур SCROLL_UP и SCROLL_DOWN, прокручивающих изображение сектора на экране. В дисковой версии Dskpatch, SCROLL_UP и SCROLL_DOWN могут перемещать (прокручивать) изображение на любое количество строк, от одной до шестнадцати. Представленные здесь версии этих процедур прокручивают изображение половины сектора целиком, то есть на экране может изображаться либо первая, либо вторая половина изображения сектора.

Скроллинг половины сектора

Старые версии PHANTOM_UP и PHANTOM_DOWN оставляли курсор в нижней или верхней части экрана при попытке вывести его ниже или выше изображения половины сектора. Сейчас мы изменим эти процедуры так, чтобы они вызывали SCROLL_UP или SCROLL_DOWN при переходе курсора соответственно через верхний или нижний край изображения. Процедуры SCROLL_UP и SCROLL_DOWN будут прокручивать изображение и устанавливать курсор в новую позицию.

Ниже приведена модифицированная версия файла PHANTOM.ASM с добавлением процедур скроллинга:

Листинг 27.1. Измененный PHANTOM.ASM

```
PHANTOM_UP      PROC    NEAR
    CALL    ERASE_PHANTOM    ;Очищает текущую позицию курсора
    DEC     PHANTOM_CURSOR_Y ;Перемещает курсор на строку
    ;                               вверх
    JNS     WASNT_AT_TOP     ;Если не верх, то записывает курсор
    MOV     PHANTOM_CURSOR_Y,0 ;Если верх, оставить там
    CALL    SCROLL_DOWN      ;Если верх, то прокрутить
WASNT_AT_TOP:
    CALL    WRITE_PHANTOM    ;Записать псевдокурсор на новом
    ;                               месте
    RET
PHANTOM_UP      ENDP

    PUBLIC  PHANTOM_DOWN
PHANTOM_DOWN    PROC    NEAR
    CALL    ERASE_PHANTOM    ;Очищает текущую позицию курсора
    INC     PHANTOM_CURSOR_Y ;Перемещает курсор на строку
    ;                               вниз CMP
    PHANTOM_CURSOR_Y, 16     ;Он внизу?
```

```
JB      WASNT_AT_BOTTOM      ;Нет, записать псевдокурсор
MOV     PHANTOM_CURSOR_Y,15  ;Да, оставить здесь
CALL    SCROLL_UP            ;Если низ, то прокрутить
WASNT_AT_BOTTOM:
CALL    WRITE_PHANTOM        ;Записать псевдокурсор
RET
PHANTOM_DOWN      ENDP
```

Не забудьте также изменить заголовки процедур PHANTOM_UP и PHANTOM_DOWN, в них необходимо указать, что теперь в этих процедурах используются SCROLL_UP и SCROLL_DOWN.

Листинг 27.2. Изменения в PHANTOM.ASM

```
; Четыре процедуры перемещения псевдокурсора
; Используются:      ERASE_PHANTOM, WRITE_PHANTOM
;                   SCROLL_DOWN, SCROLL_UP
; Считываются:      PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
; Записываются:      PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
```

SCROLL_UP и SCROLL_DOWN - достаточно простые процедуры. Они просто переключают экран на отображение первой или второй половины сектора. Например, если на экране изображена первая половина сектора и PHANTOM_DOWN вызывает SCROLL_UP, то мы увидим вторую половину. SCROLL_UP сначала изменяет SECTOR_OFFSET на 256, начало второй половины сектора, затем смещает реальный курсор в начало изображения сектора, выводит на экран изображение второй половины сектора и, наконец, устанавливает псевдокурсор в верхний правый угол изображения. Вы можете подробно ознакомиться с процедурами SCROLL_UP и SCROLL_DOWN в приводимом ниже листинге. Добавьте его к PHANTOM.ASM.

Листинг 27.3. Добавьте эти процедуры в PHANTOM.ASM.

```
EXTRN   DISP_HALF_SECTOR:NEAR, GOTO_XY:NEAR
DATA_SEG SEGMENT PUBLIC
    EXTRN   SECTOR_OFFSET:WORD
    EXTRN   LINES_BEFORE_SECTOR:BYTE
DATA_SEG ENDS
;
; Две процедуры перемещения между изображениями половин
; сектора;
;   Используются: WRITE_PHANTOM, DISP_HALF_SECTOR,
;ERASE_PHANTOM, GOTO_XY
;SAVE_REAL_CURSOR, RESTORE_REAL_CURSOR
;   Читаются:      LINES_BEFORE_SECTOR
;   Записываются:  SECTOR_OFFSET, PHANTOM_CURSOR_Y
```

```
;
SCROLL_UP      PROC      NEAR
    PUSH       DX
    CALL       ERASE_PHANTOM      ;Удалить псевдокурсор
    CALL       SAVE_REAL_CURSOR   ;Сохранить позицию реального
;                                     курсора
    XOR        DL,DL              ;Установить курсор для изображения
;                                     половины сектора
    MOV        DH,LINES_BEFORE_SECTOR
    ADD        DH,2
    CALL       GOTO_XY
    MOV        DX,256             ;Вывести вторую половину сектора
    MOV        SECTOR_OFFSET,DX
    CALL       DISP_HALF_SECTOR
    CALL       RESTORE_REAL_CURSOR ;Восстановить позицию
;                                     реального курсора
    MOV        PHANTOM_CURSOR_Y,0 ;Курсор в верху второй
;                                     половины сектора
    CALL       WRITE_PHANTOM      ;восстановить псевдокурсор
    POP        DX
    RET
SCROLL_UP      ENDP

SCROLL_DOWN    PROC      NEAR
    PUSH       DX
    CALL       ERASE_PHANTOM      ;Удалить псевдокурсор
    CALL       SAVE_REAL_CURSOR   ;Сохранить позицию реального
;                                     курсора
    XOR        DL,DL              ;Установить курсор для изображения
;                                     половины сектора
    MOV        DH,LINES_BEFORE_SECTOR
    ADD        DH,2
    CALL       GOTO_XY
    XOR        DX,DX              ;Вывести на экран первую половину
;                                     сектора
    MOV        SECTOR_OFFSET,DX
    CALL       DISP_HALF_SECTOR
    CALL       RESTORE_REAL_CURSOR ;Восстановить позицию
;                                     реального курсора
    MOV        PHANTOM_CURSOR_Y,15 ;Курсор внизу первой
;                                     половины сектора
    CALL       WRITE_PHANTOM      ;Восстановить псевдокурсор
    POP        DX
    RET
SCROLL_DOWN    ENDP
```

SCROLL_UP и SCROLL_DOWN работают вполне удовлетворительно, но мы получили ещё одну проблему. Запустите Dskpatch и оставьте курсор в верхней части экрана. Нажмите клавишу перемещения курсора

вверх и вы увидите, что Dskpatch перепишет первую половину изображения сектора. Почему? Мы не проверили это граничное условие. Dskpatch переписывает изображение вне зависимости от того, в каком направлении мы перемещаем курсор - выше или ниже изображения сектора.

Попробуйте модифицировать Dskpatch самостоятельно, заставьте его правильно функционировать с учётом соблюдения граничных условий. Если псевдокурсor находится в верхней части первой половины сектора, то он не должен перемещаться при нажатии клавиши перемещения курсора вверх. Аналогично, если псевдокурсor находится в нижней части второй половины сектора, нажатие клавиши перемещения курсора вниз также не должно вызывать перемещения курсора. Обновление изображения половины сектора также не должно производиться.

Итог

Мы завершили в этой книге работу над Dskpatch. Нашей целью было создание на примере Dskpatch "живой" программы, позволяющей как пройти все основные этапы программирования на ассемблере, так и одновременно создать работоспособные полезные программы и процедуры, которые в дальнейшем могут быть использованы в качестве составных частей разрабатываемых программ. Dskpatch в том виде, в каком она разработана на настоящем этапе книги, не является законченной версией. Дисковая версия более богата по своим возможностям, но вы можете либо самостоятельно попробовать расширить Dskpatch, либо воспользоваться непосредственно дисковой версией, поставляемой на отдельном диске. Однако учтите, что любая программа практически никогда не может быть завершена, поэтому программист должен сам решить, когда наступает пора ему остановиться.

Окончание этой книги будет посвящено обсуждению двух понятий: перемещению и дальнейших деталей сегментов.

ЧАСТЬ IV. Дополнение к сказанному

Глава 28. Перемещение

Объект, который всегда казался окруженным тайной, - это разница между .EXE-и .COM-файлами и значение перемещаемых программ. Давайте рассмотрим вопросы перемещаемости и узнаем, как можно создавать программы, объём которых превышает 64Кбайт - вам не обязательно придётся это делать, но многим приходится.

Программы, состоящие из нескольких сегментов

Если мы начнём писать программы, превышающие 64Кбайт, то столкнёмся с проблемами при запуске .COM-файлов. Почему? Об этом мы и узнаем здесь.

Прежде всего, каждая программа должна быть создана из одного или более сегментов, каждый длиной не более 64Кбайт. Но многие программы увеличивают количество используемой ими памяти за счёт применения нескольких отдельных сегментов; например, сегмент кода для программы, сегмент данных для данных и стековый сегмент для стека и динамического выделения памяти. Если каждый из этих трёх сегментов используется целиком, то заполняется $3 \times 64\text{К} = 192\text{Кбайт}$ памяти. Именно таким образом мы можем получить доступ к большему объёму памяти, и именно отсюда вытекает разница между .COM и .EXE- программами: .EXE-программы специально предназначены для такого типа работы.

Все программы, приведённые в этой книге, являются .COM-файлами и состоят из одного сегмента, или из нескольких, сгруппированных в один. Помните, что псевдооператор GROUP просто комбинирует

____Дополнение к сказанному____

несколько различных сегментов в единую область, действующую как один сегмент. Если мы захотим использовать более чем один сегмент, чтобы задействовать более чем 64Кбайт памяти, то предстоит ещё немного поработать. Рассмотрим пример.

Программа печати строки символов из главы 3 прекрасно подойдет для этого. Пример, написанный в виде групп на ассемблере, выглядит так:

```
CGROUP      GROUP      CODE_SEG, DATA_SEG
ASSUME      CS:CGROUP, DS:CGROUP

CODE_SEG    SEGMENT    PUBLIC
ORG         100h
WRITE_STRING PROC      FAR
    MOV     AH,9                ;Вызов вывода строки
    MOV     DX,OFFSET CGROUP:STRING ;Загрузка адреса строки
    INT     21h                ;Записать строку
    INT     20h                ;Возврат в DOS
WRITE_STRING ENDP
CODE_SEG    ENDS

DATA_SEG    SEGMENT PUBLIC
STRING      DB      "Hello,DOS here.$"
DATA_SEG    ENDS
END         WRITE_STRING
```

Два сегмента, CODE_SEG и DATA_SEG, помещены в одну 64-Килобайтную группу, CGROUP, так что "OFFSET CGROUP:STRING" даёт смещение STRING от начала группы CGROUP.

Когда DOS загружает в память .COM-программу, то он устанавливает все четыре сегментных регистра (CS, DS, ES и SS) так, чтобы они указывали на начало CGROUP, поэтому DS:OFFSET CGROUP: STRING является полным адресом STRING. А что если у нас есть два различных сегмента, а не группа? Предел памяти для двух сегментов не 64Кбайта, а 128Кбайт. Как мы заставим сегментные регистры указывать на их собственные сегменты?-Применяя .EXE-программы, позволяющие использовать сегменты, расположенные по разным адресам.

DOS устанавливает сегментные регистры для .EXE-программы с помощью специальных инструкций. Эти назначения не так просты, как может показаться,

___ Программы, состоящие из нескольких сегментов ___

но мы к ним ещё вернёмся. Сначала воссоздадим WRITE_STRING как .EXE-программу.

Для любой .EXE-программы необходимо иметь по крайней мере два сегмента: сегмент кода и сегмент стека. Эти два сегмента являются для DOS особым случаем. DOS сначала устанавливает четыре регистра CS, SS, IP и SP, а затем загружает .EXE-программу в память. DOS устанавливает регистровую пару CS:IP для того, чтобы указывать на первую инструкцию, адрес которой появляется после псевдооператора END. В .EXE-программе первая инструкция может быть где угодно, в то время как в .COM-программе эта инструкция должна быть первой инструкцией в сегменте кода.

Аналогично пара SS:IP указывают на конец области стека, определённой с помощью псевдооператора SEGMENT STACK. Например, приведённая ниже версия WRITE_STRING содержит стек длиной в 80 байт, следовательно, IP будет установлен в 80 - конец области стека внутри сегмента стека. Вот текст программы:

```
CODE_SEG    SEGMENT    PUBLIC
    ORG      100h
WRITE_STRING PROC        FAR
    MOV      AX,DATA_SEG    ;Адреса сегментов для DATA_SEG
    MOV      DS,AX          ;Подготовить регистр DS для
DATA_SEG
    MOV      AH,9            ;Вызвать вывод строки
    MOV      DX,OFFSET STRING ;Загрузить адрес строки
    INT      21h             ;Записать строку
    PUSH     ES              ;Сохранить адрес возврата для RET
    XOR      AX,AX           ;Здесь инструкция INT 20h (по адресу ES:0)
    PUSH     AX

    RET                                ;Возврат в DOS
WRITE_STRING ENDP

CODE_SEG    ENDS

DATA_SEG    SEGMENT    PUBLIC
STRING      DB"Hello,DOS here.$"
DATA_SEG    ENDS

STACK_SEG   SEGMENT    STACK
    DB      10 DUP (STACK    )    ;После слова 'STACK' три
;                                  пробела
```

```
STACK_SEG      ENDS  
END             WRITE_STRING
```

Эта программа будет готова к запуску после того, как вы произведёте её компоновку, но сначала удалите с диска файл WRITESTR.COM. Это необходимо сделать потому, что если на диске есть две версии файла: одна с расширением .COM, а другая с расширением .EXE, то DOS будет выполнять .COM-файл.

Между .EXE-файлом и исходным .COM-файлом есть много различий. Для возвращения в DOS вместо инструкции "INT 20h" теперь появился набор непонятных инструкций, начинающийся с "PUSH ES". Две инструкции PUSH сохраняют в стеке полный адрес возврата - ES:0. Это адрес первого байта 256-байтной области данных, которую DOS помещает в память перед программой, а первая инструкция этой области данных - "INT 20h".

Регистр CS должен указывать на начало этой области данных, когда мы выполняем инструкцию "INT 20h". Это условие выполнялось в .COM-программах, сразу же после начала. Но .EXE-программы начинаются с того, что регистр CS устанавливается на начало сегмента кода, а не области данных. Выполняя FAR RET к ES:0, мы установили CS на начало области данных, и, как видите, адрес, на который указывает ES:0, содержит инструкцию "INT 20h":

```
A:\>DEBUG WRITESTR.EXE  
-U ES:0  
39AF:0000  CD20  INT      20  
39AF:0002  006000  ADD      [BX + SI + 00],AH  
.  
.  
.
```

Псевдооператор GROUP отсутствует, потому что три различных сегмента не объединяются в общую область памяти. Каждый из этих трёх сегментов независим и каждый из сегментных регистров (CS, DS и SS) указывает на разный сегмент. Значение регистров CS и SS устанавливается DOS, что можно увидеть с помощью Debug:

```
A:\>DEBUG WRITESTR.EXE  
-R
```

```
AX=0000 BX=0000 CX=0100 DX=0000 SP=0050 BP=0000 SI=0000 DI=0000
DS=39AF ES=39AF SS=39C3 CS=39BF IP=0000 NV UP DI PL NZ NA PO NC
39BF:0000 B8C139 MOV AX,39C1
```

DS и ES указывают на сегменты, расположенные в памяти по более младшим адресам, чем те сегменты, на которые указывают регистры CS и SS. Как было показано в главе 11, и DS, и ES указывают на область данных длиной в 256 байт, установленную DOS перед нашей программой. В .COM-программах мы резервировали эту область с помощью оператора "ORG 100h". Для .EXE-файлов нет необходимости в резервировании подобной области памяти, так как сегменты данных и кода находятся в различных частях памяти. Сегмент данных находится где-нибудь в другом месте, но DS не указывает на DATA_SEG. Это причина появления первой инструкции в WRITE_STRING. Инструкция "MOV AX,DATA_SEG" пересылает номер сегмента DATA_SEG в регистр AX. Если мы посмотрим на программу

```
-U
39BF:0000 B8C139 MOV AX,39C1
39BF:0003 8ED8 MOV DS,AX
39BF:0005 B409 MOV AH,09
39BF:0007 BA0000 MOV DX,0000
39BF:000A CD21 INT 21
39BF:000C 06 PUCH ES
39BF:000D 33C0 XOR AX,AX
39BF:000F 50 PUSH AX
39BF:0010 CB RETF
39BF:0011 0000 ADD [BX+SI],AL
```

то увидим, что инструкция MOV транслирована в "MOV AX,39C1", где 39C1 - номер сегмента, в котором расположен DATA_SEG. Для того чтобы переслать это число в регистр DS, необходимо применить две инструкции MOV, так как мы не можем записать номер сегмента прямо в сегментный регистр (смотрите таблицу режимов адресации в Приложении "Е").

Откуда появилось число 39C1? Конечно, ни ассемблер, ни компоновщик не знали о том, где именно DOS загрузит программу; только DOS может это знать.

(StIV) : строка потеряна в издательстве

ления по перемещению для .EXE-программ, но не для .COM-программ. Именно по этой причине .COM-программы загружаются в память немного быстрее. Они также более компактны, так как не содержат специальной информации, на основе которой DOS определяет необходимое перемещение.

Ради любопытства давайте посмотрим, что получится, если попытаться перевести .EXE-программу в .COM-файл, используя Exe2bin:

```
A:\>EXE2BIN WRITESTR WRITESTR.COM
```

```
File cannot be converted
```

```
A:\>
```

Exe2bin знает, что нельзя создать из файла .COM-программу, но не сообщает почему. Нам остается самим строить предположения. Рассмотрим проблему более внимательно.

DOS загружает .COM-программу прямо в память сразу после того как создаёт 256-байтный заголовок. Если нам нужно использовать несколько сегментов, как в программе WRITE_STRING, и мы, кроме того, хотим создать .COM-файл, то придется выполнять перемещение самостоятельно, инструкциями программы. Это не очень трудно, и мы покажем вам, как это делается, чтобы лучше разобраться в том, как DOS выполняет перемещение программ. Если понадобится написать .COM-программу, использующую более чем 64Kбайт памяти, то вы найдете этот метод полезным.

Перемещение

Нашей целью является установить регистр DS на начало DATA_SEG, а регистр SS - на начало STACK_SEGMENT. Мы можем сделать это, применив ряд приёмов. Прежде всего, надо убедиться в том, что все три сегмента загружены в память в правильном порядке:

- сегмент кода
- сегмент данных
- сегмент стека

К счастью, мы уже позаботились об этом. Компоновщик загружает эти три сегмента в том порядке, в котором они следуют в исходном файле.

Предупреждение: когда вы используете приведённый нами метод в COM-файле для установки сегментных регистров, убедитесь в том, что вы знаете порядок загрузки сегментов.

