Зубков С. В.

Assembler

для DOS, Windows и UNIX

БЕСТСЕЛЛЕР

ПИТЕР®

ДМК ПРЕСС

Для программистов

УДК 004.438Assembler
ББК 32.973.26-018.1
3-91

Зубков С. В.
3-91 Assembler для DOS, Windows и UNIX / Зубков Сергей Владимирович. - 3-е изд., стер. - М. : ДМК Пресс ; СПб. : Питер, 2004. - 608 с. : ил. - (Серия «Для программистов»).

ISBN 5-94074-259-9

В книге освещаются все аспекты современного программирования на ассемблере для DOS, Windows 95/NT и UNIX (Solaris, Linux и FreeBSD), включая создание резидентных программ и драйверов, прямое программирование периферийных устройств, управление защищенным режимом и многое другое. Детально рассматривается архитектура процессоров Intel вплоть до Pentium III. Все главы иллюстрируются подробными примерами работоспособных программ.

Издание ориентировано как на профессионалов, так и на начинающих без опыта программирования.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность наличия технических и просто человеческих ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

© Зубков С. В., 2004
© ДМК Пресс, 2004

ISBN 5-94074-259-9

Зубков Сергей Владимирович

Assembler для DOS, Windows и UNIX

Главный редактор *Захаров И. М.*
editor-in-chief@dmkpress.ru
Научный редактор *Шалаев Н. В.*
Литературный редактор *Космачева Н. А.*
Верстка *Али-Заде В. Х.*
Дизайн обложки *Антонов А. И.*

Подписано в печать 12.11.2003. Формат $70\times100^{1}/_{16}$.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 49,4. Тираж 3000 экз. Зак. № **3501**

Издательство «ДМК Пресс», 105023, Москва, пл. Журавлева, 2/8
Web-сайт издательства: www.dmk.ru
Internet-магазин: www.abook.ru, www.dmk.ru

Ордена Трудового Красного Знамени
ГУП Чеховский полиграфический комбинат
Министерства Российской Федерации по делам печати,
телерадиовещания и средств массовых коммуникаций
142300, г. Чехов Московской области
Тел. (272) 71-336, факс (272) 62-536

Содержание

Введение

Первый вопрос, который задает себе человек, впервые услышавший об ассемблере, - а зачем он, собственно, нужен? Особенно теперь, когда все пишут на C/C++, Delphi или других языках высокого уровня? Действительно очень многое можно создать на C, но ни один язык, даже такой популярный, не может претендовать на то, чтобы на нем можно было написать абсолютно *все*.

Итак, на ассемблере пишут:

- *все, что требует максимальной скорости выполнения:* основные компоненты компьютерных игр, ядра операционных систем реального времени и просто критические участки программ;
- *все, что взаимодействует с внешними устройствами:* драйверы, программы, работающие напрямую с портами, звуковыми и видеоплатами;
- *все, что использует полностью возможности процессора: ядра* многозадачных операционных систем, DPMI-серверы и вообще любые программы, переводящие процессор в защищенный режим;
- *все, что полностью использует возможности операционной системы:* вирусы и антивирусы, защиты от несанкционированного доступа, программы, обходящие эти защиты, и программы, защищающиеся от данных программ;
- *и многое другое.* Стоит познакомиться с ассемблером поближе, как оказывается, что большую часть из того, что обычно пишут на языках высокого уровня, лучше, проще и быстрее написать на ассемблере.

«Как же так? - спросите вы, прочитав последний пункт. - Ведь всем известно, что ассемблер - неудобный язык, и писать на нем долго и сложно!» Попробуем перечислить мотивы, которые обычно выдвигаются в доказательство того, что ассемблер не нужен.

Говорят, что ассемблер трудно выучить. Любой язык программирования трудно выучить. Легко выучить C или Delphi после Pascal, потому что они похожи. А попробуйте освоить Lisp, Forth или Prolog, и окажется, что ассемблер в действительности даже проще, чем любой абсолютно незнакомый язык программирования.

Говорят, что программы на ассемблере трудно понять. Разумеется, на ассемблере легко написать неудобочитаемую программу... точно так же, как и на любом другом языке! Если вы знаете язык и если автор программы не старался ее запутать, то понять программу будет не сложнее, чем если бы она была написана на Basic.

Говорят, что программы на ассемблере трудно отлаживать. Программы на ассемблере легко отлаживать - опять же при условии, что вы знаете язык. Более того, знание ассемблера часто помогает отлаживать программы на других языках, потому что оно дает представление о том, как на самом деле функционирует компьютер и что происходит при выполнении команд языка высокого уровня.

Говорят, что современные компьютеры такие быстрые, что ассемблер больше не нужен. Каким бы быстрым ни был компьютер, пользователю всегда хочется большей скорости, иначе не наблюдалось бы постоянного спроса на еще более мощные компьютеры. И самой быстрой программой на данном оборудовании всегда будет программа, написанная на ассемблере.

Говорят, что писать на ассемблере сложно. В этом есть доля правды. Очень часто авторы программ на ассемблере «изобретают велосипеды», программируя заново элементарные процедуры типа форматированного вывода на экран или генератора случайных чисел, в то время как программисты на С просто вызывают стандартные функции. Библиотеки таких функций существуют и для ассемблера, но они не стандартизированы и не распространяются вместе с компиляторами.

Говорят, что программы на ассемблере не переносятся. Действительно, в этом заключается самая сильная и самая слабая сторона ассемблера. Во-первых, благодаря этой особенности программы на ассемблере используют возможности компьютера с наибольшей полнотой; во-вторых, эти же программы не будут работать на другом компьютере. Стоит заметить, что и другие языки часто не гарантируют переносимости - та же программа на С, написанная, например, под Windows 95, не скомпилируется ни на Macintosh, ни на SGI.

Далеко не всё, что говорят об ассемблере, является правдой, и далеко не все, кто говорят об ассемблере, на самом деле знают его. Но даже ярые противники согласятся с тем, что программы на ассемблере — самые быстрые, самые маленькие и могут то, что не под силу программам, созданным на любом другом языке программирования.

Эта книга рассчитана на читателей с разным уровнем подготовки - как на начинающих, которые хотят познакомиться с ассемблером серьезно или желают лишь написать пару программ, выполняющих необычные трюки с компьютером, так и на профессиональных программистов, которые тоже найдут здесь интересные разделы.

Почти все, что надо знать об ассемблере, где-нибудь да объяснено, а также объяснено многое из того, что не заботит большинство программистов. С одной стороны, чтобы написать простую программу, не нужно знать язык и устройство процессора в совершенстве, но, с другой стороны, по-настоящему серьезная работа потребует и основательной подготовки. Уровень сложности в этой книге возрастает от начала к концу, но в первой ее половине отдельные абзацы помечены специальной пиктограммой  , которая означает, что данный абзац лучше пропустить при чтении, если вы знакомитесь с ассемблером впервые. Впрочем, если у вас есть время и желание выучить ассемблер с нуля, - читайте все по порядку. Если же вам хочется немедленно приступить к написанию программ, начните сразу с главы 4, но будьте готовы к тому, что иногда придется возвращаться к предыдущим главам за более подробным описанием тех или иных команд. И наконец, если вам уже доводилось программировать на ассемблере, - выбирайте то, что интересно.

Глава 1. Предварительные сведения

1.1. Что нужно для работы с ассемблером

Прежде всего вам потребуется *ассемблер*. Здесь самое время сказать, что язык программирования, которым мы собираемся заниматься, называется «язык ассемблера» (assembly language). Ассемблер - это программа, которая переводит текст с языка, понятного человеку, в язык, понятный процессору, то есть говорят, что она переводит язык ассемблера в машинный код. Однако сначала в повседневной речи, а затем и в литературе слово «ассемблер» стало также и названием самого языка программирования. Понятно, что, когда говорят «программа на ассемблере», имеют в виду язык, а когда говорят «макроассемблер версии 6.13», имеют в виду программу. Вместе с ассемблером обязательно должна быть еще одна программа - *компоновщик* (linker), которая и создает исполнимые файлы из одного или нескольких объектных модулей, полученных после запуска ассемблера. Помимо этого для разных целей могут потребоваться дополнительные вспомогательные программы - компиляторы ресурсов, расширители DOS и тому подобное (см. табл. 1).