Как мы подсчитаем значение DS? Начнём с того, что рассмотрим три метки, которые помещены в различные сегменты в приведённом ниже листинге. Это метки: END_OF_CODE_SEG, END_OF_DATA_SEG и END_OF_STACK_SEG. Эти метки не совсем то, что вы, возможно, ожидаете. Почему? Когда мы определяем сегмент как

```
CODE_SEG    SEGMENT    PUBLIC
```

то не говорим компоновщику о том, как соединять вместе различные сегменты. Он начинает каждый новый сегмент на границе параграфа, с шестнадцатеричного адреса, оканчивающегося нулем, например 32C40h. Так как компоновщик, для того чтобы начать новый сегмент, переходит к границе параграфа, то между сегментами будут часто встречаться небольшие неиспользуемые области памяти. Поместив метку END_OF_CODE_SEGMENT в начало DATA_SEG, мы включаем в сегмент этот чистый промежуток. Если поместить END_OF_CODE_SEG в конец CODE_SEG, то мы бы не смогли включить этот промежуток между сегментами (посмотрите на разассемблированный листинг нашей программы на с. 307. Вы обнаружите промежуток, заполненный нулями, имеющий длину 15 байт).

Что касается значения регистра DS, то DATA_SEG начинается с адреса 39AF:0130 или 39C2:0000. Инструкция "OFFSET CODE_SEG:END_OF_CODE_SEG" вернёт 130h, число байт, используемых CODE_SEG. Мы делим это число на 16, чтобы получить число, которое надо прибавить к DS для того, чтобы DS указывал на DATA_SEG. Мы используем тот же метод, чтобы установить SS.

Вот листинг программы, включая инструкции для перемещения для .COM-файла:

```
ASSUME      CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
CODE_SEG    SEGMENT    PUBLIC
    ORG      100h      ;Зарезервировать область памяти для .COM-
;                      program
WRITE_STRING    PROC    FAR
```

```
ORG      100h      ;Зарезервировать область памяти для .COM-
;                  program
WRITE_STRING      PROC      FAR
    MOV     AX,OFFSET CODE_SEG:END_OF_CODE_SEG
    MOV     CL,4      ;Вычислить число параграфов
    SMR     AX,CL      ; (по 16 байт), используемых сегментом кода
    MOV     BX,CS
    ADD     AX,BX      ;Добавить к результату CS
    MOV     DS,AX      ;Установить регистр DS на DATA_SEG

    MOV     BX,OFFSET DATA_SEG :END_OF_DATA_SEG

    SHR     BX,CL      ;Вычислить число параграфов, используемых
;                  сегментом данных
    ADD     AX,BX      ;Суммировать с числом, полученным для
;                  сегмента кода
    MOV     SS,AX      ;Установить регистр SS для STACK_SEG

    MOV     AX,OFFSET STACK_SEG:END_OF_STACK_SEG

    MOV     SP,AX      ;Установить SP на конец области стека
    MOV     AH,9      ;Вызов для вывода строки

    MOV     DX,OFFSET STRING      ;Загрузить адрес строки

    INT     21h      ;Записать строку
    PUSH    ES      ;Сохранить адрес возврата для RET
    XOR     AX,AX      ;Здесь инструкция INT 20h
    PUSH    AX
    RET          ;Возврат в DOS
WRITE_STRING      ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT      PUBLIC
END_OF_CODE_SEG      LABEL      BYTE
STRING      DB      "Hello, DOS here.$"
DATA_SEG      ENDS

STACK_SEG      SEGMENT PUBLIC
END_OF_DATA_SEG      LABEL BYTE
STRING      DB      10 DUP (STACK      )      ;После слова 'STACK '
;                  следуют три пробела
END_OF_STACK_SEG
STACK_SEG      ENDS      LABEL      BYTE

    END      WRITE_STRING
```

Вы можете пронаблюдать результаты всей этой работы с помощью Debug:

```
A:\>DEBUG WRITESTR.COM
```

___Перемещение___

```
-U
39AF:0100  B83001  MOV     AX,0130
39AF:0103  B104    MOV     CL,04
39AF:0105  D3E     SHR     AX,CL
39AF:0107  8CCB    MOV     BX,CS
39AF:0109  03C3    ADD     AX,BX
39AF:010B  8ED8    MOV     DX,AX
39AF:010D  BB2000  MOV     BX,0020
39AF:0110  D3EB    SHR     BX,CL
39AF:0112  03C     ADD     AX,BX
39AF:0114  8ED0    MOV     SS,AX
39AF:0116  B85000  MOV     AX,0050
39AF:0119  8BE     MOV     SP,AX
39AF:011B  B409    MOV     AH,09
39AF:011D  BA0000  MOV     DX,0000

-U
39AF:0120  CD21    INT     21
39AF:0122  06      PUSH    ES
39AF:0123  33C0    XOR     AX,AX
39AF:0125  50      PUSH    AX
39AF:0126  CB      RETF
39AF:0127  0000    ADD     [BX+SI],AL
39AF:0129  0000    ADD     [BX+SI],AL
39AF:012B  0000    ADD     [BX+SI],AL
39AF:012D  0000    ADD     [BX+SI],AL
39AF:012F  004865  ADD     [BX+SI+65],CL
39AF:0132  6C      DB     6C
39AF:0133  6C      DB     6C
39AF:0134  6F      DB     6F
39AF:0135  2C20    SUB     AL,2
39AF:0137  44      INC     SP
39AF:0138  4F      DEC     DI
39AF:0139  53      PUSH    BX
39AF:013A  206865  AND     [BX+SI+65],CH
39AF:013D  7265    JB      01A4
39AF:013F  2E      CS:
39AF:0140  2400    AND     AL,00

AX=0950 BX=0002 CX=0004 DX=0000 SP=0050 BP=0000 SI=0000 DI=0000
DS=39C2 ES=39AF SS=39C4 CS=39AF IP=0120 NV UP DI PL HZ NA PO NC
39AF:0120 CD21 INT21
```

Выполняя перемещение для более чем одного сегмента самостоятельно, мы увеличиваем количество

памяти, которое может использовать .COM-программа. Большая часть программистов не нуждается в таких трюках, но знание того, как работает перемещение, позволит понять, как DOS выполняет его для .COM-файлов.

.COM-программы в сравнении с .EXE-программами

Мы закончим эту главу, суммировав отличия между .COM и .EXE-файлами.

.COM-программы хранятся на диске в виде точного образа программы в памяти. Поэтому .COM-программы ограничены единственным сегментом, если они не выполняют перемещение самостоятельно, как мы это делали в этой главе.

С другой стороны, .EXE-программа позволяет DOS самому заняться перемещением. Это делегирование значительно упрощает использование .EXE-программами нескольких сегментов. По этой причине большинство длинных программ .EXE, а не .COM-типа.

Чтобы провести последнее сравнение .COM и .EXE-программ, рассмотрим подробнее процесс того, как DOS загружает их и выполняет. Это сделает ясными и более конкретными различия между программами этих типов. Мы начнём с .COM-программ.

Когда DOS загружает в память .COM-программу, то он выполняет три шага:

- создаёт PSP - префикс программного сегмента, который и является той рабочей областью, с которой мы познакомились в главе 11. Кроме всего прочего, этот PSP содержит командную строку, которую мы печатаем;
- целиком копирует .COM-файл с диска в память, непосредственно после 256 байт PSP;
- устанавливает все четыре сегментных регистра (CS, DS, ES и SS) на начало PSP;
- устанавливает значение регистра IP в 100h (начало .COM-программы), а регистр SP - на конец сегмента - FFFE, которое обычно является последним словом сегмента.

По сравнению с вышеприведенными действиями шаги по загрузке .EXE-файла сложнее, так как DOS

выполняет перемещение. Где DOS находит информацию, нужную ему для перемещения?

Как оказывается, каждый .EXE-файл содержит заголовок, который хранится в начале файла. Этот заголовок, или таблица перемещения ("relocation table"), всегда имеет в длину по крайней мере 512 байт и содержит всю необходимую DOS информацию. В наиболее поздние версии макроассемблера фирма Microsoft включила программу, называющуюся EXE-MOD, которую можно использовать для того, чтобы просмотреть некоторую информацию в этом заголовке:

```
A:\>EXEMOD WRITESTR

Microsoft (R) EXE File Header Utility Version 4.00
Copyright (C) Microsoft Corp 1985. All rights reserved.

WRITESTR                (hex)                (dec)

.EXE size (bytes)        290                    656
Minimum load size (bytes) 90                    144
Overlay number           0                      0
Initial CS:IP             0000:0000
Initial SS:SP             0004:0050            80
Minimum allocation (para) 0                      0
Maximum allocation (para) FFFF                65535
Header size (para)        20                    32
Relocation table offset   1E                    30
Relocation entries        1                      1

A:\>
```

В начале этой таблицы размещена точка входа перемещения для программы WRITESTR. Каждый раз, когда мы ссылаемся на адрес сегмента (инструкция "MOV AX,DATA_SEG"), LINK добавляет в таблице ещё одну точку входа перемещения. Адрес сегмента не известен до тех пор, пока DOS не начнёт загружать программу в память, и поэтому мы должны позволить DOS установить номер сегмента.

В таблице также содержатся некоторые другие интересные данные, например начальные значения CS:IP и SS:SP. Эти данные сообщают начальные значения для IP и SP. Таблица также сообщает DOS о том, какое количество памяти необходимо загружаемой программе для того, чтобы запуститься ("Minimum load size").

Так как DOS использует эту таблицу перемещения, чтобы установить абсолютные адреса сегментов, то он

предпринимает ряд дополнительных шагов при загрузке программы в память. Вот этапы, которым следует DOS при загрузке .EXE-программы:

- создаёт PSP (префикс программного сегмента), так же как он это делает для .COM-программы;
- просматривает заголовок .EXE-файла, из которого определяет, где именно оканчивается заголовок и начинается программа. Затем загружает в память остальную часть программы после PSP;
- затем, используя информацию, содержащуюся в заголовке, DOS находит и исправляет все перемещаемые ссылки в программе, например ссылки на адреса сегментов;
- устанавливает значения регистров ES и DS так, чтобы они указывали на начало PSP. Если у вашей программы есть свой сегмент данных, то ей надо изменить DS и/или ES, чтобы они указывали на ваш сегмент данных;
- устанавливает регистр CS на начало сегмента кода, с IP, значение которого установлено исходя из информации, содержащейся в заголовке .EXE-файла. Аналогично, DOS устанавливает SS:SP в соответствии с информацией из .EXE-заголовка. В приведённом примере заголовок сообщает о том, что SS:SP будут указывать на 0004:0050. DOS устанавливает значение SP в 0050, и значение регистра SS таким образом, чтобы он указывал на адрес в памяти на четыре параграфа старше, чем конец PSP.

Глава 29. Подробнее о сегментах и ASSUME

В этой последней главе мы ещё раз взглянем на оператор ASSUME и рассмотрим, как он связан с применением сегментов. Мы также узнаем о приёме, называемом переприсвоение сегментов ("segment override"). Переприсвоение сегментов непосредственно связано с оператором ASSUME.

Переприсвоение сегментов

До этого мы всегда считывали и записывали данные, размещённые в сегменте данных. В этой книге

мы имели дело только с одним сегментом (применяя группировку сегментов), так что у нас не было причины считывать или записывать данные в других сегментах.

Но, как мы видели, .EXE-программы содержат несколько сегментов, и даже .COM-программы могут содержать или использовать несколько сегментов. Классический пример этого - запись прямо на экран: многие коммерческие программы выводят данные непосредственно на экран, пересылая данные прямо в экранную область памяти, совершенно не используя при этом подпрограммы ROM BIOS. Этот приём значительно повышает быстродействие. Экранная область в IBM PC располагается в сегменте B800h для цветного графического адаптера (CGA) и в сегменте BOOOh - для монохромных графических адаптеров. Запись данных прямо в экранную область и означает, что необходимо организовать запись в различные сегменты.

В этом разделе мы напишем в качестве примера короткую программу, в которой покажем пример записи данных в два различных сегмента с применением регистров DS и ES в качестве указателей на эти сегменты.

Программа очень коротка и содержит два сегмента данных, в каждом из которых расположена одна переменная:

```
DATA_SEG      SEGMENT      PUBLIC
DS_VAR        DW           1
DATA_SEG      ENDS

EXTRA_SEG      SEGMENT      PUBLIC
DS_VAR        DW           2
EXTRA_SEG      ENDS

STACK_SEG      SEGMENT      PUBLIC
DB            10 DUP ( 'STACK ' )      ; за словом
;                                       "STACK" следуют
;                                       три пробела
STACK_SEG      ENDS

CODE_SEG      SEGMENT      PUBLIC
ASSUME        SD:CODE_SEG, DS:DATA_SEG, ES:EXTRA_SEG,
SS:STACK_SEG
TEST          PROC          FAR
    PUSH      ES              ;Сохранить адрес для RET
    XOR       AX,AX
```

```
PUSH    AX
MOV     AX,DATA_SEG    ;Адрес сегмента для DATA_SEG
MOV     DS,AX          ;Подготовить регистр DS для
DATA_SEG
MOV     AX,EXTRA_SEG    ;Адрес сегмента для EXTRA_SEG
MOV     ES,AX          ;Подготовить регистр ES для
EXTRA_SEG
MOV     AX,DS_VAR       ;Считать переменную из сегмента
;данных
MOV     BX,ES:ES_VAR    ;Считать переменную из EXTRA
;сегмента данных
RET                                ;Возврат в DOS
TEST    ENDP
CODE_SEG ENDS
END      TEST
```

На примере этой программы мы попробуем поглубже разобраться с псевдооператором ASSUME и процедурой переприсвоения сегментов.

Обратите внимание, что оба сегмента данных и сегмент стека размещены перед сегментом кода, а псевдооператор ASSUME расположен после всех сегментных объявлений. Такая перестановка является прямым следствием использования двух сегментов данных.

Рассмотрим две инструкции MOV в этой программе:

```
MOV     AX,DS_VAR
MOV     BX,ES:ES_VAR
```

ES: перед второй инструкцией сообщает микропроцессору 8088 о необходимости применения в операции считывания данных из дополнительного сегмента регистра ES, а не DS. Каждая инструкция имеет свой сегментный регистр, использующийся по умолчанию при обращении к данным. Но так же, как и в случае с регистром ES, мы можем сообщить 8088 о том, что для данных будут применяться другие сегментные регистры.

Например, инструкция "MOV AX,ES:ES_VAR" транслируется в две инструкции. В разассемблированной программе это выглядит следующим образом:

```
2CF4:0011    26      ES:
2CF4:0012    8B1E0000  MOV     BX,[0000]
```

Ассемблер транслировал инструкцию

```
MOV     AX,ES:ES_VAR
```

в инструкцию переприсвоения сегмента и в следующую за ней инструкцию MOV. Теперь MOV будет считывать данные из сегмента ES, а не DS.

Если вы протрассируете программу, то увидите, что первая инструкция MOV устанавливает значение регистра AX, равное 1(DS_VAR), а вторая инструкция MOV устанавливает регистр BX в 2 (ES_VAR). Другими словами, мы считываем данные из двух различных регистров.

Ещё один взгляд на ASSUME

Посмотрим, что произойдет, если удалить из программы "ES:". Измените строку:

```
MOV     BX,ES:ES_VAR
```

чтобы она выглядела так:

```
MOV     BX,ES_VAR
```

Сейчас ассемблеру не сообщается о том, что необходимо применить регистр ES во время считывания из памяти, и следовательно, он должен вернуться к сегментам, использующимся по умолчанию. Правильно ли это? Нет.

Примените Debug и посмотрите на результат изменения. Переприсвоение сегмента "ES:" по-прежнему находится перед инструкцией MOV. Каким образом ассемблер узнал о том, что переменная находится в дополнительном сегменте, а не в сегменте данных? Это произошло на основе информации, переданной ассемблеру псевдооператором ASSUME.

Оператор ASSUME "говорит" ассемблеру, что регистр DS указывает на сегмент DATA_SEG, в то время как регистр ES указывает на EXTRA_SEG. Всякий раз, встретив инструкцию, в которой содержится имя переменной памяти, ассемблер ищет определение этой переменной, чтобы узнать, в каком сегменте она объявлена. После этого ассемблер просматривает список ASSUME, из которого узнает, какой регистр указывает на этот сегмент и затем использует этот сегмент при генерации инструкции.

Прочитав в программе инструкцию

```
"MOV BX,ES_VAR",
```

ассемблер заметил, что ES_VAR находится в сегменте EXTRA_SEG и что регистр ES указывает на этот сегмент, поэтому он самостоятельно сгенерировал инструкцию переприсвоения сегментов "ES:". Если мы

перенесём `ES_VAR` в `STACK_SEG` (сегмент стека), то ассемблер будет генерировать инструкцию переприсвоения сегмента `"SS:"`. Ассемблер автоматически генерирует необходимую инструкцию переприсвоения сегмента при условии что псевдооператор `ASSUME` отражает действительное состояние сегментных регистров.

Фазовые ошибки

Вы можете столкнуться с тем, что ассемблер выдаст несколько странное сообщение об ошибке `"Phase error between passes"` ("Фазовая ошибка между проходами"). Это сообщение имеет много значений, но мы рассмотрим один особый случай, чтобы помочь понять его.

В основном ассемблер выполняет несколько проходов через программу, когда генерирует её машинную версию. Иногда при этом, как мы увидим далее, программа изменяет длину.

Используя пример программы ещё раз, переместите два сегмента данных (`DATA_SEG` и `EXTRA_SEG`) так, чтобы они находились после сегмента кодов. Теперь ассемблер будет ассемблировать главную программу перед тем, как он просмотрит сегменты данных. В результате он будет генерировать для

```
"MOV BX,ES_VAR"
```

обычную инструкцию `MOV`, так как не знает, что эта переменная находится в другом сегменте.

Затем ассемблер будет транслировать два сегмента данных. В этот момент он сохранит информацию, что `ES_VAR` находится в сегменте `EXTRA_SEG`. При следующем проходе через программу ассемблер заметит, что ему теперь необходимо место для инструкции переприсвоения сегмента. Так как он не резервировал места для этой инструкции при первом проходе, то ассемблер выдаст сообщение об ошибке: `"Phase error between passes"` ("Фазовая ошибка между проходами").

Вот почему мы поместили все наши сегменты данных перед сегментом кода: чтобы ассемблер знал, какой сегмент какие именно переменные содержит. Однако не совсем ясно, почему мы поместили оператор `ASSUME` в `CODE_SEG`, а не в начало этого файла.

Сообщение о фазовой ошибке появится и в случае, если поместить `ASSUME` в файл первым. По некоторым причинам (не вполне ясным для нас до сих пор)

[pause]

необходимо объявлять сегменты перед псевдооператором ASSUME, если мы собираемся использовать переприсвоения сегментов. Таким образом, наиболее безопасным подходом будет объявить все данные перед сегментом кодов и поместить псевдооператор ASSUME в сегменте кода.

Послесловие

Вы познакомились с большим количеством программ, написанных на языке ассемблера. При работе с ними мы постоянно придавали особое значение самому процессу программирования, а не деталям микропроцессора 8088 IBM Personal Computer. В результате вы не увидели ни всех инструкций 8088, ни всех псевдооператоров. Но большая часть ассемблерных программ может быть написана с помощью того, что мы здесь изучили и не более. Наиболее эффективный способ дальнейшего изучения ассемблера - взять программы из этой книги и модифицировать их.

Если вы считаете, что есть лучший способ написания какой-либо части Dskpatch, то сделайте это. Именно таким способом мы учились писать собственные программы. Правда тогда писали на Бейсике, но сама идея остается неизменной. Мы брали работающие программы и начинали изучение языка, переписывая и изменяя части этих программ. То же самое можете сделать с Dskpatch.

После того, как опробуете некоторые из этих примеров, вы будете готовы писать свои собственные программы. Не пытайтесь написать программу полностью самостоятельно; вначале это будет слишком тяжело. Чтобы начать, используйте программы из этой книги как каркас. Не пытайтесь создать совершенно новую структуру или технический приём (свой эквивалент модульного конструирования), пока вы не почувствуете себя в ассемблере достаточно уверенно.

Если вы действительно очарованы языком ассемблера, то вам также потребуется более полное руководство по набору инструкций микропроцессора 8088. Ниже приводится список подобных руководств, вышедших ко времени создания книги. Этот список, конечно, не полный, и книги, в нем содержащиеся, - это только те, которые мы читали.

Первые две книги - это хорошие руководства для программистов:

iAPX 88 Book; Intel; 1981. ;

Rector, Russel, and Alexy, George; The 8086 Book
Osborne/McGraw-Hill; 1980.

Следующие три книги написаны для IBM PC.
Большая часть информации в каждой из них носит
общий характер, только примеры в последних частях
этих книг специфичны для IBM PC. Мы рекомендуем
вам просмотреть все три книги, чтобы решить, какая из
них для вас более интересна:

Scanlon, Leo J.; IBM PC & XT Assembly Language:
A Guide for Programmers, Enhanced and Enlarged; Brady
Communication Co.; 1985.;

Widen, David C., and Krantz, Jeffrey I.; 8088
Assembler Language Programming: The IBM PC; Howard
W. Sams & Co.; 1983.;

Bradley, David J.; Assembly Language Programming
for the IBM Personal Computer; Prentice-Hall; 1984.

Ещё одна рекомендуемая книга не руководство и
не введение в IBM PC. Это введение в сам
микропроцессор '8088, написанное членами конструкторской команды фирмы Intel:

Morse, Stephen P.; The 8086/8088 Primer; Hayden;
1982.

Наконец, последняя книга это руководство, которое
полезно для любого, кто занимается программирова-
нием на IBM PC. При её создании мы попытались вклю-
чить в нее все, что может потребоваться знать прог-
раммисту о микропроцессорах семейства IBM PC.

Norton, Peter; Programmer's Guide to the IBM PC;
Microsoft Press; 1985.

Приложение А. Руководство по диску

Дискета, которая может быть куплена дополнительно к этой книге, содержит большую часть примеров по Dskpatch, представленных в предыдущих главах, а также расширенную версию программы, включающую дополнительные возможности. Файлы разделены на две группы: примеры к главам и улучшенная версия программы Dskpatch. Это приложение разъясняет содержимое дискеты.

Примеры к главам

Все примеры относятся к главам 9 - 27. Примеры в более ранних главах невелики по объёму и поэтому не представлены на дискете. С главы 9 мы начали создавать Dskpatch, который к концу этой книги вырос до девяти отдельных файлов.

В каждой из глав изменяются только некоторые из этих девяти файлов. На дискете представлены примеры в том состоянии, в каком они находятся в конце каждой главы. Таким образом, если мы изменим программу несколько раз на протяжении, скажем, главы 19, то дискета будет содержать только окончательную версию.

Таблица на странице 294 показывает то место в книге, в котором изменяется соответствующий файл. В ней также представлены имена всех дисковых файлов. Если вы хотите убедиться в том, что двигаетесь в правильном направлении, или не понимаете, каким образом вносить изменения в программу, то просмотрите таблицу. Вы можете либо исправить программу либо просто скопировать её.

Ниже представлен полный список файлов на поставляемом диске (без улучшенной версии Dskpatch):

10-6588

-- 289 [pause]

VIDEO_9.ASM	VIDEO_10.ASM	VIDEO_13.ASM	TEST13.ASM
DISP_S14.ASM	CURSOR14.ASM	VIDEO_14.ASM	DISP_S15.ASM
DISK_15.ASM	DISP_S16.ASM	VIDEO_16.ASM	DISK_16.ASM
DSKPAT17.ASM	DISP_S17.ASM	CURSOR17.ASM	VIDEO_17.ASM
DISK_17.ASM	CURSOR18.ASM	VIDEO_18.ASM	DSKPAT19.ASM
DISPAT19.ASM	KBD_IO19.ASM	VIDEO_19.ASM	DISK_19.ASM
DISP_S21.ASM	PHANTO21.ASM	VIDEO_21.ASM	DISPAT22.ASM
EDITOR22.ASM	PHANTO22.ASM	KBD_IO23.ASM	TEST23.ASM
KBD_IO24.ASM	DISPAT25.ASM	DISPAT26.ASM	DISK_26.ASM
PHANTO27.ASM			

Улучшенная версия Dskpatch

На диске содержатся не только примеры программ, приведенные в книге. Хотя мы не полностью закончили Dskpatch в конце главы 27, версия Dskpatch, приведенная на диске, содержит много дополнительных свойств, делающих её более удобной в использовании.

Ниже приводится краткий обзор того, что вы найдёте в ней.

В версии, которая содержится в этой книге, Dskpatch может считывать только следующий или предыдущий сектора. То есть, если вы хотите считать сектор 576, вам придется нажать клавишу F2 575 раз. Это неудобно. А что если вам захочется просмотреть сектора, из которых состоит весь файл? Вам придется просматривать сектор директории и затем высчитывать, где искать сектора файла. Не очень привлекательное занятие. Дисковая версия, так же как и книжная, может считывать абсолютные сектора, но дополнительно может считывать сектора, относящиеся только к заданному файлу. В таком виде Dskpatch стал очень удобной.

Улучшенная версия Dskpatch содержит слишком много изменений, чтобы описывать их здесь детально, поэтому рассмотрим только новые функции, которые мы добавили к дисковой версии. Большую часть этих изменений вы можете предугадать, исследуя Dskpatch и внося свои собственные коррективы.

Улучшенная версия Dskpatch содержит по-прежнему девять файлов, все они содержатся на диске:

DSKPATCH.ASM	DISPATCH.ASM	DISP_SEC.ASM	KBD_IO.ASM
CURSOR.ASM	EDITOR.ASM	PHANTOM.ASM	VIDEO_IO.ASM
DISK_IO.ASM	DSKPATCH.COM		

Вы также найдете на диске ассемблированную и откомпилированную .COM версию, готовую к запуску, поэтому можете опробовать её, не ассемблируя.

Вы отметите улучшения, только взглянув на изображение экрана. Усовершенствованный Dskpatch использует теперь восемь функциональных клавиш. Для того чтобы упростить работу с ними, усовершенствованный Dskpatch содержит "строку клавиш" внизу экрана. После нажатия функциональных клавиш происходят следующие действия:

F1,F2 использовались в этой книге. F1 считывает предыдущий сектор, а F2 считывает следующий сектор.

F3 изменяет номер или букву дисководов. Нажмите F3 и введите букву, например A (без двоеточия, ":"), или введите номер дисководов, например 0. Когда вы нажмете клавишу Enter, Dskpatch сменит текущий дисковод и считывает сектор с нового дисководов. Вы, возможно, захотите изменить Dskpatch так, чтобы он не считывал новый сектор при изменении текущего дисководов, однако мы сделали это только для того, чтобы усложнить запись сектора не на тот диск.

F4 изменяет номер сектора. Нажмите F4 и напечатайте номер сектора в десятичной системе счисления. Dskpatch считывает этот сектор.

F5 используется в этой книге. Нажмите клавишу Shift и F5 для того, чтобы записать сектор на диск.

F6 переводит Dskpatch в файловый режим. Введите имя файла, и Dskpatch будет считывать сектора, относящиеся к этому файлу. После нажатия клавиши F1 ("предыдущий сектор") и F2 ("следующий сектор") будут считываться соответствующие сектора указанного файла. F3 завершает файловый режим и переключается обратно в режим считывания абсолютных секторов.

F7 запрашивает смещение в файле. Это похоже на действие F4 ("сектор"), только считываются сектора, принадлежащие к указанному ранее файлу.

Если вы укажете смещение, равное 3, Dskpatch считает четвёртый сектор в файле.

F10 выход из Dskpatch. Если вы случайно нажмёте эту клавишу, то вернетесь обратно в DOS и потеряете изменения, сделанные в последнем секторе. Возможно, вы захотите изменить Dskpatch таким образом, чтобы он спрашивал, действительно ли мы хотим выйти обратно в DOS.

Новая версия содержит целый ряд других изменений, которые не так просты, как те, которые мы уже рассмотрели. Например, Dskpatch теперь осуществляет прокрутку (скроллинг) экрана по одной строке за раз. Если вы подведёте курсор к нижней строке и нажмёте клавишу Cursor-Down (клавиша со стрелкой - "курсор вниз"), то Dskpatch сдвинет изображение на одну строку, поместив внизу экрана новую строку изображения.

Кроме того, теперь заработали некоторые другие клавиши клавиатуры:

Home перемещает псевдокурсor в начало изображения половины сектора и прокручивает изображение так, что можно видеть его первую половину. **End** перемещает псевдокурсor в нижнюю часть изображения половины сектора и прокручивает изображение так, что можно видеть вторую половину сектора.

PgUp прокручивает изображение половины сектора на четыре строки. Эта возможность очень удобна, если вы хотите быстро передвигаться по изображению сектора. Если вы нажмёте **PgUp** четыре раза, то увидите вторую половину сектора.

PgDn прокручивает изображение половины сектора на четыре строки в направлении, противоположном **PgUp**.

Вы можете, если хотите, модифицировать усовершенствованный Dskpatch так, чтобы он лучше соответствовал вашим собственным нуждам. Именно поэтому диск содержит все исходные тексты улучшенного Dskpatch: таким образом вы можете модифицировать Dskpatch. Например, вы можете попробовать добавить возможности по проверке ошибок. В окончательной версии Dskpatch отсутствует проверка на выход за границы диска или файла при нажатии на

клавишу F2. Если вы чувствуете себя уже уверенно, то можете исправить Dskpatch так, чтобы он отлавливал и корректировал подобные ошибки.

Или, возможно, вы захотите убыстрить обновление экрана. Для этого необходимо изменить некоторые процедуры, такие, как WRITE_CHAR и WRITE_ATTRIBUTE_N_TIME, чтобы они записывали значения сектора непосредственно в экранную область памяти. Пока же они используют достаточно медленные подпрограммы ROM BIOS. Если вы действительно честолюбивы, попробуйте написать собственные, более быстродействующие процедуры вывода символов.

No Главы	DSKPATCH	DSKPATCH	DISP_SEC	KBD_IO	CURSOR	EDITOR	PHANTOM	VIDEO_IO	DISK_IO	TEST
9								VIDEO_9.ASM		
10								VIDEO_10.ASM		
13								VIDEO_13.ASM		TEST13.ASM
14			DISP_S14.ASM		CURSOR14.ASM			VIDEO_14.ASM		
15			DISP_S15.ASM		v			v	DISK_115.ASM	
16			DISP_S16.ASM		v			VIDEO_16.ASM	DISK_116.ASM	
17	DSKPAT17.ASM		DISP_S17.ASM		CURSOR17.ASM			VIDEO_17.ASM	DISK_117.ASM	
18	v		v		CURSOR18.ASM			VIDEO_18.ASM	v	
19	DSKPAT19.ASM	DSKPAT17.ASM	v	KBD_IO19.ASM	v			VIDEO_19.ASM	DISK_119.ASM	
21		v	DISP_S21.ASM	v	v		PHANTO21.ASM	VIDEO_21.ASM	v	
22		DSKPAT22.ASM	v	v	v	EDITOR22.ASM	PHANTO22.ASM	v	v	
23			v	KBD_IO23.ASM	v	v	v	v	v	TEST23.ASM
24			v	KBD_IO24.ASM	v	v	v	v	v	
25		DSKPAT25.ASM	v	v	v	v	v	v	v	
26		DSKPAT26.ASM	v	v	v	v	v	v	DISK_126.ASM	
27		v	v	v	v	v	PHANTO27.ASM	v		

Приложение В. Листинг DSKPATCH

Это приложение содержит листинг последней версии Dskpatch, описанной в этой книге. Если вы создаёте собственные программы, то в этом приложении найдете много процедур общего назначения, которые помогут вам. Мы также включили сюда краткое описание действия каждой процедуры.

Описание процедур

CURSOR.ASM

CLEAR_SCREEN действует как команда Бейсика CLS; очищает экран в текстовом режиме.

CLEAR_TO_THE_END_OF_LINE стирает все символы от позиции курсора до конца текущей строки.

CURSOR_RIGHT смещает курсор на одну позицию вправо, не стирая предыдущий символ.

GOTO_XY очень похоже на команду Бейсика LOCATE; изменяет местоположение курсора на экране.

SEND_CRLF посылает на экран пару символов "возврат каретки"/"перевод строки". Эта процедура просто переводит курсор на следующую строку.

DISK_IO.ASM

NEXT_SECTOR добавляет единицу к текущему номеру сектора, считывает этот сектор в память и отображает его на экран.

PREVIOUS_SECTOR считывает предыдущий сектор. Процедура вычитает единицу из старого номера сектора (**CURRENT_SECTOR_NO**) и считывает новый сектор в переменную в памяти **SECTOR**. Эта процедура также обновляет изображение на экране.

READ_SECTOR считывает с диска один сектор (512 байт) в буфер в памяти, **SECTOR**.

WRITE_SECTOR записывает один сектор (512 байт) из буфера в памяти, **SECTOR**, на диск.

DISPATCH.ASM

DISPATCHER — центральный диспетчер считывает символы, вводимые с клавиатуры, и затем вызывает процедуры, выполняющие необходимые действия. Добавление в Dskpatch новых команд производится через таблицу DISPATCH_TABLE в этом же файле.

DISP_SEC.ASM

DISP_HALF_SECTOR — выполняет работу по выводу на экран всех шестнадцатеричных и ASCII символов, формирующих изображение половины сектора, вызывая процедуру DISP_LINE 16 раз.

DISP_LINE — выводит на экран одну строку изображения половины сектора. DISP_HALF_SECTOR обращается к этой процедуре 16 раз, чтобы высветить все 16 строк изображения половины сектора.

INI_SEC_DISP — инициализирует изображение половины сектора, которое вы видите в Dskpatch. Эта процедура перерисовывает изображение половины сектора вместе с обрамляющими его линиями и шестнадцатеричными числами вверху, но не печатает заголовок и приглашение редактора.

WRITE_PROMPT_HEX_NUMBERS — печатает строку шестнадцатеричных символов вдоль верхней части изображения половины сектора. Эта процедура ни для чего другого не используется.

DSKPATCH.ASM

DISK_PATCH — основная часть Dskpatch. DISK_PATCH просто обращается к большому числу других процедур, которые делают всю работу. Она также содержит много определений переменных, используемых в Dskpatch.

EDITOR.ASM

EDIT_BYTE — выполняет редактирование байта изображения половины сектора, изменяя один байт как в памяти (SECTOR), так и на экране. Dskpatch использует эту процедуру для изменения байтов в считанном секторе.

WRITE_TO_MEMORY — вызывается из EDIT_BYTE для изменения одного байта в SECTOR. Эта процедура изменяет байт, на который указывает псевдокурсор.

KBD_IO.ASM

BACKSPACE используется процедурой **READ_STRING** для удаления одного символа как с экрана, так и из буфера клавиатуры, как если бы вы нажали клавишу **Backspace**.

CONVERT_HEX_DIGIT переводит один ASCII символ в его шестнадцатеричный эквивалент. Например, эта процедура переведет букву 'A' в шестнадцатеричное число 0AH.

Примечание. **CONVERT_HEX_DIGIT** работает только с заглавными буквами.

HEX_TO_BYTE переводит строку символов, состоящую из двух символов, из строки шестнадцатеричных символов, например AS, в байт, имеющий это шестнадцатеричное значение. **HEX_TO_BYTE** подразумевает, что оба символа являются цифрами или заглавными буквами.

READ_BYTE использует **READ_STRING** для считывания строки символов. Эта процедура возвращает номер функциональной клавиши или шестнадцатеричный байт, который вы ввели в виде строки, состоящей из двух шестнадцатеричных символов.

READ_DECIMAL считывает десятичное число без знака с клавиатуры, используя **READ_STRING** для считывания символов. **READ_DECIMAL** может считывать числа от 0 до 65535.

READ_STRING считывает введенную с клавиатуры строку символов почти так же, как это делает DOS. Эта процедура также считывает специальные функциональные клавиши; функция считывания DOS этого не делает.

STRING_TO_UPPER процедура общего назначения, переводит строку формата DOS в заглавные буквы.

PHANTOM.ASM

ERASE_PHANTOM удаляет два псевдокурсора с экрана, изменяя атрибут символа на нормальный (7) для всех символов в позиции псевдокурсоров.

MOV_TO_ASCII_POSITION перемещает реальный курсор в позицию псевдокурсора в ASCII-окне изображения половины сектора.

MOV_TO_HEX_POSITION перемещает реальный курсор к началу псевдокурсора в шестнадцатеричном окне изображения половины сектора.

PHANTOM_DOWN перемещает псевдокурсор вниз и прокручивает изображение, если вы попытаетесь опуститься ниже шестнадцатой строки изображения половины сектора.

PHANTOM_LEFT перемещает псевдокурсор влево на одну позицию, не пересекая левой границы изображения.

PHANTOM_RIGHT перемещает псевдокурсор вправо на одну позицию, но не далее правой границы изображения половины сектора.

PHANTOM_UP перемещает курсор вверх на одну строку внутри изображения половины сектора или прокручивает изображение, если вы попытаетесь вывести курсор за верхнюю границу экрана.

RESTORE_REAL_CURSOR возвращает реальный курсор в позицию, сохраненную **SAVE_REAL_CURSOR**.

SAVE_REAL_CURSOR сохраняет позицию реального курсора в двух переменных. Вызывайте эту процедуру перед тем, как переместить реальный курсор, если хотите затем восстановить его позицию, после внесения изменений на экране.

SCROLL_DOWN высвечивает первую половину сектора. Усовершенствованную версию этой функции можно найти на диске, которая может быть куплена в дополнение к этой книге. Улучшенная версия прокручивает изображение по одной строчке.

SCROLL_UP к этой процедуре обращается **PHANTOM_DOWN**, когда вы пытаетесь переместить курсор дальше нижней границы изображения. Версия этой процедуры в книге на самом деле не прокручивает изображение: она просто высвечивает вторую половину сектора. На диске есть более совершенные версии **SCROLL_UP** и **SCROLL_DOWN**, прокручивающие изображение по одной строчке, а не сразу 16.