Трудно говорить о том, продукция какой из трех компаний (Borland, Microsoft или Watcom) однозначно лучше. С точки зрения удобства компиляции TASM лучше подходит для создания 16-битных программ для DOS, WASM - для 32-битных программ для DOS, MASM - для Windows. С точки зрения удобства программирования развитость языковых средств растет в ряду WASM - MASM - TASM. Все примеры программ в этой книге построены так, что можно использовать любой из этих компиляторов.

Таблица 1. Ассемблеры и сопутствующие программы

	Microsoft	Borland	Watcom
DOS, 16 бит	masm или ml, link (16 бит)	tasm tlink	wasm wlink
DOS, 32 бита	masm или ml, link (32 бита) и dosx link (16 бит) и dos32	tasm tlink wdosx или dos32	wasm wlink dos4gw, pmodew, zrdx или wdosx
Windows EXE	masm386 или ml link (32 бита) rc	tasm tlink32 brcc32	wasm wlink wrc
Windows DLL	masm386 или ml link (32 бита)	tasm tlink32 implib	wasm wlink wlib

Разумеется, существуют и другие компиляторы, например бесплатно распространяемый в сети Internet NASM или условно бесплатный A86, но пользоваться ими проще, если вы уже знаете турбо- или макроассемблер. Бесплатно распространяемый GNU ассемблер, gas, вообще использует совершенно непохожий синтаксис, который будет рассмотрен в главе 11, рассказывающей о программировании для UNIX.

Во всех программах встречаются ошибки. Если вы собираетесь не только упражняться на примерах из книги, но и написать что-то свое, то вам рано или поздно обязательно потребуется отладчик. Кроме поиска ошибок отладчики иногда применяют и для того, чтобы исследовать работу существующих программ. Безусловно, самый мощный отладчик на сегодняшний день - SoftICE от NuMega Software. Это фактически единственный отладчик для Windows 95/NT, позволяющий исследовать все - от ядра Windows до программ на C++, поддерживающий одновременно 16- и 32-битный код и т. п. Другие популярные отладчики, распространяемые вместе с соответствующими ассемблерами, - Codeview (MS), Turbo Debugger (Borland) и Watcom Debugger (Watcom).

Еще одна особенность ассемблера, отличающая его от всех остальных языков программирования, - возможность дизассемблирования. То есть, имея исполняемый файл, с помощью специальной программы *(дизассемблера)* почти всегда можно получить исходный текст на ассемблере. Например, можно дизассемблировать BIOS вашего компьютера и узнать, как выполняется переключение видеорежимов, или драйвер для DOS, чтобы написать такой же для Windows. Дизассемблер не является необходимой программой, но иногда очень удобно иметь его под рукой. Лучшие дизассемблеры на сегодняшний день - Sourcer от V Communications и IDA.

И наконец, последняя необязательная, но весьма полезная утилита - шестнадцатеричный редактор. Многие подобные редакторы (hiew, proview, lview, hexit) имеют встроенный дизассемблер, так что можно, например, открыв в таком редакторе свою программу, посмотреть, как скомпилировался тот или иной участок программы, поправить какую-нибудь команду ассемблера или изменить значения констант и тут же, без перекомпиляции, запустить программу, чтобы посмотреть на результат изменений.

1.2. Представление данных в компьютерах

Для того чтобы освоить программирование на ассемблере, следует познакомиться с двоичными и шестнадцатеричными числами. Иногда в тексте программы можно обойтись и обычными десятичными числами, но без понимания того, как на самом деле хранятся данные в памяти компьютера, очень трудно использовать логические и битовые операции, упакованные форматы данных и многое другое.

1.2.1. Двоичная система счисления

Практически все существующие сейчас компьютерные системы, включая Intel, используют для вычислений двоичную систему счисления. В их электрических

цепях напряжение может принимать два значения, и эти значения назвали нулем и единицей. Двоичная система счисления как раз и использует только эти две цифры, а вместо степеней десяти, как в обычной десятичной системе, здесь применяют степени двойки. Чтобы перевести двоичное число в десятичное, надо сложить двойки в степенях, соответствующих позициям, где в двоичном стоят единицы. Например:

$$10010111b=1\times2^7+0\times2^6+0\times2^5+1\times2^4+0\times2^3+1\times2^2+1\times2^1+1\times2^0=128+16+4+2+1=151$$

Для перевода десятичного числа в двоичное можно, например, разделить его на 2, записывая остаток справа налево (см. табл. 2).

Таблица 2.Перевод числа из десятичной системы в двоичную

	Остаток
151/2 = 75	1
75/2 = 37	1
37/2 = 18	1
18/2 = 9	0
9/2 = 4	1
4/2 = 2	0
2/2 = 1	0
1/2 = 0	1
Результат:	10010111b

Чтобы отличать двоичные числа от десятичных, в ассемблерных программах в конце каждого двоичного числа ставят букву b.

7.2.2. *Биты, байты и слова*

Минимальная единица информации называется *битом.* Бит принимает только два значения - обычно О и 1. На самом деле они совершенно необязательны - один бит может принимать значения «да» и «нет», показывать присутствие и отсутствие жесткого диска, а также является ли персонаж игры магом или воином - важно лишь то, что бит имеет только два значения. Но многие величины принимают большее число значений, следовательно, для их описания нельзя обойтись одним битом.

Единица информации размером 8 бит называется *байтом.* Байт - это минимальный объем данных, который реально может использовать компьютерная программа. Даже для изменения значения одного бита в памяти надо сначала считать байт, содержащий его. Биты в байте нумеруют справа налево, от нуля до семи, нулевой бит часто называют младшим битом, а седьмой - старшим (см.рис.1).

Так как всего в байте восемь бит, он может принимать до $2^8 = 256$ разных значений. Байт используют для представления целых чисел от 0 до 255 (тип unsigned char в С), целых чисел со знаком от -128 до +127 (тип signed char в С), набора символов ASCII (тип char в С) или переменных, принимающих менее 256 значений, например для представления десятичных чисел от 0 до 99.

Следующий по размеру базовый тип данных - *слово.* Размер одного слова в процессорах Intel - два байта (см. рис. 2). Биты с 0 по 7 составляют младший байт слова, а биты с 8 по 15 - старший. В слове содержится 16 бит, а значит, оно может

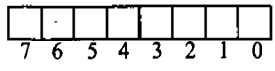

РИС. 1.Байт

принимать до 2^{16} = 65 536 разных значений. Слова используют для представления целых чисел без знака со значениями 0-65 535 (тип unsigned short в C), целых чисел со знаком от -32 768 до +32 767 (тип short int в C), адресов сегментов и смещений при 16-битной адресации. Два слова подряд образуют *двойное слово,* состоящее из 32 бит, а два двойных слова - одно *учетверенное слово* (64 бита). Байты, слова и двойные слова - основные типы данных, с которыми мы будем работать.

Еще одно важное замечание: в компьютерах с процессорами Intel все данные хранятся так, что младший байт находится по младшему адресу, поэтому слова записываются задом наперед, то есть сначала (по младшему адресу) - последний (младший) байт, а потом (по старшему адресу) - первый (старший) байт. Если из программы всегда обращаться к слову как к слову, а к двойному слову как к двойному слову, это не оказывает никакого влияния. Но если вы хотите прочитать первый (старший) байт из слова в памяти, то придется увеличить адрес на 1. Двойные и учетверенные слова записываются так же - от младшего байта к старшему.

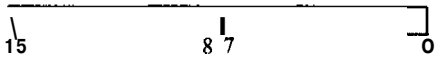

Рис. 2. Слово

1.2.3. Шестнадцатеричная система счисления

Главное неудобство двоичной системы счисления - это размеры чисел, с которыми приходится обращаться. На практике с двоичными числами работают, только если необходимо следить за значениями отдельных битов, а когда размеры переменных превышают хотя бы четыре бита, используется шестнадцатеричная система. Она хороша тем, что компактнее десятичной, и тем, что перевод в двоичную систему и обратно происходит очень легко. В шестнадцатеричной системе используется 16 «цифр» (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F), и номер позиции цифры в числе соответствует степени, в которую надо возвести число 16, следовательно:

```
96h = 9 X 16 + 6 = 150
```

Перевод в двоичную систему и обратно осуществляется крайне просто - вместо каждой шестнадцатеричной цифры подставляют соответствующее четырехзначное двоичное число:

```
9h = 1001b, 6h = 0110b, 96h = 10010110b
```

В ассемблерных программах при записи чисел, начинающихся с A, B, C, D, E, F, в начале приписывается цифра 0, чтобы не перепутать такое число с названием переменной или другим идентификатором. После шестнадцатеричных чисел ставится буква h (см. табл. 3).