WRITE_PHANTOM рисует псевдокурсоры на изображении половины сектора: один - в шестнадцатеричном окне, один - в ASCII-окне. Эта процедура просто изменяет атрибут символа на 70H, чтобы использовались чёрные символы на белом фоне.

VIDEO_IO.ASM

Этот файл содержит в основном процедуры общего назначения, которые вы можете использовать в собственных программах.

WRITE_ATTRIBUTE_N_TIMES удобная процедура, которую вы можете использовать для изменения атрибутов группы из N символов. WRITE_PHANTOM использует эту процедуру для изображения псевдокурсоров, а ERASE_PHANTOM использует эту процедуру для удаления псевдокурсоров.

WRITE_CHAR печатает символ на экране. Так как эта процедура использует подпрограммы ROM BIOS, то она не распознает специальных символов. Символ возврата каретки будет появляться на экране в виде музыкальной ноты (символ, соответствующий 0DH). Используйте SEND_CRLF, если хотите переместить курсор к началу следующей строки.

WRITE_CHAR_N_TIMES печатает N копий одного символа на экране. Эта процедура полезна для печати строк символов, например, символов линий.

WRITE_DECIMAL печатает слово (16 разрядов) на экране в виде десятичного числа без знака в промежутке между 0 и 65535.

WRITE_HEADER печатает заголовок в верхней части изображения, которое вы видели в Dskpatch.

WRITE_HEX берет число, состоящее из одного байта, и печатает его в виде двузначного шестнадцатеричного числа.

WRITE_HEX_DIGIT печатает на экране шестнадцатеричное число, состоящее из одного символа. Эта процедура переводит тетраду (четыре бита) в ASCII-символ и печатает его на экране.

WRITE_PATTERN рисует вокруг изображения половины сектора рамки, состоящие из указанного символа ("pattern"). Используйте WRITE_PATTERN для печати любых символов на экране.

WRITE_STRING процедура общего назначения, с помощью которой на экране печатается строка символов. Последним символом в указанной строке должен быть нулевой байт.

WRITE_PROMPT_LINE печатает строку приглашения и затем стирает остаток строки, удаляя символы, оставшиеся от предыдущего приглашения.

Листинг процедур Dskpatch

DSKPATCH Make File

Приведённый ниже листинг Make-файла позволяет создавать DSKPATCH автоматически.

```
DSKPATCH.OBJ :    DSKPATCH.ASM
    MASM    DSKPATCH;
DISK_IO.OBJ :    DISK_IO.ASM
    MASM    DISK_IO;
DISP_SEC.OBJ :    DISP_SEC.ASM
    MASM    DISP_SEC;
Video_io.OBJ :    VIDEO_IO.ASM
    MASM    VIDEO_IO;
CURSOR.OBJ :    CURSOR.ASM
    MASM    CURSOR;
DISPATCH.OBJ :    DISPATCH.ASM
    MASM    DISPATCH;
KBD_IO.OBJ :    KBD_IO.ASM
    MASM    KBD_IO;
PHANTOM.OBJ :    PHANTOM.ASM
    MASM    PHANTOM;
EDITOR.OBJ :    EDITOR.ASM
    MASM    EDITOR;
DSKPATCH.COM :    DSKPATCH.OBJ DISK_IO.OBJ DISP_SEC.OBJ VIDEO_IO.OBJ
CURSOR.OBJ

DISPATCH.OBJ      KBD_IO.OBJ
PHANTOM.OBJ        EDITOR.OBJ

    LINK    @LINKINFO
    EXE2BIN    DSKPATCH DSKPATCH.COM

CURSOR.ASM

CR    EQU    13        ;Carriage return
LF    EQU    10        ;Linefeed

CGROUP    GROUP        CODE_SEG
    ASSUME    CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC

    PUBLIC        SEND_CRLF
;
; Процедура обеспечивает корректный скроллинг, посылая пару
; символов "возврат каретки/перевод строки" на экран, применяя
```

```
;процедуру DOS.
;
SEND_CRLF      PROC      NEAR
    PUSH        AX
    PUSH        DX
    MOV         AH,2
    MOV         DL,CR
    INT         21h
    MOV         DL,LF      ;LF equ 10 - LineFeed
    INT         21h
    POP         DX
    POP         AX
    RET
SEND_CRLF      ENDP

PUBLIC         CLEAR_SCREEN
;
; Процедура очистки всего экрана
;
CLEAR_SCREEN    PROC      NEAR
    PUSH        AX
    PUSH        BX
    PUSH        CX
    PUSH        DX
    XOR         AL,AL      ;Очистить окно
    XOR         CX,CX      ;Верхний левый угол (0,0)
    MOV         DH,24      ;Нижняя строка экрана No 24
    MOV         DL,79      ;Правый край в 79 столбце
    MOV         BH,7       ;Применяются нормальные атрибуты очистки
    MOV         AH,6       ;Вызвать функцию SCROLL_UP
    INT         10h        ;Очистить окно
    POP         DX
    POP         CX
    POP         BX
    POP         AX
    RET
CLEAR_SCREEN    ENDP

PUBLIC         GOTO_XY
;
; Процедура перемещает курсор
; DH  Строка (Y)
; DL  Столбец (X)
;
GOTO_XY         PROC      NEAR
    PUSH        AX
    PUSH        BX
    MOV         BH,0       ;Нулевая страница
    MOV         AH,2       ;Вызов для SET CURSOR POSITION
```

```

    INT     10h
    POP     BX
    POP     AX
GOTO_XY    ENDP

PUBLIC     CURSOR_RIGHT
;
; Процедура перемещает курсор на одну позицию вправо или, если
; курсор в конце строки, на строку вверх
;
; Используется :      SEND_CRLF
;
CURSOR_RIGHT    PROC     NEAR
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     AH,3      ;Считать текущее положение курсора
    MOV     BH,0      ;Страница 0
    INT     10h      ;Считать позицию курсора
    MOV     AH,2      ;Установить новое положение курсора
    INC     DL        ;Установить новый столбец
    CMP     DL,79     ;Столбец меньше или равен 79?
    JBE     OK
    CALL    SEND_CRLF ;Перейти на новую строку
    JMP     DONE
OK: INT     10h
DONE:
    POP     DX
    POP     CX
    POP     BX
    POP     AX
    RET
CURSOR_RIGHT    ENDP

PUBLIC     CLEAR_TO_END_OF_LINE
;
; Процедура очищает строку с текущего положения курсора до
; конца строки
;
CLEAR_TO_END_OF_LINE    PROC     NEAR
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     AH,3      ;Считать текущую позицию курсора
    XOR     BH,BH     ;Страница 0
    INT     10h      ;Сейчас (X,Y) в DL,DH
    MOV     AH,6      ;Подготовка к очистке до конца строки
    XOR     AL,AL     ;Очистить окно
```

___DISK_IO.ASM___

```
MOV     CH,DH      ;Все на той же строке
MOV     CL,DL      ;Начать с позиции курсора
MOV     DL,79      ;Закончить в конце строки
MOV     BH,7       ;Применить нормальный атрибут
INT     10h
POP     DX
POP     CX
POP     BX
POP     AX
RET

CLEAR_TO_END_OF_LINE      ENDP

CODE_SEG ENDS

END

DISK_IO.ASM

CGROUP GROUP      CODE_SEG,      DATA_SEG
ASSUME CS:CGROUP, DS:CGROUP

CODE_SEG      SEGMENT      PUBLIC

      PUBLIC      READ_SECTOR
DATA_SEG      SEGMENT      PUBLIC
      EXTRN      SECTOR: BYTE
      EXTRN      DISK_DRIVE_NO: BYTE
      EXTRN      CURRENT_SECTOR_NO:WORD
DATA_SEG      ENDS
;
;Процедура считывает один сектор (512байт) в SECTOR
; Считываются:      CURRENT_SECTOR_NO, DISK_DRIVE_NO
;      Записывается: SECTOR
;
READ_SECTOR    PROC      NEAR
      PUSH      AX
      PUSH      BX
      PUSH      CX
      PUSH      DX
      MOV       AL,DISK_DRIVE_NO      ;Номер дисковод
      MOV       CX,1                  ;Считать только один сектор
      MOV       DX,CURRENT_SECTOR_NO ;Логический номер сектора
      LEA       BX,SECTOR             ;Сохранить сектор здесь
      INT       25h                   ;Считать сектор
      POPF      ;Сбросить флаг стека, установленный DOS
      POP       DX
      POP       CX
      POP       BX
```



```
        POP     AX
        RET
READ_SECTOR      ENDP

        PUBLIC  WRITE_SECTOR
;
; Процедура записывает один сектор обратно на диск
; Считываются:   DISK_DRIVE_NO, CURRENT_SECTOR_NO, SECTOR
;
WRITE_SECTOR     PROC     NEAR
    PUSH     AX
    PUSH     BX
    PUSH     CX
    PUSH     DX
    MOV      AL,DISK_DRIVE_NO      ;Номер дисководов
    MOV      CX,1                  ;Записать один сектор
    MOV      DX,CURRENT_SECTOR_NO  ;Логический сектор
    LEA      BX,SECTOR
    INT      26h                  ;Записать сектор на диск
    POPF                      ;Сбросить флаг информации
    POP      DX
    POP      CX
    POP      BX
    POP      AX
    RET
WRITE_SECTOR     ENDP

        PUBLIC  PREVIOUS_SECTOR
        EXTRN   INIT_SEC_DISP:NEAR,WRITE_HEADER:NEAR
        EXTRN   WRITE_PROMPT_LINE:NEAR

DATA_SEG        SEGMENT PUBLIC

        EXTRN   CURRENT_SECTOR_NO:WORD,
                EDITOR_PROMPT:BYTE

DATA_SEG        ENDS
;
; Процедура считывает, если это возможно, предыдущий сектор.
; Используются:  WRITE_HEADER, READ_SECTOR, INIT_SECDISP
;
;               WRITE_PROMPT_LINE
; Считываются:  CURRENT_SECTOR_NO, EDITOR_PROMPT
; Записывается: CURRENT_SECTOR_NO

PREVIOUS_SECTOR  PROC     NEAR
    PUSH     AX
    PUSH     DX
    MOV      AX,CURRENT_SECTOR_NO  ;Взять номер текущего сектора
    OR       AX,AX                 ;Не увеличивать, если 0
    JZ       DONT_DECREMENT_SECTOR
    DEC      AX

```

__DISK_IO.ASM__

```
MOV     CURRENT_SECTOR_NO,AX      ;Сохранить новый номер
                                       ;сектора
CALL    WRITE_HEADER
CALL    READ_SECTOR
CALL    INIT_SEC_DISP             ;Вывести новый сектор на
                                       ; экран
LEA     DX,EDITOR_PROMPT
CALL    WRITE_PROMPT_LINE

DONT_DECREMENT_SECTOR:
POP     DX
POP     AX
RET

PREVIOUS_SECTOR    ENDP

PUBLIC   NEXT_SECTOR
EXTRN    INIT_SEC_DISP:NEAR,WRITE_HEADER:NEAR
EXTRN    WRITE_PROMPT_LINE:NEAR

DATA_SEG SEGMENT PUBLIC
    EXTRN    CURRENT_SECTOR_NO:WORD
    EDITOR_PROMPT:BYTE
DATA_SEG    ENDS
;
;Считать следующий сектор.
; Используются:  WRITE_HEADER, READ_SECTOR, INIT_SEC_DISP
;
; Считываются:  CURRENT_SECTOR_NO, EDITOR_PROMPT
; Записывается: CURRENT_SECTOR_NO
NEXT_SECTOR PROC    NEAR
    PUSH    AX
    PUSH    DX
    MOV     AX,CURRENT_SECTOR_NO
    INC     AX                    ;Перейти к следующему сектору
    MOV     CURRENT_SECTOR_NO,AX
    CALL    WRITE_HEADER
    CALL    READ_SECTOR
    CALL    INIT_SEC_DISP        ;Вывести новый сектор на экран
    LEA     DX,EDITOR_PROMPT
    CALL    WRITE_PROMPT_LINE
    POP     DX
    POP     AX
    RET
NEXT_SECTOR ENDP

CODE_SEG    ENDS
END
```

```
DISPATCH.ASM

CGROUP    GROUP    CODE_SEG, DATA_SEG
          ASSUME    CS:CGROUP, DS:CGROUP

CODE_SEG    SEGMENT PUBLIC

PUBLIC     DISPATCHER
          EXTRN     READ_BYTE:NEAR, EDIT_BYTE:NEAR
          EXTRN     WRITE_PROMPT_LINE:NEAR
DATA_SEG    SEGMENT PUBLIC
          EXTRN     EDITOR_PROMPT:BYTE
DATA_SEG    ENDS

;
; Это центральный диспетчер. В режиме обычного редактирования
; и просмотра эта процедура считывает символ с клавиатуры и,
; если он является командой (например, клавиша управления
; курсором), DISPATCHER вызывает соответствующую процедуру
; для выполнения действия. Такое управление сделано для
; командных клавиш, список которых представлен в таблице
; DISPATCH_TABLE с именами процедур и их адресами
; расположения в памяти. Если символ не является командой, то он
; помещается непосредственно в буфер сектора - включается
; режим редактирования.
;   Используются:   READ_BYTE, EDIT_BYTE, WRITE_PROMPT_LINE
;   Считывается:   EDITOR_PROMPT
;
DISPATCHER PROC      NEAR
    PUSH    AX
    PUSH    BX
    PUSH    DX
DISPATCH_LOOP:
    CALL    READ_BYTE      ;Считать символ в AX
    OR      AH,AH          ;AH = 0, если символа нет
                                ;AH = -1 для расширенного кода.
    JZ      NO_CHARS_READ  ;Символ не считан, считать ещё раз
    JS      SPECIAL_KEY    ;Считан расширенный код
    MOV     DL,AL
    CALL    EDIT_BYTE      ;Считан обычный символ, редактировать байт
    JMP     DISPATCH_LOOP  ;Считать другой символ
SPECIAL_KEY:
    CMP     AL,68           ;F10-выход?
    JE      END_DISPATCH   ;Да, выйти
                                ;Применить BX для таблицы?
    LEA     BX,DISPATCH_TABLE
SPECIAL_LOOP:
    CMP     BYTE PTR [BX],0 ;Конец таблицы?
    JE      NOT_IN_TABLE    ;Да, команды в таблице нет
    CMP     AL,[BX]         ;Это вход таблицы?
```

__DISPATCH.ASM__

```
JE      DISPATCH      ;Да, управлять
ADD     BX,3           ;Нет, ещё раз
JMP     SPECIAL_LOOP   ;Проверить следующий вход
DISPATCH:
INC     BX             ;Отметить для адреса процедуры
CALL    WORD PTR[BX]   ;Вызвать процедуру
JMP     DISPATCH_LOOP  ;Ждать другой клавиши
NOT_IN_TABLE:          ;Ничего не делать, только считать
                        ;следующий символ
JMP     DISPATCH_LOOP
NO_CHARS_READ:
LEA     DX,EDITOR_PROMPT
CALL    WRITE_PROMPT_LINE ;Очистить любой напечатанный
                        ;неправильный символ
JMP     DISPATCH_LOOP  ;Повторить
END_DISPATCH:
POP     DX
POP     BX
POP     AX
RET
DISPATCHER ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
CODE_SEG      SEGMENT PUBLIC

EXTRN  NEXT_SECTOR:NEAR      ;B DISK_IO.ASM
EXTRN  PREVIOUS_SECTOR.NEAR  ;B DISK_IO.ASM
EXTRN  PHANTOM_UP:NEAR,PHANTOM_DOWN:NEAR ;B
                        ;PHANTOM.ASM

EXTRN  PHANTOM_LEFT:NEAR, PHANTOM_RIGHT:NEAR
EXTRN  WRITE_SECTOR:NEAR    ;B DISK_IO.ASM

CODE_SEG      ENDS

; Эта таблица содержит разрешенные расширенные ASCII коды и
; адреса процедур, выполняемых при нажатии соответствующих
; клавиш.
; Формат таблицы:
; DB      72
; Расширенный код перемещения курсора вверх
; DW      OFFSET CGROUP:PHANTOM_UP
;
DISPATCH_TABLE LABEL      BYTE
DB      59      ;F1
DW      OFFSET CGROUP:PREVIOUS_SECTOR
DB      60      ;F2
DW      OFFSET CGROUP:NEXT_SECTOR
DB      72      ;Курсор вверх
DW      OFFSET CGROUP:PHANTOM_UP
```

```
DB      80                      ;Курсор вниз
DW      OFFSET CGROUP:PHANTOM_DOWN
DB      75                      ;Курсор влево
DW      OFFSET CGROUP:PHANTOM_LEFT
DB      77                      ; Курсор вправо
DW      OFFSET CGROUP:PHANTOM_RIGHT
DB      88                      ;Шифт (shift) F5
DW      OFFSET CGROUP:WRITE_SECTOR
DB      0                      ;Конец таблицы

DATA_SEG ENDS

END
```

DISP_SEC.ASM

```
CGROUP    GROUP CODE_SEG, DATA_SEG ; Два сегмента
;                                     объединяются вместе
ASSUME     CS:CGROUP, DS:CGROUP
;
;Графические символы обрамления сектора
;
VERTICAL_BAR      EQU      0BAh
HORIZONTAL_BAR    EQU      0CDh
UPPER_LEFT        EQU      0C9h
UPPER_RIGHT       EQU      0BBh
LOWER_LEFT        EQU      0C8h
LOWER_RIGHT       EQU      0BCh
TOP_T_BAR         EQU      0CBh
BOTTOM_T_BAR      EQU      0CAh
TOP_TICK          EQU      0D1h
BOTTOM_TICK       EQU      0CFh

CODE_SEG    SEGMENT PUBLIC

PUBLIC      INIT_SEC_DISP
EXTRN       WRITE_PATTERN:NEAR, SEND_CRLF:NEAR
EXTRN       GOTO_XY:NEAR, WRITE_PHANTOM:NEAR

DATA_SEG    SEGMENT PUBLIC
EXTRN       LINES_BEFORE_SECTOR:BYTE
EXTRN       SECTOR_OFFSET:WORD
DATA_SEG    ENDS
;
; Процедура начинает вывод на экран половины сектора
; Используются:      WRITE_PATTERN, SEND_CRLF, DISP_HALF_SECTOR
; WRITE_TOP_HEX_NUMBERS, GOTO_XY,
; WRITE_PHANTOM
; Считываются:      TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN
; LINES_BEFORE_SECTOR
```

```

; Записывается:      SECTOR_OFFSET
;
INIT_SEC_DISP      PROC      NEAR

    PUSH      DX
    XOR       DL,DL                ;Сместить курсор в позицию
    MOV       DH,UNES_BEFORE_SECTOR
    CALL      GOTO_XY
    CALL      WRITE_TOP_HEX_NUMBERS
    LEA       DX,TOP_LINE_PATTERN
    CALL      WRITE_PATTERN
    CALL      SEND_CRLF
    XOR       DX,DX                ;Старт в начале сектора
    MOV       SECTOR_OFFSET,DX    ;Установить смещение сектора в 0
    CALL      DISP_HALF_SECTOR
    LEA       DX,BOTTOM_LINE_PATTERN
    CALL      WRITE_PATTERN
    CALL      WRITE_PHANTOM        ;Записать псевдокурсор
    POP       DX
    RET

INIT_SEC_DISP      ENDP

EXTRN WRITE_CHAR_N_TIMES:NEAR,WRITE_HEX:NEAR
EXTRN WRITE_CHAR:NEAR
EXTRN WRITE_HEX_DIGIT:NEAR, SEND_CRLF:NEAR

;Процедура пишет номер индекса (от 0 до F) в верхней части
;изображения половины сектора.

; Используются: WRITE_CHAR_N_TIMES, WRITE_HEX, WRITE_CHAR
;              WRITE_HEX_DIGIT, SEND_CRLF
WRITE_TOP_HEX_NUMBERS  PROC      NEAR

    PUSH      CX
    PUSH      DX
    MOV       DL,' '              ;Записать 9 пробелов с левой стороны
    MOV       CX,9
    CALL      WRITE_CHAR_N_TIMES
    XOR       DH,DH              ;Начать с 0
HEX_NUMBER_LOOP:
    MOV       DL,DH
    CALL      WRITE_HEX
    MOV       DL,' '
    CALL      WRITE_CHAR
    INC       DH
    CMP       DH,10h             ;Уже сделано?
    JB        HEX_NUMBER_LOOP
    MOV       DL,' '              ;Написать шестнадцатеричные номера
;                                над окном ASCII
    MOV       CX,2

```

```

        CALL    WRITE_CHAR_N_TIMES
        XOR     DL,DL
HEX_DIGIT_LOOP:
        CALL    WRITE_HEX_DIGIT
        INC     DL
        CMP     DL,10h
        JB      HEX_DIGIT_LOOP
        CALL    SEND_CRLF
        POP     DX
        POP     CX
        RET

WRITE_TOP_HEX_NUMBERS    ENDP
PUBLIC    DISP_HALF_SECTOR
EXTRN     SEND_CRLF:NEAR
;
;Процедура изображает половину сектора (256 byte)
;
; DS:DX Смещение в секторе в байтах, должно быть кратно 16.
; Uses:    DISP_LINE, SEND_CRLF
;
DISP_HALF_SECTOR    PROC    NEAR
        PUSH    CX
        PUSH    DX
        MOV     CX,16    ;Вывести 16 строк
HALF_SECTOR:
        CALL    DISP_LINE
        CALL    SEND_CRLF
        CALL    SEND_CRLF
        ADD     DX,16
        LOOP    HALF_SECTOR
        POP     DX
        POP     CX
        RET
DISP_HALF_SECTOR    ENDP

PUBLIC    DISP_LINE
EXTRN     WRITE_HEX:NEAR
EXTRN     WRITE_CHAR:NEAR
EXTRN     WRITE_CHAR_N_TIMS:NEAR
;
;Процедура изображает строку данных (16 байт), сначала в
;шестнадцатеричном виде, затем в ASCII.
;
; DS:DX Смещение в секторе в байтах.
;
; Используются:    WRITE_CHAR, WRITE_HEX, WRITE_CHAR_N_TIMS
; Считывается:    SECTOR
;
DISP_LINE    PROC    NEAR
        PUSH    BX

```

```
PUSH    CX
PUSH    DX
MOV     BX,DX           ;Смещение более полезно в BX
MOV     DL,' '
MOV     CX,3            ;Написать 2 пробела перед строкой
CALL    WRITE_CHAR_N_TIMES
;Написать смещение в шестнадцатеричном виде
CMP     BX,100h         ;Первая цифра 1?
JB      WRITE_ONE       ;Нет, пробел уже DL
MOV     DL,'1'          ;Да, поместить '1' в DL для вывода
WRITE_ONE:
CALL    WRITE_CHAR
MOV     DL,BL           ;Копировать младший байт DL для
;                               шестнадцатеричного вывода
;
CALL    WRITE_HEX
;
;                               Написать разделитель
MOV     DL,' '
CALL    WRITE_CHAR
MOV     DL,VERTICAL_BAR ;Нарисовать левую сторону рамки
CALL    WRITE_CHAR
MOV     DL,' '
CALL    WRITE_CHAR
;Сейчас запишем 16 байт для вывода
MOV     CX,16           ;Дампировать 16 байт
PUSH    BX
;Сохранить смещение для ASCII_LOOP
HEX_LOOP:
MOV     DL,SECTOR[BX]   ;Взять 1 байт
CALL    WRITE_HEX       ;Преобразовать (дампировать) этот байт
;                               в шестнадцатеричную форму
MOV     DL,' '          ;Написать пробел между числами
CALL    WRITE_CHAR
INC     BX
LOOP    HEX_LOOP
MOV     DL,VERTICAL_BAR ;Написать разделитель
CALL    WRITE_CHAR
MOV     DL,' '
CALL    WRITE_CHAR
MOV     CX,16
PUSH    BX              ;Вернуть смещение в SECTOR
ASCII_LOOP:
MOV     DL,SECTOR[BX]
CALL    WRITE_CHAR
INC     BX
LOOP    ASCII_LOOP
MOV     DL,' '
CALL    WRITE_CHAR
MOV     DL,VERTICAL_BAR ;Написать правую сторону рамки
CALL    WRITE_CHAR
POP     DX
```



```
        POP     CX
        POP     BX
        RET
DISP_LINE ENDP

CODE_SEG      ENDS

DATA_SEG SEGMENT PUBLIC
        EXTRN   SECTOR:BYTE

TOP_LINE_PATTERN LABEL      BYTE
        DB      ' ',7
        DB      UPPER_LEFT, 1
        DB      HORIZONTAL_BAR, 12
        DB      TOP_TICK, 1
        DB      HORIZONTAL_BAR, 11
        DB      TOP_TICK ,1
        DB      HORIZONTAL_BAR ,11
        DB      TOP_TICK, 1
        DB      HORIZONTAL_BAR, 12
        DB      TOP_T_BAR,1
        DB      HORIZONTAL_BAR, 18
        DB      UPPER_RIGHT,1
        DB      0
BOTTOM_LINE_PATTERN LABEL      BYTE
        DB      ' ',7
        DB      LOWER_LEFT, 1
        DB      HORIZONTAL_BAR,12
        DB      BOTTOM_TICK, 1
        DB      HORIZONTAL BAR,11
        DB      BOTTOM_TICK, 1
        DB      HORIZONTAL BAR,11
        DB      BOTTOM_TICK, 1
        DB      HORIZONTAL BAR,12
        DB      BOTTOM_T_BAR,1
        DB      HORIZONTAL BAR, 18
        DB      LOWER_RIGHT, 1
        DB      0
DATA_SEG ENDS

END
```

DSKPATCH.ASM

```
CGROUP      GROUP CODE_SEG, DATA_SEG
ASSUME      CS:CGROUP, DS:CGROUP

CODE_SEG      SEGMENT PUBLIC
```

```
ORG 100h
EXTRN CLEAR_SCREEN:NEAR, READ_SECTOR:NEAR
EXTRN INIT_SEC_DISP:NEAR, WRITE_HEADER:NEAR
EXTRN WRITE_PROMPT_LINE:NEAR, DISPATCHER:NEAR
DISK_PATCH PROC NEAR
CALL CLEAR_SCREEN
CALL WRITE_HEADER
CALL READ_SECTOR
CALL INIT_SEC_DISP
LEA DX, EDITOR_PROMPT
CALL WRITE_PROMPT_LINE
CALL DISPATCHER
INT 20h
DISK_PATCH ENDP

CODE_SEG ENDS

DATA_SEG SEGMENT PUBLIC

PUBLIC SECTOR_OFFSET
;
;SECTOR_OFFSET смещение половины изображаемого сектора в
;полном секторе. Оно должно быть кратно 16 и не превышать 256
;
SECTOR_OFFSET DW 0

PUBLIC CURRENT_SECTOR_NO, DISK_DRIVE_NO
CURRENT_SECTOR_NO DW 0 ;Начальный сектор 0
DISK_DRIVE_NO DB 0 ;Начальный дисковод A:

PUBLIC LINES_BEFORE_SECTOR, HEADER_LINE_NO

PUBLIC HEADER_PART_1, HEADER_PART_2
;
;LINES_BEFORE_SECTOR число линий в верхней части экрана перед
;изображением половины сектора
;
LINES_BEFORE_SECTOR DB 2
HEADER_LINE_NO DB 0
HEADER_PART_1 DB 'Диск ',0
HEADER_PART_2 DB ' Сектор ',0
PUBLIC PROMPT_LINE_NO, EDITOR_PROMPT
PROMPT_LINE_NO DB 21
EDITOR_PROMPT DB 'Нажмите функциональную клавишу, или ENTER'
DB ' символ или байт: ',0
PUBLIC SECTOR
```

Листинг DSKPATCH

```
;
; Полный сектор (до 8192 bytes) записан в этой части памяти
;
SECTOR      DB      8192 DUP (0)

DATA_SEG    ENDS

        END DISK_PATCH
```

EDITOR.ASM

```
CGROUP     GROUP     CODE_SEG, DATA_SEG
ASSUME      CS:CGROUP , DS:CGROUP

CODE_SEG    SEGMENT PUBLIC

DATA_SEG    SEGMENT PUBLIC
EXTRN       SECTOR:BYTE
EXTRN       SECTOR_OFFSET:WORD
EXTRN       PHANTOM_CURSOR_X:BYTE
EXTRN       PHANTOM_CURSOR_Y:BYTE
DATA_SEG    ENDS
;
; Процедура записывает один байт в SECTOR, по адресу,
; соответствующему положению псевдокурсора на изображении
; сектора
;
;     DL      Байт, записываемый SECTOR
;
;     Смещение вычисляется как:
;     OFFSET = SECTOR_OFFSET + (16*PH ANTOM_CURSOR_Y) +
;     PHANTOM_CURSOR_X
;
;     Считываются: PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y,
; SECTOR_OFFSET
;     Записывается: SECTOR
;
WRITE_TO_MEMORY  PROC     NEAR
    PUSH        AX
    PUSH        BX
    PUSH        CX
    MOV         BX, SECTOR_OFFSET
    MOV         AL, PHANTOM_CURSOR_Y
    XOR         AH, AH
    MOV         CL, 4          ; Умножить PHANTOM_CURSOR_Y на 16
    SHL         AX, CL
    ADD         BX, AX         ; BX = SECTOR-OFFSET + (16*Y)
    MOV         AL, PHANTOM_CURSOR_X
    XOR         AH, AH
```

```

    ADD     BX,AX           ;Это адрес!
    MOV     SECTOR[BX],DL   ;Сейчас запишем байт
    POP     CX
    POP     BX
    POP     AX
    RET

WRITE_TO_MEMORY      ENDP

PUBLIC EDIT_BYTE

EXTRN      SAVE_REAL_CURSOR:NEAR,
EXTRN      RESTORE_REAL_CURSOR:NEAR
EXTRN      MOV_TO_HEX_POSITION:NEAR,
EXTRN      MOV_TO_ASCII_POSITION:NEAR
EXTRN      WRITE_PHANTOM:NEAR, WRITE_PROMPT_LINE:NEAR
EXTRN      CURSOR_RIGHT:NEAR,WRITE_HEX:NEAR
EXTRN      WRITE_CHAR:NEAR

DATA_SEG SEGMENT PUBLIC
EXTRN      EDITOR_PROMPT:BYTE
DATA_SEG ENDS
;
; Процедура изменяет байт в памяти и на экране
; DL Байт, записываемый в SECTOR, и изменяемый на экране
; Используются: SAVE_REAL_CURSOR, RESTORE_REAL_CURSOR
; MOV_TO_HEX_POSITION, MOV_TO_ASCII_POSITION
; WRITE_PHANTOM, WRITE_PROMPT_LINE,
; CURSOR_RIGHT WRITE_HEX, WRITE_CHAR,
; WRITE_TO_MEMORY
; Считывается: EDITOR_PROMPT
;
EDIT_BYTE PROC NEAR
    PUSH    DX
    CALL    SAVE_REAL_CURSOR      ;Сместить к шестнадцатеричному
                                   числу
;
    CALL    MOV_TO_HEX_POSITION   ;в шестнадцатеричном окне
    CALL    CURSOR_RIGHT          ;Записать новое число
    CALL    MOV_TO_ASCII_POSITION ;Сместить к символу
                                   в окне ASCII
;
    CALL    WRITE_CHAR            ;Записать новый символ
    CALL    RESTORE_REAL_CURSOR   ;Вернуть курсор на место
    CALL    WRITE_PHANTOM         ;Переписать псевдокурсор
    CALL    WRITE_TO_MEMORY       ;Сохранить новый байт SECTOR
    LEA     DX, EDITOR_PROMPT
    CALL    WRITE_PROMPT_LINE
    POP     DX
    RET
EDIT_BYTE ENDP

CODE_SEG ENDS

END

```

KBD_IO.ASM

```

CGROUP      GROUP      CODE_SEG, DATA_SEG
ASSUME      CS:CGROUP, DS:CGROUP

        BS      EQU      8      ;Символ "Забой (backspace)"
        CR      EQU      13     ;Символ "возврат каретки"
        ESC     EQU      27     ;Символ "Escape"

CODE_SEG    SEGMENT PUBLIC

PUBLIC      STRING_TO_UPPER
;
; Процедура преобразует строку, применяя формат
; DOS ко всем заглавным буквам
;      DS:DX      Адрес буфера строки
;
STRING_TO_UPPER    PROC    NEAR
        PUSH     AX
        PUSH     BX
        PUSH     CX
        MOV      BX,DX
        INC      BX      ;Отметить счетчик символов
        MOV      CL,[BX]  ;Считать символы во 2 байте буфера
        XOR      CH,CH    ;Очистить старший байт счетчика
UPPER_LOOP:
        INC      BX      ;Отметить следующий символ в буфере
        MOV      AL,[BX]
        CMP      AL,'a'   ;Это заглавная буква?
        JB       NOT_LOWER ;Нет оператора
        CMP      AL,'z'
        JA       NOT_LOWER
        ADD      AL,'A'-'a' ;Преобразовать в заглавную
        MOV      [BX],AL
NOT_LOWER:
        LOOP     UPPER_LOOP
        POP      CX
        POP      BX
        POP      AX
        RET
STRING_TO_UPPER    ENDP
;
;Процедура преобразует символ из ASCII (hex) в тетраду (4 бита).
;      AL Преобразуемый символ
; Возвращаются      AL Тетрада
;      CF Установлен при ошибке, сброшен, если

```

```

;                                ошибки нет
CONVERT_HEX_DIGIT  PROC  NEAR
    CMP     AL, '0'              ;Разрешённая цифра?
    JB      BAD_DIGIT           ;Нет оператора
    CMP     AL, '9'              ;ещё не уверен
    JA      TRY_HEX             ;Может быть шестнадцатеричная
    SUB     AL, '0'              ;Если десятичная, преобразовать в тетраду
;                                (4 бита)
    CLC     ;Сбросить флаг, ошибки нет
    RET
TRY_HEX:
    CMP     AL, 'A'              ;Не уверен
    JB      BAD_DIGIT           ;Не шестнадцатеричная
    CMP     AL, 'F'              ;ещё не уверен
    JA      BAD_DIGIT           ;Не шестнадцатеричная
    SUB     AL, 'A'-10           ;Если шестнадцатеричная, преобразовать в
;                                тетраду
    CLC     ;Сбросить флаг, ошибки нет
    RET
BAD_DIGIT:
    STC     ;Установить флаг, ошибка
    RET
CONVERT_HEX_DIGIT  ENDP

PUBLIC  HEX_TO_BYTE
;
; Процедура преобразует два символа в DS:DX из
; шестнадцатеричного вида в один байт
;
; DS:DX    Адрес  двух символов для преобразования
; Возвращаются:
; AL        Байт
; CF        Установлен при ошибке, сброшен, если
; ошибки нет
;
;    Используется:      CONVERT_HEX_DIGIT
;
HEX_TO_BYTE  PROC  NEAR
    PUSH    BX
    PUSH    CX
    MOV     BX, DX              ;Поместить адрес в BX для не прямой
;                                адресации
    MOV     AL, [BX]            ;Взять первую цифру
    CALL    CONVERT_HEX_DIGIT
    JC      BAD_HEX            ;Цифра плохая, если флаг установлен
    MOV     CX, 4               ;Умножить на 16
    SHL     AL, CL
    MOV     AH, AL              ;Сохранить копию

```

```
INC     BX                ;Взять вторую цифру
MOV     AL,[BX]
CALL    CONVERT_HEX_DIGIT
JC      BAD_HEX          ;Цифра плохая, если флаг установлен
OR      AL,AH            ;Объединить две тетрады
CLC     ;Сбросить флаг, ошибки нет
DONE_HEX:
POP     CX
POP     BX
RET
BAD_HEX:
STC     ;Установить флаг, ошибка
JMP     DONE_HEX
HEX_TO_BYTE    ENDP

PUBLIC READ_STRING
EXTRN WRITE_CHAR:NEAR
;
;Процедура преобразует функцию почти аналогично DOS 0Ah.
;Отличие в том, что при нажатии функциональной клавиши будет
;возвращаться специальный символ. Esc - уничтожает сделанный
;ввод.
;
; DS:DX    Адрес буфера клавиатуры. Первый байт должен
;          содержать максимальное число считываемых символов
;          (плюс один для возврата). Второй байт будет
;          использоваться этой процедурой для возврата
;          действительного числа считанных символов:
;          0      Нет считанных символов
;          -1     считан специальный символ, в другом случае
;                  число считанных символов (без ENTER)
; Используются:    BACK_SPACE, WRITE_CHAR
;
READ_STRING    PROC    NEAR
PUSH    AX
PUSH    BX
PUSH    SI
MOV     SI,DX            ;SI применяется в качестве регистра индекса
START_OVER:
MOV     BX,2             ;BX для ввода без проверки
MOV     AH,7             ;Вызов для ввода без проверки
INT     21h             ;для CTRL-BREAK и без вывода
OR      AL,AL            ;Символ расширенный ASCII?
JZ      EXTENDED        ;Да, считать расширенным символ
NOT_EXTENDED:
;Расширенный символ ошибка пока буфер пуст
CMP     AL,CR            ;Это возврат каретки?
JE      END_INPUT        ;Да мы сделали ввод
CMP     AL,BS            ;Это символ пробела?
JNE     NOT_BS           ;Нет оператора
```

```

CALL    BACK_SPACE      ;Да, уничтожить символ
CMP     BL,2             ;Буфер пуст?
JE      START_OVER      ;Да, можем считать расширенный символ
;
;      again (опять/снова/ещё)
JMP     SHORT READ_NEXT_CHAR ;Нет, продолжаем нормальное
;      считывание
NOT_BS:
CMP     AL,ESC           ;Это очистка буфера через ESC?
JE      PURGE_BUFFER     ;Да, очистить буфер
CMP     BL,[SI]          ;Проверить, полон ли буфер
JA      BUFFER_FULL      ;Буфер полон
MOV     [SI+BX],AL       ;Записать ещё символ в буфер
INC     BX               ;Отметить следующий свободный символ в
;      буфере
PUSH    DX
MOV     DL,AL            ;Отразить символ на экран
CALL    WRITE_CHAR
POP     DX
READ_NEXT_CHAR:
MOV     AH,7
INT     21h
OR      AL,AL            ;Все расширенные ASCII символы
;      неправильные, если буфер не пуст
JNE     NOT_EXTENDED     ;Символ правильный
MOV     AH,7
INT     21h              ;Вывести расширенный символ
;
; Сообщает о наличии ошибки звуковым сигналом, посылая символ
; chr$(7)
;
SIGNAL_ERROR:

PUSH    DX
MOV     DL,7             ;Включить сигнал chr$(7)
MOV     AH,2
INT     21h
POP     DX
JMP     SHORT READ_NEXT_CHAR ;Считать следующий символ
;
; Очистить буфер строки и удалить все символы, выведенные на
; экран
;
PURGE_BUFFER:

PUSH    CX
MOV     CL,[SI]          ;Очистить число считанных символов
XOR     CH,CH            ; (?) [pause]
PURGE_LOOP:              ; в буфере.
CALL    BACK_SPACE
LOOP    PURGE_LOOP
POP     CX
JMP     START_OVER       ;Сейчас, пока буфер пуст, можно считать