Таблица 3. Двоичные и шестнадцатеричные числа

Десятичное	Двоичное	Шестнадцатеричное
0	0000b	00h
1	0001b	01h
2	0010b	02h
3	0011b	03h
4	0100b	04h
5	0101b	05h
6	0110b	08h
7	0111b	07h
8	1000b	08h
9	1001b	09h
10	1010b	OAh
11	1011b	0Bh
12	1100b	OCh
13	1101b	ODh
14	1110b	OEh
15	1111b	OFh
16	10000b	10h

7.2.4. *Числа со знаком*

Легко использовать байты, слова или двойные слова для представления целых положительных чисел - от 0 до 255, 65 535 или 4 294 967 295 соответственно. Чтобы применять те же самые байты или слова для представления отрицательных чисел, существует специальная операция, известная как дополнение до двух. Для изменения знака числа выполняют инверсию, то есть заменяют в двоичном представлении числа все единицы нулями и нули единицами, а затем прибавляют 1.

Например, пусть используются переменные типа слова:

```
150 = 0096h = 0000 0000 1001 0110b
инверсия дает: 1111 1111 0110 1001b
+1 = 1111 1111 0110 1010b = OFF6Ah
```

Проверим, что число на самом деле -150: сумма с + 150 должна быть равна нулю:

```
+150 + (-150) = 0096h + OFF6Ah = 10000h
```

Единица в 16-м разряде не помещается в слово, следовательно, мы действительно получили 0. В данном формате старший (7-й, 15-й, 31-й для байта, слова, двойного слова) бит всегда соответствует знаку числа: 0 - для положительных и 1 - для отрицательных. Таким образом, схема с использованием дополнения до двух выделяет для положительных и отрицательных чисел равные диапазоны:

-128...+127 - для байта, -32 768...+32 767 - для слов, -2 147 483 648...+2 147 483 647 - для двойных слов.

7.2.5. Логические операции

Самые распространенные варианты значений, которые может принимать один бит, - это значения «правда» и «ложь», используемые в логике, откуда происходят так называемые «логические операции» над битами. Так, если объединить «правду» и «правду» — получится «правда», а если объединить «правду» и «ложь» - «правды» не получится. В ассемблере нам встретятся четыре основные операции - И (AND), ИЛИ (OR), «исключающее ИЛИ» (XOR) и отрицание (NOT), действие которых приводится в табл. 4.

Таблица 4. Логические операции

И	ИЛИ	Исключающее ИЛИ	Отрицание
0 AND 0 = 0	0 OR 0 = 0	0 XOR 0 = 0	NOT 0 = 1
0 AND 1 = 0	0 OR 1 = 1	0 XOR 1 = 1	NOT 1 = 0
1 AND 0 = 0	1 OR 0 = 1	1 XOR 0 = 1	
1 AND 1 = 1	1 OR 1 = 1	1 XOR 1 = 0	

Все перечисленные операции являются побитовыми, поэтому для выполнения логического действия над числом надо перевести его в двоичный формат и произвести операцию над каждым битом, например:

```
96h AND 0Fh = 10010110b AND 00001111b = 00000110b = 06h
```

1.2.6. Коды символов

Для представления всех букв, цифр и знаков, появляющихся на экране компьютера, обычно используется всего один байт. Символы, соответствующие значениям от 0 до 127, то есть первой половине всех возможных значений байта, были стандартизированы и названы символами ASCII (хотя часто кодами ASCII именуют всю таблицу из 256 символов). Сюда входят некоторые управляющие коды (символ с кодом 0Dh - конец строки), знаки препинания, цифры (символы с кодами 30h - 39h), большие (41h - 5Ah) и маленькие (61h - 7Ah) латинские буквы. Вторая половина символьных кодов используется для алфавитов других языков и псевдографики, набор и порядок символов в ней отличаются в разных странах и даже в пределах одной страны. Например, для букв одного только русского языка существует пять вариантов размещения во второй половине таблицы символов ASCII (см. приложение 1). Существует также стандарт, использующий слова для хранения кодов символов, известный как UNICODE или UCS-2, и даже двойные слова (UCS-4), но мы пока не будем на нем останавливаться.

12.7. *Организация памяти*

Память *с* точки зрения процессора представляет собой последовательность байтов, каждому из которых присвоен уникальный адрес со значениями от 0 до $2^{32}-1$ (4 Гб). Программы же могут работать с памятью как с одним непрерывным массивом (модель памяти flat) или как с несколькими массивами (сегментированные *модели* памяти). Во втором случае для задания адреса любого байта требуется два числа - адрес начала массива и адрес искомого байта внутри массива. Помимо основной памяти программы могут использовать *регистры* - специальные ячейки памяти, расположенные физически внутри процессора, доступ к которым осуществляется не по адресам, а по именам. Но здесь мы вплотную подходим к рассмотрению собственно работы процессора, о чем подробно рассказано в следующей главе.

Глава 2. Процессоры Intel в реальном режиме

Процессор Intel x86 после включения питания оказывается в так называемом режиме реальной адресации памяти, или просто реальном режиме. Большинство операционных систем сразу же переводит его в защищенный режим, позволяющий им обеспечивать многозадачность, распределение памяти и другие функции. Пользовательские программы в таких операционных системах часто работают еще и в режиме V86, из которого им доступно все то же, что и из реального, кроме команд, относящихся к управлению защищенным режимом. Следовательно, данная глава описывает реальный режим и V86, то есть все, что доступно программисту в подавляющем большинстве случаев, если он не проектирует операционную систему или DPMI-сервер.

2.1. Регистры процессора

Начиная с 80386 процессоры Intel предоставляют 16 основных регистров для пользовательских программ плюс еще 11 регистров для работы с мультимедийными приложениями (MMX) и числами с плавающей запятой (FPU/NPX). Все команды так или иначе изменяют значения регистров, и всегда быстрее и удобнее обращаться к регистру, чем к памяти.

Из реального (но не из виртуального) режима помимо основных регистров доступны также регистры управления памятью (GDTR, IDTR, TR, LDTR), регистры управления (CRO, CR1 - CR4), отладочные регистры (DRO - DR7) и машинно-специфичные регистры, но они не применяются для решения повседневных задач и рассматриваются далее в соответствующих разделах.

2.1.1. Регистры общего назначения

32-битные регистры EAX (аккумулятор), EBX (база), ECX (счетчик), EDX (регистр данных) могут использоваться без ограничений для любых целей - временного хранения данных, аргументов или результатов различных операций. Названия регистров происходят от того, что некоторые команды применяют их специальным образом: так, аккумулятор часто необходим для хранения результата действий, выполняемых над двумя операндами, регистр данных в этих случаях получает старшую часть результата, если он не умещается в аккумулятор, регистр-счетчик работает как счетчик в циклах и строковых операциях, а регистр-база - при так называемой адресации по базе. Младшие 16 бит каждого из этих регистров применяются как самостоятельные регистры с именами AX, BX, CX, DX. На

самом деле в процессорах 8086-80286 все регистры были 16-битными и назывались именно так, а 32-битные EAX - EDX появились с введением 32-битной архитектуры в 80386. Кроме этого, отдельные байты в 16-битных регистрах AX - DX тоже могут использоваться как 8-битные регистры и иметь свои имена. Старшие байты этих регистров называются AH, BH, CH, DH, а младшие - AL, BL, CL, DL (см. рис. 3).

Остальные четыре регистра - ESI (индекс источника), EDI (индекс приемника), EBP (указатель базы), ESP (указатель стека) - имеют более конкретное назначение и применяются для хранения всевозможных временных переменных. Регистры ESI tfEDI необходимы в строковых операциях, EBP и ESP - при работе со стеком (см. раздел 2.1.3). Так же как и в случае с регистрами EAX - EDX, младшие половины этих четырех регистров называются SI, DI, BP и SP соответственно, и в процессорах до 80386 только они и присутствовали.