```



```
;    расширенный ASCII символ
;
;Буфер был полон, поэтому не можем считать символ.
;Сигнализировать пользователю звуковым сигналом о полном
;буфере.
;
BUFFER_FULL:
    JMP     SHORT SIGNAL_ERROR    ;Если буфер полон - сигнал
;
;Считать расширенный ASCII-код и поместить его в буфер как
;символ, затем вернуть -1 как номер считанного символа.
;
EXTENDED:    ;Считать расширенный ASCII код
    MOV     AH,7
    INT     21h
    MOV     [SI + 2],AL    ;Поместить символ в буфер
    MOV     BL,0FFh        ;Число считанных символов = -1 для
                           ;специальных
;
JMP SHORT END_STRING
;
; Сохранить число считанных символов и вернуть.
;
END_INPUT:    ;Сделано с вводом
    SUB     BL,2           ;Счетчик считанных символов
END_STRING:
    MOV     [SI + 1],BL    ;Вернуть число считанных символов
    POP     SI
    POP     BX
    POP     AX
    RET
READ_STRING   ENDP

PUBLIC    READ_BYTE
;
; Процедура считывает ASCII символ шестнадцатеричного числа.
; Возвращает байт в AL  Код символа (пока AH = 0)
;
;             AH  1 если считан ASCII символ или шести, число,
;             0  если символ не считан,
;             -1  если считан спец. клавиша.
; Используются: HEX_TO_BYTE-STRING_TO_UPPER, READ_STRING
; Читаются:    KEYBOARD_INPUT, и т.д.
; Записываются: KEYBOARD_INPUT, и т.д.
;
READ_BYTE    PROC    NEAR
    PUSH     DX
    MOV     CHAR_NUM_UMIT,3    ;Только два символа (плюс Enter)
    LEA     DX,KEYBOARD_INPUT
    CALL    READ_STRING
    CMP     NUM_CHARS_READ,1    ;Много символов
```

```
JE      ASCII_INPUT      ;Только один ASCII символ
JB      NO_CHARACTERS    ;Нажат только Enter
CMP     BYTE PTR NUM_CHARS_READ,0FFh      ; Функциональная
;                                     клавиша?

JE      SPECIAL_KEY      ;Да
CALL    STRING_TO_UPPER  ;Нет, преобразовать в заглавные
LEA     DX, CHARS         ;Адрес строки для преобразования
CALL    HEX_TO_BYTE      ;Преобразовать строку из шестнадц.
;                                     в байт

JC      NO_CHARACTERS    ;Ошибка, вернуть 'символ не считан'
MOV     AH,1              ;Сигнал, считан один символ
DONE_READ:
POP     DX
RET

NO_CHARACTERS:
XOR     AH,AH             ;Установить 'символ не считан'
JMP     DONE_READ

ASCII_INPUT:
MOV     AL,CHARS          ;Загрузить считанный символ
MOV     AH,1              ;Сигнал, считан один символ
JMP     DONE_READ

SPECIAL_KEY:
MOV     AL,CHARS[0]       ;Вернуть скэн код
MOV     AH,0FFh           ;Спец. клавиша, -1
JMP     DONE_READ

READ_BYTE ENDP

PUBLIC READ_DECIMAL
;
; Процедура преобразует результат READ_STRING (строка
; десятичных цифр) в слово
;
; AX Слово, преобразованное из десятичных
; CF Установлен при ошибке, сброшен при отсутствии ошибки
;
; Используется: READ_STRING
; Читаются:   KEYBOARD_INPUT, и т.д.
; Записываются: KEYBOARD_INPUT, и т.д.
;
READ_DECIMAL PROC NEAR
PUSH    BX
PUSH    CX
PUSH    DX
MOV     CHAR_NUM_LIMIT,6 ;Максимальное число (65535)
LEA     DX, KEYBOARD_INPUT
CALL    READ_STRING
MOV     CL,NUM_CHARS_READ ;Взять число считанных символов
XOR     CH,CH             ;Установить старший байт счетчика в 0
CMP     CL,0              ;Вернуть ошибку, если считанных символов
```

```

;нет
    JLE     BAD_DECIMAL_DIGIT    ;Символ не считан, ошибка
    XOR     AX,AX                ;Старт с числа 0
    XOR     BX,BX                ;Старт с начала строки
CONVERT_DIGIT:
    MOV     DX,10                ;Умножить число на 10
    MUL     DX                    ;Умножить AX на 10
    JC      BAD_DECIMAL_DIGIT    ; Установить CF (переполнение) если
;                                MUL переполняет одно слово
    MOV     DL,CHARS[BX]         ;Взять следующую цифру
    SUB     DL,'0'                ;И преобразовать а тетраду (4 бита)
    JS      BAD_DECIMAL_DIGIT    ;Если меньше 0, то плохая цифра
    CMP     DL,9                 ;Цифра плохая?
    JA      BAD_DECIMAL_DIGIT    ;Да
    ADD     AX,DX                ;Нет, добавить её к числу
    INC     BX                    ;Отметить следующий символ
    LOOP    CONVERT_DIGIT        ;Взять следующую цифру
DONE_DECIMAL:
    POP     DX
    POP     CX
    POP     BX
    RET
BAD_DECIMAL_DIGIT:
    STC                                ;Установить флаг для сигнала об ошибке
    JMP     DONE_DECIMAL
READ_DECIMAL ENDP

PUBLIC     BACK_SPACE
EXTRN      WRITE_CHAR:NEAR
;
; Процедура уничтожает символы из буфера и экрана, по одному
; каждый раз, если буфер не пуст.
;
;   DS:SI + BX    Самые первые символы в буфере
;
;   Uses:         WRITE_CHAR
;
BACK_SPACE  PROC NEAR    ;Очистить один символ
    PUSH    AX
    PUSH    DX
    CMP     BX,2          ;Буфер пуст?
    JE      END_BS        ;Да, считать следующий символ
    DEC     BX            ;Удалить один символ из буфера
    MOV     AH,2          ;Удалить символ с экрана
    MOV     DL,BS
    INT     21h
    MOV     DL,20h        ;Записать пробел
    CALL    WRITE_CHAR
    MOV     DL,BS        ;Вернуться

```

```
        INT      21h
END_BS:
        POP      DX
        POP      AX
        RET
BACK_SPACE  ENDP

CODE_SEG  ENDS

DATA_SEG SEGMENT PUBLIC
KEYBOARD_INPUT LABEL BYTE
CHAR_NUM_LIMIT DB 0 ;Длина входного буфера
NUM_CHARS_READ DB 0 ;Число считанных символов
CHARS DB 80 DUP (0) ;Буфер клавиатурного ввода
DATA_SEG ENDS

END

PHANTOM.ASM

CGROUP GROUP CODE_SEG, DATA_SEG
ASSUME CS:CGROUP, DS:CGROUP

CODE_SEG SEGMENT PUBLIC

        PUBLIC MOV_TO_HEX_POSITION
        EXTRN GOTO_XY:NEAR
DATA_SEG SEGMENT PUBLIC
        EXTRN LINES_BEFORE_SECTOR:BYTE
DATA_SEG ENDS
;
; Процедура перемещает реальный курсор в позицию
; псевдокурсора в шестнадцатеричном окне
; Используется: GOTO_XY
; Считываются: LINES_BEFORE_SECTOR, PHANTOM_CURSOR_X,
; PHANTOM_CURSOR_Y
;
MOV_TO_HEX_POSITION PROC NEAR
        PUSH     AX
        PUSH     CX
        PUSH     DX
        MOV      DH, LINES_BEFORE_SECTOR ;Найти строку псевдокурсора (0,0)
        ADD      DH, 2 ;Плюс строка шестн. цифр и рамка
        ADD      DH, PHANTOM_CURSOR_Y ;DH = строка псевдокурсора
        MOV      DL, 8 ;Отступ слева
        MOV      CL, 3 ;Каждый столбец в три символа,
        MOV      AL, PHANTOM_CURSOR_X ;поэтому умножить CURSOR_X
;                                     на 3

```

```
MUL      CL
ADD      DL,AL      ;И добавить к значению отступа, для
;                получения столбца псевдокурсора
CALL     GOTO_XY
POP      DX
POP      CX
POP      AX
RET
MOV_TO_HEX_POSITION      ENDP

PUBLIC    MOV_TO_ASCII_POSITION
EXTRN     GOTO_XY:NEAR
DATA_SEG SEGMENT PUBLIC
EXTRN     LINES_BEFORE_SECTOR:BYTE
DATA_SEG ENDS
;
; Процедура перемещает реальный курсор в начало
; псевдокурсора в окне ASCII
;   Используется: GOTO_XY
;   Считываются:   LINES_BEFORE_SECTOR, PHANTOM_CURSOR_X,
;                 PHANTOM_CURSOR_Y
;
MOV_TO_ASCII_POSITION      PROC    NEAR
    PUSH    AX
    PUSH    DX
    MOV     DH,LINES_BEFORE_SECTOR ;Найти строку псевдокурсора (0,0)
    ADD     DH,2      ;Плюс строка шести, чисел и горизонтальная
;                рамка
    ADD     DH,PHANTOM_CURSOR_Y    ;DH = строка псевдокурсора
    MOV     DL,59      ;Отступ слева
    ADD     DL,PHANTOM_CURSOR_X    ;Прибавить CURSOR_X для
;                того, чтобы получить координату X
    CALL    GOTO_XY    ; положения псевдокурсора
    POP     DX
    POP     AX
    RET
MOV_TO_ASCII_POSITION      ENDP

PUBLIC    SAVE_REAL_CURSOR
;
; Процедура записывает позицию псевдокурсора в двух
; переменных
; REAL_CURSOR_X и REAL_CURSOR_Y
;
; Записываются: REAL_CURSOR_X, REAL_CURSOR_Y
;
SAVE_REAL_CURSOR      PROC    NEAR
    PUSH    AX
```

```
PUSH    BX
PUSH    CX
PUSH    DX
MOV     AH,3           ;Считать позицию курсора
XOR     BH,BH          ;на странице 0
INT     10h            ;и вернуть в DL,DH
MOV     REAL_CURSOR_Y,DL ;Сохранить позицию
MOV     REAL_CURSOR_X,DH
POP     DX
POP     CX
POP     BX
POP     AX
RET
SAVE_REAL_CURSOR      ENDP

PUBLIC   RESTORE_REAL_CURSOR
EXTRN    GOTO_XY:NEAR
;
; Процедура восстанавливает реальный курсор в прежней позиции,
; записанной в
;   REAL_CURSOR_X and REAL_CURSOR_Y.
;
; Используется: GOTO_XY
; Читается:     REAL_CURSOR_X, REAL_CURSOR_Y
;
RESTORE_REAL_CURSOR   PROC    NEAR
    PUSH    DX
    MOV     DL,REAL_CURSOR_Y
    MOV     DH,REAL_CURSOR_X
    CALL    GOTO_XY
    POP     DX
    RET
RESTORE_REAL_CURSOR   ENDP

PUBLIC   WRITE_PHANTOM
EXTRN    WRITE_ATTRIBUTE_N_TIMES:NEAR
;
; Процедура использует CURSOR_X и CURSOR_Y, через
; MOV_TO_...., как
; координаты псевдокурсора. WRITE_PHANTOM записывает
; псевдокурсор.
;
; Используются:   WRITE_ATTRIBUTE_N_TIMES, SAVE_REAL_CURSOR
; RESTORE_REAL_CURSOR, MOV_TO_HEX_POSITION,
; MOV_TO_ASCII_POSITION
WRITE_PHANTOM   PROC    NEAR
    PUSH    CX
```

```
PUSH    DX
CALL    SAVE_REAL_CURSOR
CALL    MOV_TO_HEX_POSITION      ;Координаты курсора в
;                                шестнадц. окне
MOV     CX,4      ;Сделать псевдокурсор 4 символа шириной
MOV     DL,70h
CALL    WRITE_ATTRIBUTE_N_TIMES
CALL    MOV_TO_ASCII_POSITION    ;Координаты курсора в ASCII ;окне
MOV     CX,1      ;Ширина курсора 1 символ
CALL    WRITE_ATTRIBUTE_N_TIMES
CALL    RESTORE_REAL_CURSOR
POP     DX
POP     CX
RET
WRITE_PHANTOM ENDP

PUBLIC  ERASE_PHANTOM
EXTRN  WRITE_ATTRIBUTE_N_TIMES_NEAR
;
; Процедура удаляет псевдокурсор, противоположна процедуре
; WRITE_PHANTOM.
;
; Используются: WRITE_ATTRIBUTE_N_TIMES, SAVE_REAL_CURSOR
;               RESTORE_REAL_CURSOR, MOV_TO_HEX_POSITION
;               MOV_TO_ASCII_POSITION
;
ERASE_PHANTOM PROC    NEAR
    PUSH    CX
    PUSH    DX
    CALL    SAVE_REAL_CURSOR
    CALL    MOV_TO_HEX_POSITION    ;Координаты курсора в
;                                шестнадцат. окне
MOV     CX,4      ;Изменить черный на белый
MOV     DL,7
CALL    WRITE_ATTRIBUTE_N_TIMES
CALL    MOV_TO_ASCII_POSITION
MOV     CX,1
CALL    WRITE_ATTRIBUTE_N_TIMES
CALL    RESTORE_REAL_CURSOR
POP     DX
POP     CX
RET
ERASE_PHANTOM ENDP
;
; Четыре процедуры перемещения курсора
;
; Используются:      ERASE_PHANTOM, WRITE_PHANTOM
;
```

___PHANTOM.ASM___

```
;          SCROLL_DOWN, SCROLL_UP
;  Считываются:   PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
;  Записываются:   PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
;
PUBLIC PHANTOM_UP
PHANTOM_UP PROC NEAR
    CALL    ERASE_PHANTOM          ;Очистить позицию курсора
    DEC     PHANTOM_CURSOR_Y       ;На строку вверх
    JNS     WASNT_AT_TOP           ;Если не верх окна, записать
    CALL    SCROLL_DOWN           ;Если на верху, оставить на месте
WASNT_AT_TOP:
    CALL    WRITE_PHANTOM         ;Записать псевдокурсор
    RET
PHANTOM_UP ENDP

PUBLIC PHANTOM_DOWN
PHANTOM_DOWN PROC NEAR
    CALL    ERASE_PHANTOM          ;Очистить текущую позицию
    INC     PHANTOM_CURSOR_Y       ;Переместить курсор на строку вниз
    CMP     PHANTOM_CURSOR_Y,16   ;Это низ окна?
    JB      WASNT_AT_BOTTOM       ;Нет, записать псевдокурсор
    CALL    SCROLL_UP             ;Низ окна, оставить здесь
WASNT_AT_BOTTOM:
    CALL    WRITE_PHANTOM         ;Записать псевдокурсор
    RET
PHANTOM_DOWN ENDP

    PUBLIC PHANTOM_LEFT
PHANTOM_LEFT PROC NEAR
    CALL    ERASE_PHANTOM          ;Очистить текущую позицию
    DEC     PHANTOM_CURSOR_X       ;Сместить курсор на столбец влево
    JNS     WASNT_AT_LEFT         ;Если не у левого края, записать
    MOV     PHANTOM_CURSOR_X,0    ;Если левый край, оставить здесь
WASNT_AT_LEFT:
    CALL    WRITE_PHANTOM         ;Записать позицию псевдокурсора
    RET
PHANTOM_LEFT ENDP

PUBLIC PHANTOM_RIGHT
PHANTOM_RIGHT PROC NEAR
    CALL    ERASE_PHANTOM          ;Очистить текущую позицию
    INC     PHANTOM_CURSOR_X       ;Сместить на столбец вправо
    CMP     PHANTOM_CURSOR_X,16   ;Уже правый край?
    JB      WASNT_AT_RIGHT        ;Если правый край, оставить здесь
WASNT_AT_RIGHT:
    CALL    WRITE_PHANTOM         ;Записать псевдокурсор
    RET
PHANTOM_RIGHT ENDP
```



```

EXTRN DISP_HALF_SECTOR:NEAR, GOTO_XY:NEAR
DATA_SEG SEGMENT PUBLIC
EXTRN SECTOR_OFFSET:WORD
EXTRN LINES_BEFORE_SECTOR:BYTE
DATA_SEG ENDS
;
; Две процедуры перемещения между изображениями половин
; сектора
; Используются: WRITE_PHANTOM, DISP_HALF_SECTOR,
; ERASE_PHANTOM, GOTO_XY
; SAVE_REAL_CURSOR, RESTORE_REAL_CURSOR
; Читаются: LINES_BEFORE_SECTOR
; Записываются: SECTOR_OFFSET, PHANTOM_CURSOR_Y
;
SCROLL_UP PROC NEAR
    PUSH DX
    CALL ERASE_PHANTOM ;Удалить псевдокурсор
    CALL SAVE_REAL_CURSOR ;Сохранить позицию реального
; курсора
    XOR DL,DL ;Установить курсор для
; изображения половины сектора
    MOV DH,LINES_BEFORE_SECTOR
    ADD DH,2
    CALL GOTO_XY
    MOV DX,256 ;Вывести вторую половину сектора
    MOV SECTOR_OFFSET,DX
    CALL DISP_HALF_SECTOR
    CALL RESTORE_REAL_CURSOR ;Восстановить позицию реального
; курсора
    MOV PHANTOM_CURSOR_Y,0 ;Курсор вверху второй
; половины сектора
    CALL WRITE_PHANTOM ;Восстановить псевдокурсор
    POP DX
    RET
SCROLL_UP ENDP

SCROLL_DOWN PROC NEAR
    PUSH DX
    CALL ERASE_PHANTOM ;Удалить псевдокурсор
    CALL SAVE_REAL_CURSOR ;Сохранить позицию реального
; курсора
    XOR DL,DL ;Установить курсор для изображения
; половины сектора
    MOV DH,LINES_BEFORE_SECTOR
    ADD DH,2
    CALL GOTO_XY
    XOR DX,DX ;Вывести на экран первую половину
; сектора