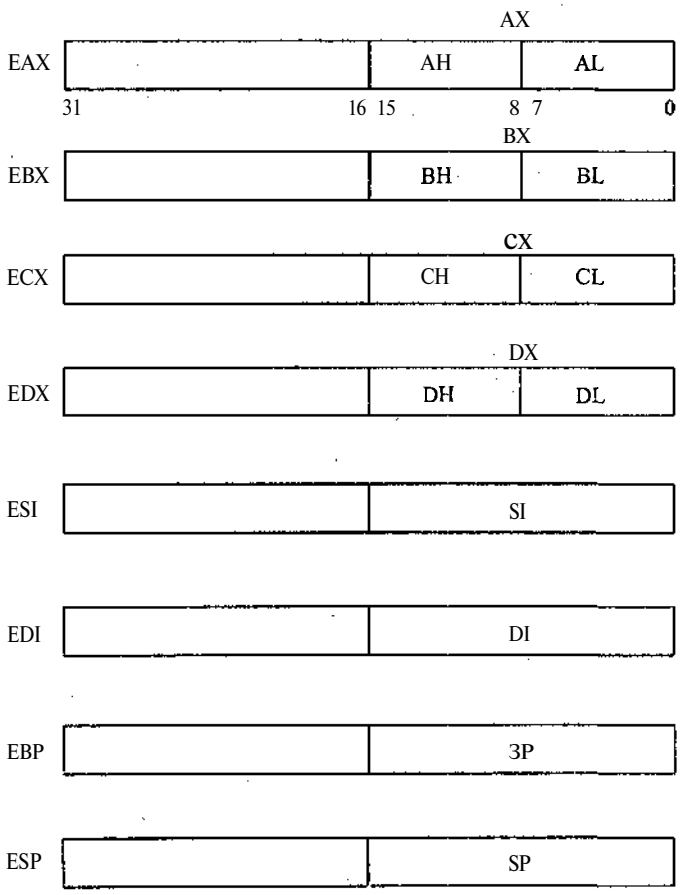

Рис. 3. Регистры общего назначения

2.1.2. Сегментные регистры

При использовании сегментированных моделей памяти для формирования любого адреса нужны два числа - адрес начала сегмента и смещение искомого байта относительно этого начала (в бессегментной модели памяти flat адреса начал всех сегментов равны). Операционные системы (кроме DOS) могут размещать сегменты, с которыми работает программа пользователя, в разных местах памяти и даже временно записывать их на диск, если памяти не хватает. Так как сегменты способны оказаться где угодно, программа обращается к ним, применяя вместо настоящего адреса начала сегмента 16-битное число, называемое селектором. В процессорах Intel предусмотрено шесть 16-битных регистров - CS, DS, ES, FS, GS, SS, где хранятся селекторы. (Регистры FS и GS отсутствовали в 8086, но появились уже в 80286.) Это означает, что в любой момент можно изменить параметры, записанные в этих регистрах.

В реальном режиме селектор каждого сегмента равен адресу его начала, деленному на 16. Чтобы получить адрес в памяти, 16-битное смещение складывают с этим селектором, предварительно сдвинутым влево на 4. Таким образом, оказывается, что максимальный доступный адрес в реальном режиме $2^{20} - 1 = 1\,048\,575$. Для сравнения: в защищенном режиме адрес начала для каждого сегмента хранится отдельно, так что возможно 2^{46} (64 Тб) различных логических адреса в формате сегментхмещение (программа определяет до 16 383 сегментов, каждый из которых до 4 Тб), хотя реально процессор адресуется только к 4 или 64 (для Pentium Pro) Тб памяти.

В отличие от DS, ES, GS, FS, которые называются регистрами сегментов данных, CS и SS отвечают за сегменты двух особенных типов - сегмент кода и сегмент стека. Первый содержит программу, исполняющуюся в данный момент, следовательно, запись нового селектора в этот регистр приводит к тому, что далее будет исполнена не следующая по тексту программы команда, а команда из кода, находящегося в другом сегменте, с тем же смещением. Смещение очередной выполняемой команды всегда хранится в специальном регистре EIP (указатель инструкции, 16-битная форма IP), запись в который также приведет к тому, что далее будет исполнена какая-нибудь другая команда. На самом деле все команды передачи управления - перехода, условного перехода, цикла, вызова подпрограммы и т. п. - и осуществляют эту самую запись в CS и EIP.

2.1.3. Стек

Стек - организованный специальным образом участок памяти, который используется для временного хранения переменных, передачи параметров вызываемым подпрограммам и сохранения адреса возврата при вызове процедур и прерываний. Легче всего представить стек в виде стопки листов бумаги (это одно из значений слова «stack» в английском языке) — вы можете класть и забирать листы только с вершины стопки. Поэтому, если записать в стек числа 1, 2, 3, то при чтении они окажутся в обратном порядке - 3, 2, 1. Стек располагается в сегменте памяти, описываемом регистром SS, и текущее смещение вершины стека отражено

в регистре ESP, причем во время записи значение этого смещения *уменьшается,* то есть он «растет вниз» от максимально возможного адреса (см. рис. 4). Такое расположение стека «вверх ногами» может быть необходимо, к примеру, в бессегментной модели памяти, когда все сегменты, включая сегменты стека и кода, занимают одну и ту же область - память целиком. Тогда программа исполняется в нижней области памяти, в области малых адресов, и EIP растет, а стек располагается в верхней области памяти, и ESP уменьшается.

При вызове подпрограммы параметры в большинстве случаев помещают в стек, а в EBP записывают текущее значение ESP. Если подпрограмма использует стек для хранения локальных переменных, ESP изменится, но EBP можно будет использовать для того, чтобы считывать Значения параметров напрямую из стека (их смещения запишутся как EBP + номер параметра). Более подробно вызовы подпрограмм и все возможные способы передачи параметров рассмотрены в разделе 5.2.1

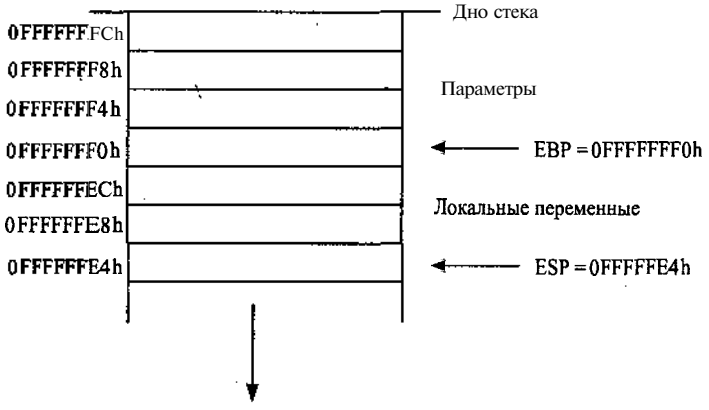

Рис. 4. Стек

2.1.4. Регистр флагов

Еще один важный регистр, использующийся при выполнении большинства команд, - регистр флагов. Как и раньше, его младшие 16 бит, представлявшие собой весь этот регистр до процессора 80386, называются FLAGS. В EFLAGS каждый бит является *флагом,* то есть устанавливается в 1 при определенных условиях или установка его в 1 изменяет поведение процессора. Все флаги, расположенные в старшем слове регистра, имеют отношение к управлению защищенным режимом, поэтому здесь рассмотрен только регистр FLAGS (см. рис. 5):

- CF - флаг переноса. Устанавливается в 1, если результат предыдущей операции не уместился в приемнике и произошел перенос из старшего бита или

Рис. 5. Регистр флагов FLAGS

если требуется заем (при вычитании), в противном случае - в 0. Например, после сложения слова OFFFFh и 1, если регистр, в который надо поместить результат, - слово, в него будет записано 0000h и флаг CF = 1.

а PF - флаг четности. Устанавливается в 1, если младший байт результата предыдущей команды содержит четное число битов, равных 1, и в 0, если нечетное. Это не то же самое, что делимость на два. Число делится на два без остатка, если его самый младший бит равен нулю, и не делится, когда он равен 1.

- AF - флаг полупереноса или вспомогательного переноса. Устанавливается в 1, если в результате предыдущей операции произошел перенос (или заем) из третьего бита в четвертый. Этот флаг используется автоматически командами двоично-десятичной коррекции.

Q ZF - флаг нуля. Устанавливается в 1, если результат предыдущей команды - ноль.

- SF - флаг знака. Он всегда равен старшему биту результата.

Q TF - флаг ловушки. Он был предусмотрен для работы отладчиков, не использующих защищенный режим. Установка его в 1 приводит к тому, что после выполнения каждой программной команды управление временно передается отладчику (вызывается прерывание 1 - см. описание команды INT).

- IF - флаг прерываний. Сброс этого флага в 0 приводит к тому, что процессор перестает обрабатывать прерывания от внешних устройств (см. описание команды INT). Обычно его сбрасывают на короткое время для выполнения критических участков кода.
- DF — флаг направления. Он контролирует поведение команд обработки строк: когда он установлен в 1, строки обрабатываются в сторону уменьшения адресов, когда DF = 0 - наоборот.
- OF - флаг переполнения. Он устанавливается в 1, если результат предыдущей арифметической операции над числами со знаком выходит за допустимые для них пределы. Например, если при сложении двух положительных чисел получается число со старшим битом, равным единице, то есть отрицательное, и наоборот.

Флаги IOPL (уровень привилегий ввода-вывода) и NT (вложенная задача) применяются в защищенном режиме.

2.2. Способы адресации

Большинство команд процессора вызываются с аргументами, которые в ассемблере принято называть *операндами.* Например: команда сложения содержимого регистра с числом требует задания двух операндов — содержимого регистра и числа. Далее рассмотрены все существующие способы задания адреса хранения операндов - способы адресации.

2.2.7. *Регистровая адресация*

Операнды могут располагаться в любых регистрах общего назначения и сегментных регистрах. Для этого в тексте программы указывается название соответствующего

регистра, например: команда, копирующая в регистр AX содержимое регистра BX, записывается как

```
mov     ax,bx
```

2.2.2. *Непосредственная адресация*

Некоторые команды (все арифметические, кроме деления) позволяют указывать один из операндов непосредственно в тексте программы. Например: команда

```
mov     ax,2
```

помещает в регистр AX число 2.

2.2.3. *Прямая адресация*

Если у операнда, располагающегося в памяти, известен адрес, то его можно использовать. Если операнд - слово, находящееся в сегменте, на который указывает ES, со смещением от начала сегмента 0001, то команда

```
mov     ax,es:0001
```

поместит это слово в регистр AX. В реальных программах для задания статических переменных обычно используют директивы определения данных (раздел 3.3), которые позволяют ссылаться на статические переменные не по адресу, а по имени. Тогда, если в сегменте, указанном в ES, была описана переменная `word_var` размером в слово, можно записать ту же команду как

```
mov     ax,es:word_var
```

В таком случае ассемблер сам заменит слово `word_var` на соответствующий адрес. Если селектор сегмента данных находится в DS, то имя сегментного регистра при прямой адресации можно не указывать, DS используется по умолчанию. Прямая адресация иногда называется адресацией по смещению.

Адресация отличается для реального и защищенного режимов. В реальном (так же как и в режиме V86) смещение всегда 16-битное. Это значит, что ни непосредственно указанное смещение, ни результат сложения содержимого разных регистров в более сложных методах адресации не могут превышать границ слова. При работе в Windows, DOS4G, PMODE и в других ситуациях, когда программа будет запускаться в защищенном режиме, смещение не должно превышать границ двойного слова.

2.2.4. *Косвенная адресация*

По аналогии с регистровыми и непосредственными операндами адрес операнда в памяти также можно не указывать, а хранить в любом регистре. До процессора 80386 для этого можно было использовать только BX, SI, DI и BP, но потом ограничения были сняты и адрес операнда разрешили считывать также из EAX, EBX, ECX, EDX, ESI, EDI, EBP и ESP (но не из AX, CX, DX или SP напрямую - надо использовать EAX, ECX, EDX, ESP соответственно или предварительно скопировать смещение в BX, SI, DI или BP). Например, следующая команда помещает

в регистр AX слово из ячейки памяти, селектор сегмента которой находится в DS, а смещение - в BX:

```
mov     ax,[bx]
```

Как и в случае с прямой адресацией, DS используется по умолчанию, но не всегда: если смещение берут из регистров ESP, EBP или BP, то в качестве сегментного регистра применяется SS. В реальном режиме можно свободно работать со всеми 32-битными регистрами, надо только следить, чтобы их содержимое не превышало границ 16-битного слова.

2.2.5. *Адресация по базе со сдвигом*

Теперь скомбинируем два предыдущих метода адресации. Следующая команда

```
mov     ax,[bx+2]
```

помещает в регистр AX слово, которое есть в сегменте, указанном в DS, со смещением на два больше, чем число из BX. Так как слово занимает ровно 2 байта, эта команда поместила в AX слово, непосредственно следующее за тем, которое было в предыдущем примере. Такая форма адресации используется в тех случаях, когда в регистре находится адрес начала структуры данных, а доступ надо осуществить к какому-нибудь ее элементу. Еще один вариант применения адресации по базе со сдвигом — доступ из подпрограммы к параметрам, переданным в стеке, используя регистр BP (EBP) в качестве базы и номер параметра в качестве смещения, что детально рассмотрено в разделе 4.3.2. Другие допустимые формы записи этого способа адресации:

```
mov     ax,[bp]+2
mov     ax,2[bp]
```

До процессора 80386 в качестве базового регистра разрешалось использовать только BX, BP, SI или DI и сдвиг мог быть только байтом или словом (со знаком). Начиная с 80386 и старше, процессоры Intel позволяют дополнительно использовать EAX, EBX, ECX, EDX, EBP, ESP, ESI и EDI, так же как и для обычной косвенной адресации. С помощью этого метода разрешается организовывать доступ к одномерным массивам байтов: смещение соответствует адресу начала массива, а число в регистре - индексу элемента массива, который надо считать. Очевидно, что, если массив состоит не из байтов, а из слов, придется умножать базовый регистр на два, а если из двойных слов - на четыре. Для этого предусмотрен специальный метод - косвенная адресация.

2.2.6. *Косвенная адресация с масштабированием*

Этот метод адресации полностью идентичен предыдущему, однако с его помощью можно прочитать элемент массива слов, двойных слов или учетверенных слов, просто поместив номер элемента в регистр:

```
mov     ax, [esi*2]+2
```

Множитель, который равен 1, 2, 4 или 8, соответствует размеру элемента массива — байту, слову, двойному или учетверенному слову. Из регистров в этом

варианте адресации можно использовать только EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, но не SI, DI, BP или SP.

2.2.7. *Адресация по базе с индексированием*

В этом методе адресации смещение операнда в памяти вычисляется как сумма чисел, содержащихся в двух регистрах, и смещения, если оно указано. Все перечисленные ниже команды представляют собой разные формы записи одного и того же действия:

```
mov     ax,[bx+si+2]
mov     ax,[bx][si]+2
mov     ax,[bx+2][si]
mov     ax,[bx][si+2]
mov     ax,2[bx][si]
```

В регистр AX помещается слово из ячейки памяти со смещением, равным сумме чисел, содержащихся в BX, SI, и числа 2. Из 16-битных регистров так можно складывать только BX + SI, BX + DI, BP + SI и BP + DI, а из 32-битных — все восемь регистров общего назначения. Как и для прямой адресации, вместо непосредственного указания числа разрешено использовать имя переменной, заданной одной из директив определения данных. Таким образом можно считать, например, число из двумерного массива: если задана таблица 10×10 байт, 2 - смещение ее начала от начала сегмента данных (на практике будет использоваться имя этой таблицы), BX = 20, а SI = 7, приведенные команды прочитают слово, состоящее из седьмого и восьмого байтов третьей строки. Если таблица состоит не из одиночных байтов, а из слов или двойных слов, удобнее использовать наиболее полную форму - адресацию по базе с индексированием и масштабированием.

2.2.8. *Адресация по базе с индексированием и масштабированием*

Это самая полная схема адресации, в которую входят все случаи, рассмотренные ранее как частные. Полный адрес операнда можно записать как выражение, представленное на рис. 6.