```

___VIDEO_IO.ASM___

```
MOV SECTOR_OFFSET,DX
CALL DISP_HALF_SECTOR
CALL RESTORE_REAL_CURSOR ;Восстановить позицию
;реального курсора
MOV PHANTOM_CURSOR_Y,15 ;Курсор внизу первой половины
;сектора
CALL WRITE_PHANTOM ;Восстановить псевдокурсor
POP DX
RET
SCROLL_DOWN ENDP

CODE_SEG ENDS

DATA_SEG SEGMENT PUBLIC
REAL_CURSOR_X DB 0
REAL_CURSOR_Y DB 0
PUBLIC PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
PHANTOM_CURSOR_X DB 0
PHANTOM_CURSOR_Y DB 0

DATA_SEG ENDS

END
```

VIDEO_IO.ASM

```
VIDEO_IO.ASM
CGROUP GROUP CODE_SEG, DATA_SEG
ASSUME CS:CGROUP, DS:CGROUP
CODE_SEG SEGMENT PUBLIC
ORG 100h

PUBLIC WRITE_HEX
;
;Процедура преобразует байт в DL в шестнадцат. форму и
;записывает две цифры в текущей позиции курсора.
; DL Байт, преобразуемый в шестнадцат. форму
; Используется: WRITE_HEX_DIGIT
;
WRITE_HEX PROC NEAR ;Точка входа
PUSH CX ;Сохранить регистры, применяемые в процедуре
PUSH DX
MOV DH,DL ;Сделать копию байта
MOV CX,4 ;Взять старшие 4 бита в DL
SHR DL,CL
CALL WRITE_HEX_DIGIT ;Вывести первую шестнадц. цифру
MOV DL,DH ;Взять младшие 4 бита DL
AND DL,0Fh ;Удалить 4 старших бита
CALL WRITE_HEX_DIGIT ;Вывести вторую шестнадц. цифру
POP DX
```

```

        POP        CX
        RET
WRITE_HEX  ENDP

PUBLIC    WRITE_HEX_DIGIT
;
; Процедура преобразует 4 младших бита DL в шестнадцат. цифру
; и выводит её на экран.
; DL Младшие 4 бита содержат число, выводимое в шестнадцат.
; форме
; Используется:  WRITE_CHAR
;
WRITE_HEX_DIGIT PROC NEAR
        PUSH      DX                      ;Сохранить используемый регистр
        CMP       DL,10                  ;Тетрада меньше 10?
        JAE       HEX_LETTER             ;Нет, преобразовать в букву
        ADD       DL,'0'                 ;Да, преобразовать в цифру
        JMP       Short WRITE_DIGIT      ;Записать этот символ
HEX_LETTER:
        ADD       DL,"A"-10              ;Преобразовать в шестнадцат. букву
WRITE_DIGIT:
        CALL      WRITE_CHAR             ;Вывести букву на экран
        POP       DX                    ;Восстановить старое значение DX
        RET
WRITE_HEX_DIGIT ENDP

PUBLIC    WRITE_CHAR
EXTRN     CURSOR_RIGHT:NEAR
;
; Процедура выводит символ на экран, применяя подпрограммы
; ROM BIOS, поэтому все выводимые символы представляются на
; экране (даже такие как пробел).
; Процедура должна выполнить подготовительную работу по
; подготовке позиции курсора.
;
; DL Байт, выводимый на экран
;
; Используется:  CURSOR_RIGHT
;
WRITE_CHAR  PROC    NEAR
        PUSH      AX
        PUSH      BX
        PUSH      CX
        PUSH      DX
        MOV       AH,9                  ;Вызов для вывода символов/атрибутов
        MOV       BH,0                  ;Установить страницу 0
        MOV       CX,1                  ;Записать только один символ
        MOV       AL,DL                 ;Записываемый символ

```

___VIDEO_IO.ASM___

```
MOV     BL,7           ;Нормальный атрибут
INT     10h           ;Записать символ и атрибут
CALL    CURSOR_RIGHT   ;Перейти к следующей позиции курсора
POP     DX
POP     CX
POP     BX
POP     AX
RET

WRITE_CHAR    ENDP

PUBLIC  WRITE_DECIMAL
;
;Процедура записывает 16-битовое число без знака в десятичной
;форме.
;   DXN:      16-битовое, беззнаковое число
;   Используется:  WRITE_HEX_DIGIT
;
WRITE_DECIMAL PROC    NEAR
    PUSH    AX          ; Сохранить используемые здесь регистры
    PUSH    CX
    PUSH    DX
    PUSH    SI
    MOV     AX,DX
    MOV     SI,10        ;Делить на 10,применяя SI
    XOR     CX,CX        ;Счет цифр, помещаемых в стек
NON_ZERO:
    XOR     DX,DX        ;Установить старшее слово N в 0
    DIV     SI            ;вычислить N/10 и (N mod 10)
    PUSH    DX            ;Поместить одну цифру в стек
    INC     CX
    OR      AX,AX        ;N = 0?
    JNE     NON_ZERO     ;Нет оператора, продолжить
WRITE_DIGIT_LOOP:
    POP     DX            ;Взять цифры в обратном порядке
    CALL    WRITE_HEX_DIGIT
    LOOP    WRITE_DIGIT_LOOP
END_DECIMAL:
    ( ? ) (может, удалить строку) [pause]
    POP     SI
    POP     DX
    POP     CX
    POP     AX
    RET
WRITE_DECIMAL ENDP

PUBLIC  WRITE_CHAR_N_TIMES
;
;Процедура записывает более одной копии символа
;   DL      Код символа
;   CX      Сколько раз записать символ
```

```
;      Используется:   WRITE_CHAR
;
WRITE_CHAR_N_TIMES      PROC    NEAR
    PUSH    CX
N_TIMES:
    CALL    WRITE_CHAR
    LOOP    N_TIMES
    POP     CX
    RET
WRITE_CHAR_N_TIMES      ENDP
PUBLIC      WRITE_PATTERN
;
; Процедура записывает строку на экран в соответствии с формой
;      DB {символ, сколько раз его записать}, 0
; Где {x} означает, что x может быть повторен любое количество
; раз;
;      DS:DX            Адрес данных
;
;Используется: WRITE_CHAR_N_TIMES
;
WRITE_PATTERN      PROC    NEAR
    PUSH    AX
    PUSH    CX
    PUSH    DX
    PUSH    SI
    PUSHF                                ;Записать флаг направления
    CLD                                  ;Установить на увеличение
    MOV     SI,DX                        ;Сдвинуть смещение в SI для LODSB
PATTERN_LOOP:
    LODSB                                ;Взять символ данных AL
    OR      AL,AL                        ;Конец данных (0h)?
    JZ      END_PATTERN                  ;Да, возврат
    MOV     DL,AL                        ;Нет, установить запись символов N раз
    LODSB                                ;Взять число повторений в AL
    MOV     CL,AL                        ;и поместить в CX для WRITE_CHAR_N_TIMES
    XOR     CH,CH                        ;Сбросить в 0 старший байт CX
    CALL    WRITE_CHAR_N_TIMES
    JMP     PATTERN_LOOP
END_PATTERN:
    POPF                                  ;Восстановить флаг направления
    POP     SI
    POP     DX
    POP     CX
    POP     AX
    RET
WRITE_PATTERN      ENDP
PUBLIC      WRITE_HEADER
```

```

DATA SEG SEGMENT PUBLIC
    EXTRN    HEADER_LINE_NO:BYTE
    EXTRN    HEADER_PART_1:BYTE
    EXTRN    HEADER_PART_2:BYTE
    EXTRN    DISK_DRIVE_NO:BYTE
    EXTRN    CURRENT_SECTOR_NO:WORD
DATA SEG    ENDS
    EXTRN        GOTO_XY:NEAR, CLEAR_TO_END_OF_LINE:NEAR
;
; Процедура строит заголовок с номерами дисковогода и сектора
;
;   Используются:    GOTO_XY, WRITE_STRING, WRITE_CHAR,
;                   WRITE_DECIMAL,    CLEAR_TO_END_OF_LINE
;   Считываются:    HEADER_LINE_NO, HEADER_PART_1,
;                   HEADER_PART_2, DISK_DRIVE_NO,
;                   CURRENT_SECTOR_NO
;
WRITE_HEADER    PROC    NEAR
    PUSH    DX
    XOR     DL,DL    ;Переместить курсор в строку заголовка
    MOV     DH,HEADER_LINE_NO
    CALL    GOTO_XY
    LEA     DX,HEADER_PART_1
    CALL    WRITE_STRING
    MOV     DL,DISK_DRIVE_NO
    ADD     DL,'A'    ;Печатать A,B...
    CALL    WRITE_CHAR
    LEA     DX,HEADER_PART_2
    CALL    WRITE_STRING
    MOV     DX,CURRENT_SECTOR_NO
    CALL    WRITE_DECIMAL
    CALL    CLEAR_TO_END_OF_LINE    ;Очистить остаток номера
;                                     сектора
    POP     DX
    RET
WRITE_HEADER    ENDP

PUBLIC    WRITE_STRING
;
; Процедура записывает строку символов на экран.
; Строка должна заканчиваться DB0
;   DS:DX    Адрес строки
;
;   Используется:    WRITE_CHAR
;
WRITE_STRING    PROC    NEAR
    PUSH    AX
    PUSH    DX
    PUSH    SI

```

```

    PUSHF                ;Сохранить флаг направления
    CLD                  ;Установить направление на увеличение
;                          (вперёд)
    MOV     SI,DX         ;Поместить адрес в SI для LODSB
STRING_LOOP:
    LODSB                ;Взять символ в AL
    OR      AL,AL         ;Найден 0?
    JZ      END_OF_STRING ;Да, строка закончена
    MOV     DL,AL         ;Нет, записать символ
    CALL    WRITE_CHAR
    JMP     STRING_LOOP
END_OF_STRING:
    POPF                ;Восстановить флаг направления
    POP     SI
    POP     DX
    POP     AX
    RET
WRITE_STRING            ENDP

PUBLIC      WRITE_PROMPT_LINE
    EXTRN   CLEAR_TO_END_OF_LINE:NEAR
    EXTRN   GOTO_XY:NEAR
DATA_SEG SEGMENT PUBLIC
    EXTRN   PROMPT_LINE_NO:BYTE
DATA_SEG ENDS
;
;Процедура выводит на экран строку приглашения и очищает
;конец строки
;    DS:DX        Адрес строки приглашения
;    Используются:  WRITE_STRING,CLEAR_TO_END_OF_LINE,
;                  GOTO_XY
;    Считывается:  PROMPT_LINE_NO
;
WRITE_PROMPT_LINE      PROC    NEAR
    PUSH    DX
    XOR     DL,DL        ;Записать строку приглашения
    MOV     DH,PROMPT_LINE_NO ; и поместить туда курсор
    CALL    GOTO_XY
    POP     DX
    CALL    WRITE_STRING
    CALL    CLEAR_TO_END_OF_LINE
    RET
WRITE_PROMPT_LINE      ENDP

    PUBLIC   WRITE_ATTRIBUTE_N_TIMES
    EXTRN    CURSOR_RIGHT:NEAR
;
;Процедура устанавливает атрибуты для N символов, начиная с
;текущего размещения курсора.

```

```
;
;   CX   Число символов, для которых устанавливаются атрибуты
;   DL   Устанавливаемый атрибут
;
;   Используется:   CURSOR_RIGHT
;
WRITE_ATTRIBUTE_N_TIMES      PROC    NEAR
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     BL,DL             ;Установить новый атрибут
    XOR     BH,BH             ;Страница 0
    MOV     DX,CX             ;CX используется подпрограммой BIOS
    MOV     CX, 1             ;Установить атрибут для одного символа
ATTR_LOOP:
    MOV     AH,8              ;Считать символ под курсором
    INT     10h
    CALL    CURSOR_RIGHT
    DEC     DX                 ;Установить атрибут для N символов?
    JNZ     ATTR_LOOP         ;Нет, продолжать
    POP     DX
    POP     CX
    POP     BX
    POP     AX
    RET
WRITE_ATTRIBUTE_N_TIMES     ENDP

CODE_SEG    ENDS

    END
```


Приложение С. Порядок загрузки сегментов

IBM Macro Assembler (версии 1.0 и 2.0) загружает сегменты в порядке, отличающемся от того, который используется во всех последних версиях Microsoft Macro Assembler. В этом приложении мы рассмотрим порядок загрузки сегментов и узнаем, как это может быть полезным, если EXE2BIN выдаёт сообщение об ошибке "File cannot be converted".

Порядок загрузки сегментов

В примерах программ, начиная с главы 13, используются два сегмента: CODE_SEG и DATA_SEG. Ассемблер фирмы IBM сообщает LINK о необходимости загрузки сегментов, в память в алфавитном порядке. Таким образом, если мы напишем:

```
CGROUP    CROUP    CODE_SEG    DATA_SEG
          ASSUME    CS:CGROUP, DS:CGROUP

DATA_SEG  SEGMENT PUBLIC
DATA_SEG  ENDS

CODE_SEG  SEGMENT PUBLIC
CODE_SEG  ENDS
END
```

то загрузка будет произведена в следующем порядке: сначала CODE_SEG, а затем DATA_SEG. Развернем этот фрагмент кода в настоящую программу так, чтобы мы смогли просмотреть карту её загрузки.

Программа примера позволяет понять, каким образом LINK загружает сегменты в память.

```
CGROUP  CROUP      CODE_SEG  DATA_SEG
        ASSUME      CS:CGROUP, DS:CGROUP

DATA_SEG  SEGMENT PUBLIC
        DB  0
DATA_SEG  ENDS

CODE_SEG  SEGMENT PUBLIC
        ORG  100h
MAIN: INT  20h
CODE_SEG  ENDS

ENDMAIN
```

Введите этот файл, назовите его SEGTEST.ASM, затем ассемблируйте и скомпонуйте его для того, чтобы создать карту загрузки:

```
A:\>LINK SEGTEST,SEGTEST, SEGTEST/MAP;
```

Если вы используете версию ассемблера фирмы IBM, то увидите следующую карту загрузки:

```
Warning: no stack segment

Start      Stop      Lenght      Name      Class
00000H     00101H     00102H      CODE_SEG
00110H     00110H     000001H     DATA_SEG

Origin      Group
0000:0      CGROUP

Address      Publics by Name

Address      Publics by Value

Program entry point at 0000:0100
```

LINK загрузил CODE_SEG в память перед DATA_SEG. Это именно тот порядок, который нам нужен. На самом деле CODE_SEG должен быть первым сегментом в памяти для того, чтобы программа начиналась со сдвигом 100h от начала группы сегментов.

С другой стороны, карта загрузки может сообщить, что эти два сегмента загружены в обратном порядке. Это означает, что вы используете версию ассемблера фирмы Microsoft. Если это так, то карта загрузки выглядит следующим образом:

```
Warning: no stack segment

      Start      Stop      Lenght      Name      Class
00000H    00000H    00001H    CODE_SEG
00010H    00111H    000102H   DATA_SEG

Origin      Group
0000:0      CGROUP

Address      Publics by Name
Address      Publics by Value

Program entry point at 0000:0110
```

Эта карта загрузки неправильна. DATA_SEG оказался загруженным в памяти перед CODE_SEG, а это означает, что выражение "ORG 100h" даст нам смещение от конца DATA_SEG, а не от начала группы сегментов.

Последняя строка в этой карте загрузки показывает, что начальный адрес нашей программы теперь 110h. Но для .COM-файла он должен быть равен 100h. Что же произойдет, если мы попытаемся создать из него .COM-файл?

Запустите EXE2BIN, и вы увидите следующее:

```
A:\>EXE2BIN SEGTEST SEGTEST.COM
File cannot be converted
A:\>
```

Это не очень полезное сообщение об ошибке - оно не разъясняет, почему EXE2BIN не может сделать из нашей программы .COM-файл. Но именно здесь удобно использовать старую карту загрузки. Взглянув на карту загрузки, мы можем увидеть, что LINK загрузил сегменты программы в память в неправильном порядке. Теперь нужно только узнать, как решить эту проблему.

Мы достаточно аккуратно создавали программы в этой книге , чтобы все они могли быть оттранслированы

и IBM, и Microsoft версиями ассемблера. Именно по этой причине мы помещали сегмент данных после сегмента кода во всех исходных файлах.

Если во время работы с ассемблерными программами вы создадите или столкнетесь с программами, в которых сегмент данных расположен в начале файла, используйте переключатель /A при трансляции этих программ трансляторами ассемблера фирмы Microsoft (MASM). Опция /A говорит MASM, что вы хотите, чтобы сегменты были загружены в алфавитном порядке. Чтобы опробовать этот переключатель, ассемблируйте SEGTEST.ASM ещё раз, задав следующую командную строку:

```
A:\>MASM SEGTEST/A;
```

Скомпонуйте этот файл и создайте новую карту загрузки. Теперь вы должны увидеть два сегмента в алфавитном порядке, с CODE_SEG, расположенным в файле первым.

Фазовые ошибки трансляции

Помещение сегмента данных в конец файлов не всегда удобно. В этом разделе мы рассмотрим ряд проблем, посвященных более оптимальным способам организации сегментов.

Рассмотрим этот конкретный пример:

```
CODE_SEG      SEGMENT PUBLIC
  ASSUME      CS:CODE_SEG,ES:CODE_SEG

BEGIN         PROC    NEAR      ;Взять номер сегмента
  MOV         AX,DATA_SEG      ;Остановить ES так, чтобы он указывал
;                                     на данные
  MOV         ES,AX            ; Считать "переменную" в AL
  MOV         AL,VARIABLE      ; Выход в DOS
  MOV         AH,4Ch
  INT         21h
BEGIN         ENDP

CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
VARIABLE      DB      0
```

```
DATA_SEG    ENS
            END    BEGIN
```

Сегмент данных помещён нами в конец программы для того, чтобы быть уверенными в том, что DATA_SEG будет загружен в память после CODE_SEG. Но если вы попытаетесь оттранслировать этот пример, то ассемблер выдаст сообщение о фазовой ошибке:

```
A:\>MASM TEST;

Microsoft (R) Macro Assembler Version 4.00

Copyright (C) Microsoft Corp 1981, 1983, 1084, 1985. All rights reserved.

TEST.ASM(IO) : error 6: Phase error between passes
51036 Bytes symbol space free

0 Warning Errors
1 Severe Errors
```

Что означает фазовая ошибка?

Оказывается, что при трансляции макроассемблер выполняет несколько проходов через программу. При первом проходе он собирает ту информацию, которая ему нужна, например тип переменных и в каком сегменте они расположены, но в интересах эффективности ассемблер также начинает ассемблировать программу и именно здесь находится корень проблем.

MASM ассемблирует инструкцию "MOV AL,VARIABLE" перед тем, как узнает о том, в каком сегменте содержится VARIABLE, поэтому он ассемблирует инструкцию MOV, как если бы мы не использовали переприсвоение сегментов (в данном случае ES:). Однако при следующем проходе MASM заметит, что ему нужно добавить переприсвоение сегментов, так как переменная VARIABLE находится в сегменте, на который указывает регистр ES. К сожалению, MASM не резервирует места для этой инструкции переприсвоения регистров во время первого прохода (или фазы), поэтому он генерирует сообщение о фазовой ошибке.