Смещение может быть байтом или двойным словом. Если ESP или EBP используются в роли базового регистра, Селектор сегмента операнда берется по умолчанию из регистра SS, во всех остальных случаях - из DS.

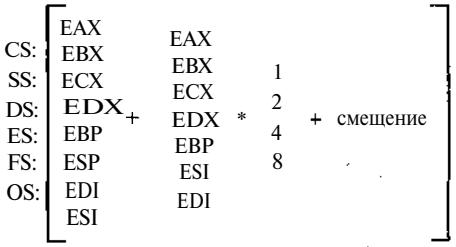

Рис. 6. Полная форма адресации

2.3. Основные непривилегированные команды

В этом разделе описаны все непривилегированные команды процессоров Intel серии x86, включая команды расширений IA NPX (чаще называемое FPU - расширение для работы с числами с плавающей запятой) и IA MMX (мультимедийное расширение). Для каждой команды указана форма записи, название и модель процессоров Intel, начиная с которой она поддерживается: 8086, 80186, 80286, 80386, 80486, P5 (Pentium), MMX, P6 (Pentium Pro и Pentium II).

2.3.1. Пересылка данных

Команда	Назначение	Процессор
MOV приемник, источник	Пересылка данных	8086

Базовая команда пересылки данных. Копирует содержимое источника в приемник, источник не изменяется. Команда MOV действует аналогично операторам присваивания из языков высокого уровня, то есть команда

```
mov     ax,bx
```

эквивалентна выражению

```
ax:=bx;
```

языка Pascal или

```
ax=bx;
```

языка C, за исключением того, что команда ассемблера позволяет работать не только с переменными в памяти, но и со всеми регистрами процессора.

В качестве источника для MOV могут использоваться: число (непосредственный операнд), регистр общего назначения, сегментный регистр или переменная (то есть операнд, находящийся в памяти); в качестве приемника: регистр общего назначения, сегментный регистр (кроме CS) или переменная. Оба операнда должны быть одного и того же размера - байт, слово или двойное слово.

Нельзя выполнять пересылку данных с помощью MOV из одной переменной в другую, из одного сегментного регистра в другой и нельзя помещать в сегментный регистр непосредственный операнд - эти операции выполняют двумя командами MOV (из сегментного регистра в обычный и уже из него в другой сегментный) или парой команд PUSH/POP.

Загрузка регистра SS командой MOV автоматически запрещает прерывания до окончания следующей за ней команды MOV, поэтому можно не опасаться, что в этот момент произойдет прерывание, обработчик которого получит неправильный стек. В любом случае для загрузки значения в регистр SS предпочтительнее команда LSS.

Команда	Назначение	Процессор
CMOVcc приемник,источник	Условная пересылка данных	P6

Это набор команд, которые копируют содержимое источника в приемник, если удовлетворяется то или иное условие (см. табл. 5). Источником может быть регистр общего назначения или переменная, а приемником - только регистр. Требование, которое должно выполниться, - просто равенство нулю или единице тех или иных флагов из регистра FLAGS, но, если использовать команды CMOVcc сразу после CMP (сравнение) с теми же операндами, условия приобретают особый смысл, например:

```
cmp     ax,bx           ; Сравнить ax и bx.
cmovl   ax,bx           ; Если ax < bx, скопировать bx в ax.
```

Слова «выше» и «ниже» в табл. 5 относятся к сравнению чисел без знака, слова «больше» и «меньше» учитывают знак.

Таблица 5. Разновидности команды CMOVcc

Код команды	Реальное условие	Условие для CMP
CMOVA CMOVNBE	CF = 0 и ZF = 0	Если выше Если не ниже и не равно
CMOVAE CMOVNB CMOVNC	CF = 0	Если выше или равно Если не ниже Если нет переноса
CMOVB CMOVNAE CMOVC	CF = 1	Если ниже Если не выше и не равно Если перенос
CMOVBE CMOVNA	CF = 1 или ZF = 1	Если ниже или равно Если не выше
CMOVE CMOVZ	ZF = 1	Если равно Если ноль
CMOVG CMOVNLE	ZF = 0 и SF = OF	Если больше Если не меньше и не равно
CMOVGE CMOVNL	SF = OF	Если больше или равно Если не меньше
CMOVL CMOVNGE	SF <>OF	Если меньше Если не больше и не равно
CMOVLE CMOVNG	ZF = 1 или SF 0 OF	Если меньше или равно Если не больше
CMOVNE CMOVNZ	ZF = 0	Если не равно Если не ноль
CMOVNO	OF = 0	Если нет переполнения
CMOVO	OF = 1	Если есть переполнение
CMOVNP CMOVPO	PF = 0	Если нет четности Если нечетное
CMOVP CMOVPE	PF = 1	Если есть четность Если четное
CMOVNS	SF = 0	Если нет знака
CMOVS	SF = 1	Если есть знак

Команда	Назначение	Процессор
XCHG операнд1,операнд2	Обмен операндов между собой	8086

Содержимое операнда 2 копируется в операнд 1, а старое содержимое операнда 1 - в операнд 2. XCHG можно выполнять над двумя регистрами или над регистром и переменной.

```
xchg    eax,ebx ; То же, что три команды на языке C:
                ; temp = eax;eax = ebx;ebx = temp.
xchg    al,al   ; А эта команда ничего не делает.
```

Команда	Назначение	Процессор
BSWAP регистр32	Обмен байтов внутри регистра	80486

Обращает порядок байтов в 32-битном регистре. Биты 0-7 (младший байт младшего слова) меняются местами с битами 24-31 (старший байт старшего слова), а биты 8-15 (старший байт младшего слова) - с битами 16-23 (младший байт старшего слова).

```
mov     eax,12345678h
bswap   eax     ; Теперь в eax находится 78563412h.
```

Чтобы обратить порядок байтов в 16-битном регистре, следует использовать команду XCHG:

```
xchg    al,ah   ; Обратить порядок байтов в AX.
```

В процессорах Intel команду BSWAP можно использовать и для обращения порядка байтов в 16-битных регистрах, но в некоторых совместимых процессорах других фирм этот вариант не реализован.

Команда	Назначение	Процессор
PUSH источник	Поместить данные в стек	8086

Помещает содержимое источника в стек. Источником может быть регистр, сегментный регистр, непосредственный операнд или переменная. Фактически эта команда уменьшает ESP на размер источника в байтах (2 или 4) и копирует содержимое источника в память по адресу SS:[ESP]. Команда PUSH почти всегда используется в паре с POP (считать данные из стека). Поэтому, чтобы скопировать содержимое одного сегментного регистра в другой (что нельзя выполнить одной командой MOV), можно использовать такую последовательность команд:

```
push    cs
pop     ds      ; Теперь DS указывает на тот же сегмент, что и CS.
```

Другой вариант применения команд PUSH/POP - временное хранение переменных, например:

```
push    eax     ; Сохраняет текущее значение EAX.
...             ; Здесь располагаются какие-нибудь команды,
                ; которые используют EAX, например CMPXCHG.
pop     eax     ; Восстанавливает старое значение EAX.
```

Начиная с процессора 80286 команда PUSH ESP (или SP) помещает в стек значение ESP до того, как она же его уменьшит, а на 8086 регистр SP располагался в стеке уже уменьшенным на два.

Команда	Назначение	Процессор
POP приемник	Считать данные из стека	8086

- Помещает в приемник слово или двойное слово, находящееся в вершине стека, увеличивая ESP на 2 или 4 соответственно. POP выполняет действие, полностью обратное PUSH. Приемником может быть регистр общего назначения, сегментный регистр, кроме CS (чтобы загрузить CS из стека, надо воспользоваться командой RET), или переменная. Если в роли приемника выступает операнд, использующий ESP для косвенной адресации, команда POP вычисляет адрес операнда уже после того, как она увеличивает ESP.

Команда	Назначение	Процессор
PUSHA	Поместить в стек	80186
PUSHAD	все регистры общего назначения	80386

PUSHA располагает в стеке регистры в следующем порядке: AX, CX, DX, BX, SP, BP, SI и DI. PUSHAD помещает в стек EAX, ECX, EDX, EBX, ESP, EBP, ESI и EDI. (В случае с SP и ESP используется значение, которое находилось в регистре до начала работы команды.) В паре с командами POPA/POPAD, считывающими эти же регистры из стека в обратном порядке, это позволяет писать подпрограммы (обычно обработчики прерываний), которые не должны изменять значения регистров по окончании своей работы. В начале такой подпрограммы вызывают команду PUSHA, а в конце - POPA.

На самом деле PUSHA и PUSHAD - одна и та же команда с кодом 60h. Ее поведение определяется тем, выполняется ли она в 16- или в 32-битном режиме. Если программист использует команду PUSHAD в 16-битном сегменте или PUSHA в 32-битном, ассемблер просто записывает перед ней префикс изменения размерности операнда (66h). Это же будет распространяться на некоторые другие пары команд: POPA/POPAD, POPF/POPFD, PUSHF/PUSHFD, JCXZ/JECXZ, CMPSW/CMPSD, INSW/INSD, LODSW/LODSD, MOVSW/MOVSD, OUTSW/OUTSD, SCASW/SCASD и STOSW/STOSD.

Команда	Назначение	Процессор
POPA	Загрузить из стека	80186
POPAD	все регистры общего назначения	80386

Команды выполняют действия, полностью обратные действиям PUSHA и PUSHAD, но помещенное в стек значение SP или ESP игнорируется. POPA загружает из стека DI, SI, BP, увеличивает SP на два, загружает BX, DX, CX, AX, а POPAD загружает EDI, ESI, EBP, увеличивает ESP на 4 и загружает EBX, EDX, ECX, EAX.

Команда	Назначение	Процессор
IN приемник,источник	Считать данные из порта	8086

Копирует число из порта ввода-вывода, номер которого указан в источнике, в приемник. Приемником может быть только AL, AX или EAX. Источник - или непосредственный операнд, или DX, причем во время использования непосредственного операнда можно указывать лишь номера портов не больше 255.

Команда	Назначение	Процессор
OUT приемник,источник	Записать данные в порт	8086

Копирует число из источника (AL, AX или EAX) в порт ввода-вывода, номер которого указан в приемнике. Приемник может быть либо непосредственным номером порта (не больше 255), либо регистром DX. На командах IN и OUT строится все общение процессора с устройствами ввода-вывода - клавиатурой, жесткими дисками, различными контроллерами, и используются они, в первую очередь, в драйверах устройств. Например, чтобы включить динамик PC, достаточно выполнить команды:

```
in      al,61h
or      al,3
out     61h,al
```

Программирование портов ввода-вывода рассмотрено подробно в разделе 5.10.

Команда	Назначение	Процессор
CWD	Конвертирование слова в двойное слово	3086
CDQ	Конвертирование двойного слова в учетверенное	80386

Команда CWD превращает слово в AX в двойное слово, младшая половина которого (биты 0–15) остается в AX, а старшая (биты 16–31) располагается в DX. Команда CDQ выполняет аналогичное действие по отношению к двойному слову в EAX, расширяя его до учетверенного слова в EDX:EAX.

Эти команды лишь устанавливают все биты регистра DX или EDX в значение, равное величине старшего бита регистра AX или EAX, сохраняя таким образом его знак.

Команда	Назначение	Процессор
CBW	Конвертирование байта в слово	8086
CWDE	Конвертирование слова в двойное слово	80386

CBW расширяет байт, находящийся в регистре AL, до слова в AX; CWDE расширяет слово в AX до двойного слова в EAX. Команды CWDE и CWD отличаются тем, что CWDE размещает свой результат в EAX, в то время как CWD, выполняющая точно такое же действие, располагает результат в паре регистров DX:AX. Так же как и в командах CWD/CDQ, расширение выполняется путем установки

каждого бита старшей половины результата равным старшему биту исходного байта или слова, то есть:

```
mov     al,0F5h          ; AL = OF5h = 245 = -11.
Cbw                      ; Теперь AX = 0FFF5h = 65 525 = -11.
```

Как и в случае с командами PUSHA/PUSHAD, пара CWD/CDQ- это одна команда с кодом 99h, и пара CBW/CWDE - одна команда с кодом 98h. Интерпретация этих команд зависит от того, в каком (16-битном или в 32-битном) сегменте они исполняются. Если указать CDQ или CWDE в 16-битном сегменте, ассемблер поставит префикс изменения разрядности операнда.

Команда	Назначение	Процессор
MOVSX приемник,источник	Пересылка с расширением знака	80386

Копирует содержимое источника (регистр или переменная размером в байт или слово) в приемник (16- или 32-битный регистр) и расширяет знак аналогично командам CBW/CWDE.

Команда	Назначение	Процессор
MOVZX приемник,источник	Пересылка с расширением нулями	80386

Копирует содержимое источника (регистр или переменная размером в байт или слово) в приемник (16- или 32-битный регистр) и расширяет нулями, то есть команда

```
movzx   ax, bl
```

эквивалентна паре команд

```
mov     al,bl
mov     ah,0
```

Команда	Назначение	Процессор
XLAT адрес	Трансляция в соответствии с таблицей	8086
XLATB		

Помещает в AL байт из таблицы в памяти по адресу ES:BX (или ES:EBX) со смещением относительно начала таблицы равным AL. В качестве аргумента для XLAT в ассемблере можно указать имя таблицы, но эта информация никак не используется процессором и служит только в качестве комментария. Если он не нужен, можно применить форму записи XLATB. Например, можно написать следующий вариант преобразования шестнадцатеричного числа в ASCII-код соответствующего ему символа:

```
mov     al,0Ch
mov     bx, offset htable
xlatb
```

если в сегменте данных, на который указывает регистр ES, было записано

```
htable  db      "0123456789ABCDEF"
```

то теперь AL содержит не число 0Ch, а ASCII-код буквы C. Разумеется, это преобразование разрешается выполнить посредством более компактного кода всего из трех арифметических команд, который будет рассмотрен в описании команды DAS, но с XLAT можно осуществить любые преобразования такого рода.

Команда	Назначение	Процессор
LEA приемник,источник	Вычисление эффективного адреса	8086

Вычисляет эффективный адрес источника (переменная) и помещает его в приемник (регистр). С помощью LEA можно вычислить адрес переменной, которая описана сложным методом адресации, например по базе с индексированием. Если адрес - 32-битный, а регистр-приемник - 16-битный, старшая половина вычисленного адреса теряется, если наоборот, приемник - 32-битный, а адресация - 16-битная, то вычисленное смещение дополняется нулями.

Команду LEA часто используют для быстрых арифметических вычислений, например умножения:

```
lea     bx,[ebx+ebx*4] ; BX = EBX x 5
```

или сложения:

```
lea     ebx,[eax+12]   ;EBX = EAX + 12
```

(эти команды меньше, чем соответствующие MOV и ADD, и не изменяют флаги)

23.2. Двоичная арифметика

Все команды этого раздела, кроме команд деления и умножения, изменяют флаги OF, SF, ZF, AF, CF, PF в соответствии с назначением каждого из них (см. раздел 2.1.4).

Команда	Назначение	Процессор
ADD приемник,источник	Сложение	8086

Команда выполняет арифметическое сложение приемника и источника, помещает сумму в приемник, не изменяя содержимое источника. Приемник может быть регистром или переменной, источник - числом, регистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника. Команда ADD никак не различает числа со знаком и без знака, но, употребляя значения флагов CF (перенос при сложении чисел без знака), OF (перенос при сложении чисел со знаком) и SF (знак результата), разрешается применять ее и для тех, и для других.

Команда	Назначение	Процессор
ADC приемник,источник	Сложение с переносом	8086

Эта команда аналогична ADD, но при этом выполняет арифметическое сложение приемника, источника и флага CF. Пара команд ADD/ADC используется для

сложения чисел повышенной точности. Сложим, например, два 64-битных целых числа. Пусть одно из них находится в паре регистров EpX:EAX (младшее двойное слово (биты 0-31) - в EAX и старшее (биты 32-63) - в EDX), а другое - в паре регистров EBX:ECX:

```
add     eax,ecx
adc     edx,ebx
```

Если при сложении младших двойных слов произошел перенос из старшего разряда (флаг CF = 1), то он будет учтен следующей командой ADC.

Команда	Назначение	Процессор
XADD приемник,источник	Обменять между собой и сложить	80486

Выполняет сложение, помещает содержимое приемника в источник, а сумму операндов - в приемник. Источник - всегда регистр, приемник может быть регистром и переменной.