Нам надо объявлять все переменные перед тем, как мы используем их в файле. Если мы сделаем это и используем Microsoft Macro Assembler, то сегмент данных будет первым в памяти, что не представляет

проблем для .EXE файлов, где мы можем использовать несколько сегментов.

Если же вы хотите, чтобы сегмент кода был загружен в память первым, то имеется простое решение: поместить заглушку сегмента перед сегментом данных. Детали можно увидеть в следующем примере:

```
CODE_SEG      SEGMENT PUBLIC      ;Загрузить CODE_SEG первым
CODE_SEG      ENDS

DATA_SEG      SEGMENT PUBLIC
VARIABLE      DB      0
DATA_SEG      ENDS

CODE_SEG      SEGMENT PUBLIC
ASSUME        CS:CODE_SEG, ES:DATA_SEG

BEGIN         PROC NEAR      ;Взять номер сегмента
MOV           AX,DATA_SEG ;Установить ES так, чтобы он указывал
;                на данные
MOV           ES,AX        ;Считать "переменную" в AL
MOV           AL,VARIABLE  ; Выйти в DOS
MOV           AH,4Ch
INT           21h

BEGIN         ENDP

CODE_SEG      ENDS

END           BEGIN
```

Сообщение EXE2BIN "File Cannot be Converted"

Если при использовании EXE2BIN вы сталкиваетесь с какими-либо проблемами, прежде всего проверьте карту загрузки и убедитесь в том, что CODE_SEG является первым сегментом в файле. Проверьте также количество и порядок размещения объявленных сегментов, потому что возможно наличие нескольких разных версий одного и того же сегмента. Например:

00000H	00103H	00104H	CODE_SEG
00110H	00105H	00006H	DATA_SEG
00120H	00101H	00102H	CODE_SEG

В этом случае CODE_SEG фрагментирован. Если в карте загрузки имеется более одной части каждого сегмента, то это означает, что вы столкнетесь с проблемами, и они могут исходить из следующих различных источников:

- может отсутствовать псевдооператор PUBLIC после всех определений сегментов;
- возможно отличие определений SEGMENT в исходных файлах, проверьте все исходные файлы и убедитесь в том, что все определения SEGMENT идентичны;
- в одном из исходных файлов может отсутствовать или быть неправильным оператор GROUP, аккуратно проверьте все выражения группировки, чтобы убедиться в том, что они одинаковы;
- если операторы GROUP в порядке, проверьте операторы ASSUME, они должны выглядеть так:

```
ASSUME CS:CGROUP, DS:CGROUP
```

- вы определили сегмент стека, но в .COM программах сегмент стека не нужен, и вам не надо его определять.
- точка входа не 100h. Это может быть из-за того, что вы не поместили имя начальной процедуры после псевдооператора END в основном исходном файле, или из-за того, что скомпоновали файлы в неправильном порядке. Главная процедура должна быть в первом файле, указанном в списке LINK.

Вы можете найти более подробную информацию о сообщениях об ошибках и о том, что они означают, в Приложении D.

Приложение D. Наиболее часто встречающиеся сообщения об ошибках

Это приложение содержит большую часть основных сообщений об ошибках, с которыми можно столкнуться при работе с MASM, LINK и EXE2BIN. Если вы не найдёте в этом списке какого-либо сообщения об ошибке, то ищите его в руководстве по макроассемблеру или по DOS.

Сообщения об ошибках делятся на три группы: для MASM, для LINK, и для EXE2BIN. Внутри каждого раздела сообщения об ошибках даны в алфавитном порядке.

MASM

Block nesting error

Вы можете увидеть это сообщение вместе с сообщениями Open procedures или Open segments. Смотрите описание этих сообщений об ошибках далее.

End of file, no END directive

Либо у вас отсутствует оператор END в конце файла, либо необходимо добавить пустую строку после существующего оператора END. Версия ассемблера фирмы Microsoft ожидает пустую строку в самом конце файла. Если у вас нет по крайней мере одной пустой строки после END, MASM не прочтает оператора END.

Must be declared in pass 1

Это сообщение обычно появляется в связи с оператором GROUP. Например, если в программе есть строка XGROUP GROUP CODE_SEG, DATA_SEG", но не определен сегмент, называющийся DATA_SEG, то вы можете увидеть это сообщение. Убедитесь, что определены все сегменты, указанные в операторе GROUP.

No or unreachable CS

MASM должен видеть оператор ASSUME для того, чтобы знать, как ассемблировать некоторые особые инструкции, например такие, как инструкции ветвления и обращения к другим процедурам. Это сообщение об ошибке означает, что MASM не может найти оператор ASSUME или найденный ASSUME содержит ошибку. Проверьте исходный файл, чтобы убедиться в том, что в нем присутствует оператор ASSUME, и что он написан правильно.

Open procedures

Это означает, что отсутствует оператор PROC или ENDP, или что имена процедуры в паре операторов PROC/ENDP разные. Убедитесь, что каждый PROC имеет соответствующий ему оператор ENDP, и проверьте, что имена процедуры в операторах PROC и ENDP одинаковы.

Open segments

Отсутствует оператор SEGMENT или ENDS, или отличаются имена в паре операторов SEGMENT/ENDS. Убедитесь в том, что каждый SEGMENT имеет соответствующий ему оператор ENDS, и проверьте имя процедуры в обоих операторах SEGMENT и ENDS чтобы убедиться в том, что они одинаковы.

Symbol not defined

Существует три вещи, которые необходимо проверить при появлении этого сообщения об ошибке:

- допущена орфографическая ошибка в имени;
проверьте строку, на которую указывает сообщение об ошибке, чтобы убедиться в том, что вы правильно напечатали это имя;
- Вы могли ошибиться при написании имени при первом определении процедуры или переменной; проверьте, правильно ли написаны имена переменных в строчке, вызвавшей ошибку;
- возможно, вы пропустили или объявление EXTRN, или имя в EXTRN было написано с ошибкой.

LINK

Fixup offset exceeds field width

(Смещение ? адресной привязки превышает ширину поля)

Довольно сложная, а иногда наиболее сложная ошибка для отладки. Сообщение обычно означает, что вы объявили некоторые процедуры как процедуры типа FAR, но позже в объявлении EXTRN определили эти же процедуры как NEAR.

Это может также означать, что группа сегментов превысила 64-Килобайтный предел. Такие ошибки можно легко найти, взглянув на размер поля в карте загрузки. Сообщение может также появиться, если сегмент становится фрагментированным. В таких случаях два фрагмента могут превышать 64Кбайт, а это означает, что для нормальной работы обращения к процедурам должны быть типа FAR. Более детальную информацию о фрагментированных сегментах вы можете найти в Приложении С.

Если же все перечисленное не относится к вашей программе, то необходимо искать глубже. Внимательно прочитайте Приложение С, затем создайте карту загрузки программы. В ней можно найти подсказку к

решению проблемы. Например, проверьте порядок загрузки сегментов. Он может оказаться неверным.

Symbol defined more than once

Это означает, что вы определили одну и ту же процедуру или переменную в двух исходных файлах. Убедитесь, что вы определили каждое имя только в одном исходном файле, а затем применяйте EXTRN в тех местах, где надо использовать эту процедуру или переменную.

Unresolved externals

Когда вы видите это сообщение, то либо в файле, в котором определена процедура или переменная, отсутствует PUBLIC, либо вы допустили ошибку в имени в объявлении EXTRN в другом исходном файле.

Warning: no stack segment

На самом деле это не сообщение об ошибке, это просто предупреждение. Вы увидите это предупреждение при компоновке примеров из этой книги, так мы создаем .COM-файлы, а .COM-файлы не используют отдельный сегмент для стека. Смотрите главу 28, в которой приведён пример программы, компоновка которой не вызовет этого предупреждения.

EXE2BIN

File cannot be converted

Это, пожалуй, единственное сообщение об ошибке, которое вы можете получить при использовании EXE2BIN, и оно не очень информативно. Наиболее часто оно возникает в одной из приведенных ниже ситуаций:

1. Сегменты расположены в неправильном порядке, поэтому один из сегментов загружен в памяти перед CODE_SEG. Проверьте карту загрузки, чтобы увидеть, действительно ли в этом суть проблемы. Более подробная информация по этому вопросу содержится в Приложении С.
2. Основная программа не является первой в списке файлов для LINK. Это вполне возможно, поэтому произведите перекомпоновку, чтобы убедиться, что проблема заключается не в этом. Подсказку о том, что проблема заключается именно в этом вы можете найти в карте загрузки.
3. Основная программа не содержит "ORG 100h" в качестве первого оператора после объявления "CODE_SEG SEGMENT PUBLIC". Кроме того, убедитесь, что оператор END в основном исходном файле включает метку инструкции, с которой вы хотите начать - например, "END DSKPATCH".

Если эти рекомендации не помогают, смотрите Приложение С для более детальной информации.

СОДЕРЖАНИЕ

Почему именно Ассемблер?.....	3
Dskpatch _ _ _ _ _	5
Требования к конфигурации компьютера.....	5
Организация этой книги _ _ _ _ _	5

Часть I. Язык машины

Глава 1. DEBUG и арифметика	
Шестнадцатеричные числа.....	7
Debug _ _ _ _ _	8
Шестнадцатеричная арифметика.....	9
Перевод шестнадцатеричных чисел в десятичную форму _ _ _ _ _	11
Пятизначные шестнадцатеричные числа.....	15
Перевод десятичных чисел в шестнадцатеричную форму _ _ _ _ _	15
Отрицательные числа.....	17
Биты, байты, слова и двоичная система счисления _ _ _ _ _	19
Дополнительный код - особый тип отрицательного числа	21
Итог _ _ _ _ _	22
Глава 2. Арифметика микропроцессора 8088	
Регистры как переменные.....	24
Память и микропроцессор 8088 _ _ _ _ _	26
Сложение, метод микропроцессора 8088.....	28
Вычитание, метод микропроцессора 8088 _ _ _ _ _	31
Отрицательные числа в микропроцессоре 8088.....	31
Байты в микропроцессоре 8088 _ _ _ _ _	32
Умножение и деление, метод микропроцессора 8088.....	33
Итог _ _ _ _ _	35
Глава 3. Вывод символов на экран	
INT - мощное прерывание.....	36
Грациозный выход - INT 20h _ _ _ _ _	39
Программа из двух строк - соединение частей вместе.....	40
Ввод программ _ _ _ _ _	41
Использование команды MOV для пересылки данных между регистрами	42
Вывод на экран строки символов _ _ _ _ _	45
Итог.....	47

Глава 4. Вывод на экран двоичных чисел	
Циклический сдвиг и флаг переноса.....	49
Сложение с использованием флага переноса _ _ _ _ _	51
Организация циклов.....	51
Вывод на экран двоичного числа _ _ _ _ _	54
Команда обхода.....	56
Итог _ _ _ _ _	57
Глава 5. Вывод на экран чисел в шестнадцатеричной форме	
Операция сравнения и биты состояния.....	59
Вывод на экран одной шестнадцатеричной цифры _ _ _ _ _	61
Ещё одна инструкция сдвига.....	64
Логика и AND _ _ _ _ _	66
Сборка всех частей программы вместе.....	68
Итог _ _ _ _ _	69
Глава 6. Ввод символов с клавиатуры	
Ввод одного символа.....	69
Считывание шестнадцатеричного числа, состоящего из одной цифры _ _ _ _ _	71
Считывание двузначного шестнадцатеричного числа.....	72
Итог _ _ _ _ _	73
Глава 7. Процедуры - двоюродные сёстры подпрограмм	
Процедуры.....	73
Стек и адрес возврата _ _ _ _ _	75
Использование инструкций PUSH и POP.....	77
Более простой способ считывания шестнадцатеричных чисел _ _ _ _ _	79
Итог.....	81

ЧАСТЬ II . Язык ассемблера

Глава 8. Добро пожаловать в ассемблер!	
Создание программы без использования Debug.....	83
Создание исходных файлов _ _ _ _ _	86
Компоновка.....	87
Обратно в Debug _ _ _ _ _	89
Комментарии.....	89
Метки _ _ _ _ _	90
Итог	92
Глава 9. Ассемблер и процедуры	
Процедуры ассемблера.....	93

Процедуры, выводящие на экран шестнадцатеричные числа.....	97
Начала модульного конструирования программ _ _ _ _ _	100
Структура программы.....	101
Итог _ _ _ _ _	102
Глава 10. Вывод на экран десятичных чисел	
Вспоминаем перевод чисел.....	103
Некоторые трюки _ _ _ _ _	106
Внутренняя работа.....	107
Итог _ _ _ _ _	109
Глава 11. Сегменты	
Секционирование памяти микропроцессором 8088.....	110
Псевдооператоры сегментов _ _ _ _ _	117
Псевдооператор ASSUME.....	118
"Ближние" и "дальные" вызовы процедур _ _ _ _ _	119
Подробнее об инструкции INT.....	122
Вектора прерываний _ _ _ _ _	124
Итог	125
Глава 12. Коррекция курса	
Дискеты, сектора и Dskpatch	126
План игры _ _ _ _ _	128
Итог	130
Глава 13. Модульное конструирование	
Раздельное ассемблирование.....	131
Три Закона Модульного Конструирования _ _ _ _ _	134
Итог	139
Глава 14. Дампирование памяти	
Режимы адресации.....	140
Добавление в дамп символов _ _ _ _ _	146
Дампирование 256 байт памяти.....	148
Итог _ _ _ _ _	153
Глава 15. Дампирование сектора диска	
Облегчение жизни.....	154
Формат файла для Make _ _ _ _ _	155
"Починка" Disp_sec	156
Считывание сектора _ _ _ _ _	158
Итог	162
Глава 16. Улучшение изображения сектора	
Добавление графических символов.....	163
Добавление к изображению адресов _ _ _ _ _	165
Добавление горизонтальных линий	168
Добавление к изображению чисел _ _ _ _ _	174
Итог	176

Часть III. ROM BIOS IBM PC

Глава 17. Подпрограммы ROM BIOS	
Video_io, подпрограммы ROM BIOS	177
Движение курсора _ _ _ _ _	182
Применение изменяемых переменных	184
Построение заголовка _ _ _ _ _	188
Итог	191
Глава 18. Окончательный вариант WRITE_CHAR	
Новая процедура WRITE_CHAR	193
Стирание до конца строки _ _ _ _ _	196
Итог	198
Глава 19. Диспетчер	
Dispatcher	199
Считывание других секторов _ _ _ _ _	206
Философия следующих глав	209
Глава 20. Вызов программистам	
Псевдокурсор.....	210
Простейшее редактирование _ _ _ _ _	212
Другие изменения и дополнения к Dskpatch	212
Глава 21. Псевдокурсоры	
Изменение атрибута символа	220
Итог _ _ _ _ _	221
Глава 22. Простейшее редактирование	
Перемещение псевдокурсоров	222
Простейшее редактирование _ _ _ _ _	226
Итог	230
Глава 23. Шестнадцатеричный и десятичный ввод	
Шестнадцатеричный ввод	231
Десятичный ввод _ _ _ _ _	240
Итог	243
Глава 24. Улучшенный ввод с клавиатуры	
Новая процедура READ_STRING	244
Глава 25. В поисках ошибок	
Исправление DISPATCHER	252
Итог _ _ _ _ _	254
Глава 26. Запись модифицированных секторов на диск	
Запись на диск	255
Некоторые приёмы отладки _ _ _ _ _	257
Построение карты загрузки	258
Трассировка ошибок _ _ _ _ _	261

Symdeb	263
Символьная отладка _ _ _ _ _	263
Экранный свопинг	264
Итог _ _ _ _ _	266
Глава 27. Другая половина сектора	
Скроллинг половины сектора	267
Итог _ _ _ _ _	270

ЧАСТЬ IV. Дополнение к сказанному

Глава 28. Перемещение	
Программы, состоящие из нескольких сегментов	271
Перемещение _ _ _ _ _	276
.COM-программы в сравнении с	
.EXE-программами	280
Глава 29. Подробнее о сегментах и ASSUME	
Переписывание сегментов	283
Ещё один взгляд на ASSUME _ _ _ _ _	285
Фазовые ошибки	286
Послесловие _ _ _ _ _	287

Приложение А . Руководство по диску

Примеры к главам	289
Улучшенная версия Dskpatch _ _ _ _ _	290

Приложение В . Листинг DSKPATCH

Описание процедур	295
CURSOR.ASM _ _ _ _ _	295
DISK_IO.ASM	295
DISPATCH.ASM _ _ _ _ _	296
DISP_SEC.ASM	296
DSKPATCH.ASM _ _ _ _ _	296
EDITOR.ASM	296
KBD_IO.ASM _ _ _ _ _	297
PHANTOM.ASM	297
VIDEO_IO.ASM _ _ _ _ _	299
Листинг процедур Dskpatch	300
DSKPATCH Make File _ _ _ _ _	300
CURSOR.ASM	300
DISK_IO.ASM _ _ _ _ _	303
DISPATCH.ASM	306
DISP_SEC.ASM _ _ _ _ _	308
DSKPATCH.ASM	312
EDITOR.ASM _ _ _ _ _	314
KBD_IO.ASM	316

PHANTOM.ASM	323
VIDEO_IO.ASM _ _ _ _ _	329

Приложение С

Порядок загрузки сегментов	336
Фазовые ошибки трансляции _ _ _ _ _	339
Сообщение "File Cannot be Converted"	341

Приложение D . Сообщения об ошибках

MASM	343
LINK _ _ _ _ _	344
EXE2BIN	345

Н83 Нортон П., Соухэ Д.
Язык ассемблера для IBM PC: Пер. с англ. - М.: Из-
дательство "Компьютер", 1992.-352с: ил.

ISBN 5-88201-008-X.

Книга предназначена для читателей-программистов, желающих изучить язык ассемблер и возможности микропроцессоров 8088 с целью написания более мощных, быстрых и коротких программ. "Гроссмейстер" программирования Питер Нортон делится своим богатым опытом с читателями. Книга существенно расширяет кругозор пользователей IBM PC.

2404090000-009

Н ----- КБ27-016-91

М00 (03) -92

ББК 32.973-01

Питер Нортон
Джон Соухэ

Язык ассемблера для IBM PC

Ответственный за выпуск Р.А. Колоев
Зав. редакцией И.Г. Дмитриева
Редакторы Л.Д. Григорьева, Т. А. Петрова
Худож. редактор Ю.И. Артюхов
Техн. редактор Г. А. Полякова
Корректор Т.М. Васильева
Переплет художника Ф.Г. Миллера

ИБ № 0009

Подписано о печать 25.08.92.

Формат 84x108 1/32. Бумага офсетная.

Гарнитура "Универс". Печать офсетная.

Усл. печ. л. 18,48. Усл. кр.-отт. 18,64. Уч.-изд. л. 19,52.

Тираж 20 000 экз. Заказ 6588. "С"009

Издательство "Компьютер"
109072, Москва, ул. Серафимовича, 2

Издательство "Финансы и статистика"
101000, Москва, ул. Чернышевского, 7.

Оригинал-макет изготовлен в ТОО "НИМА-ОКИ"

Ордена Октябрьской Революции и ордена Трудового Красного Знамени
МПО "Первая Образцовая типография" Министерства печати и
информации РФ.
113054, Москва, Валовая, 28.