Команда	Назначение	Процессор
SUB приемник,источник	Вычитание	8086

Вычитает источник из приемника и помещает разность в приемник. Приемник может быть регистром или переменной, источник - числом, регистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника. Точно так же, как и команда ADD, SUB не делает различий между числами со знаком и без знака, но флаги позволяют использовать ее и для тех, и для других.

Команда	Назначение	Процессор
SBB приемник,источник	Вычитание с займом	8086

Эта команда аналогична SUB, но она вычитает из приемника значение источника и дополнительно вычитает значение флага CE Ее можно использовать для вычитания 64-битных чисел в EDX:EAX и EBX:ECX аналогично ADD/ADC:

```
sub     eax,ecx
sbb     edx,ebx
```

Если при вычитании младших двойных слов произошел заем, то он будет учтен при вычитании старших слов.

Команда	Назначение	Процессор
IMUL источник	Умножение чисел со знаком	8086
IMUL приемник,источник		80386
IMUL приемник,источник1,источник2		80186

Эта команда имеет три формы, различающиеся числом операндов:

1. IMUL источник: источник (регистр или переменная) умножается на AL, AX или EAX (в зависимости от размера операнда), и результат располагается в AX, DX:AX или EDX:EAX соответственно.
2. IMUL приемник,источник: источник (число, регистр или переменная) умножается на приемник (регистр), и результат заносится в приемник.
3. IMUL приемник,источник1,источник2: источник 1 (регистр или переменная) умножается на источник 2 (число), и результат заносится в приемник (регистр).

Во всех трех вариантах считается, что результат может занимать в два раза больше места, чем размер источника. В первом случае приемник автоматически оказывается очень большим, но во втором и третьем случаях существует вероятность переполнения и потери старших битов результата. Флаги OF и CF будут равны единице, если это произошло, и нулю, если результат умножения поместился целиком в приемник (во втором и третьем случаях) или в младшую половину приемника (в первом случае).

Значения флагов SF, ZF, AF и PF после команды IMUL не определены.

Команда	Назначение	Процессор
MUL источник	Умножение чисел без знака	8086

Выполняет умножение содержимого источника (регистр или переменная) и регистра AL, AX, EAX (в зависимости от размера источника) и помещает результат в AX, DX:AX, EDX:EAX соответственно. Если старшая половина результата (AH, DX, EDX) содержит только нули (результат целиком поместился в младшую половину), флаги CF и OF устанавливаются в 0, иначе - в 1. Значение остальных флагов (SF, ZF, AF и PF) не определено.

Команда	Назначение	Процессор
IDIV источник	Целочисленное деление со знаком	8086

Выполняет целочисленное деление со знаком AL, AX или EAX (в зависимости от размера источника) на источник (регистр или переменная) и помещает результат в AL, AX или EAX, а остаток - в AH, DX или EDX соответственно. Результат всегда округляется в сторону нуля, знак остатка совпадает со знаком делимого, абсолютное значение остатка меньше абсолютного значения делителя. Флаги CF, OF, SF, ZF, AF и PF после этой команды не определены, а переполнение или деление на ноль вызывает исключение #DE (ошибка при делении) в защищенном режиме и прерывание 0 – в реальном.

Команда	Назначение	Процессор
DIV источник	Целочисленное деление без знака	8086

Выполняет целочисленное деление без знака AL, AX или EAX (в зависимости от размера источника) на источник (регистр или переменная) и помещает результат в AL, AX или EAX, а остаток - в AH, DX или EDX соответственно. Результат всегда

округляется в сторону нуля, абсолютное значение остатка меньше абсолютного значения делителя. Флаги CF, OF, SF, ZF, AF и PF после этой команды не определены, а переполнение или деление на ноль вызывает исключение #DE (ошибка при делении) в защищенном режиме и прерывание 0 – в реальном.

Команда	Назначение	Процессор
INC приемник	Инкремент	8086

Увеличивает приемник (регистр или переменная) на 1. Единственное отличие этой команды от ADD приемник,1 состоит в том, что флаг CF не затрагивается. Остальные арифметические флаги (OF, SF, ZF, AF, PF) устанавливаются в соответствии с результатом сложения.

Команда	Назначение	Процессор
DEC приемник	Декремент	8086

Уменьшает приемник (регистр или переменная) на 1. Единственное отличие этой команды от SUB приемник,1 заключается в том, что флаг CF не затрагивается. Остальные арифметические флаги (OF, SF, ZF, AF, PF) устанавливаются в соответствии с результатом вычитания.

Команда	Назначение	Процессор
NEG приемник	Изменение знака	8086

Выполняет над числом, содержащимся в приемнике (регистр или переменная), операцию дополнения до двух. Эта операция эквивалентна обращению знака операнда, если рассматривать его как число со знаком. Если приемник равен нулю, флаг CF устанавливается в 0, иначе - в 1. Остальные флаги (OF, SF, ZF, AF, PF) назначаются в соответствии с результатом операции.

Красивый пример использования команды NEG - получение абсолютного значения числа, применяя всего две команды - изменение знака и переход на первую команду еще раз, если знак отрицательный:

```
label0:  neg  eax
         js   labelO
```

Команда	Назначение	Процессор
CMP приемник,источник	Сравнение	8086

Сравнивает приемник и источник и устанавливает флаги. Действие осуществляется путем вычитания источника (число, регистр или переменная) из приемника (регистр или переменная; приемник и источник не могут быть переменными одновременно), причем результат вычитания никуда не записывается. Единственным следствием работы этой команды оказывается изменение флагов CF, OF, SF, ZF, AF и PF. Обычно команду CMP используют вместе с командами условного перехода (Jcc), условной пересылки данных (CMOVcc) или условной

установки байтов (SETcc), которые позволяют применить результат сравнения, не обращая внимания на детальное значение каждого флага. Так, команды CMOVE, JE и SETE выполнят соответствующие действия, если значения операндов предшествующей команды CMP были равны.

Несмотря на то что условные команды почти всегда вызываются сразу после CMP, не надо забывать, что их можно использовать после любой команды, модифицирующей флаги, например: проверить равенство AX нулю более короткой командой

```
test    ax,ax
```

а равенство единице - однобайтной командой

```
dec     ax
```

Команда	Назначение	Процессор
CMPXCHG приемник, источник	Сравнить и обменять между собой	80486

Сравнивает значения, содержащиеся в AL, AX, EAX (в зависимости от размера операндов), с приемником (регистром). Если они равны, информация из источника копируется в приемник и флаг ZF устанавливается в 1, в противном случае содержимое приемника копируется в AL, AX, EAX и флаг ZF устанавливается в 0. Остальные флаги определяются по результату операции сравнения, как после CMP. Источник - всегда регистр, приемник может быть регистром и переменной.

Команда	Назначение	Процессор
CMPXCHG8B приемник	Сравнить и обменять 8 байт	P5

Выполняет сравнение содержимого регистров EDX:EAX как 64-битного числа (младшее двойное слово - в EAX, старшее - в EDX) с приемником (8-байтная переменная в памяти). Если они равны, содержимое регистров ECX:EBX как 64-битное число (младшее двойное слово в EBX, старшее - в ECX) помещается в приемник. В противном случае содержимое приемника копируется в EDX:EAX.

2.3.3. Десятичная арифметика

Процессоры Intel поддерживают операции с двумя форматами десятичных чисел: неупакованное двоично-десятичное число - байт, принимающий значения от 00 до 09h, и упакованное двоично-десятичное число - байт, принимающий значения от 00 до 99h. Все обычные арифметические операции над такими числами приводят к неправильным результатам. Например, если увеличить 19h на 1, то получится число 1Ah, а не 20h. Для коррекции результатов арифметических действий над двоично-десятичными числами используются приведенные ниже команды.

Команда	Назначение	Процессор
DAA	BCD-коррекция после сложения	8086

Если эта команда выполняется сразу после ADD (ADC, INC или XADD) и в регистре AL находится сумма двух упакованных двоично-десятичных чисел, то в AL

записывается упакованное двоично-десятичное число, которое должно было стать результатом сложения. Например, если AL содержит число 19h, последовательность команд

```
inc     al
daa
```

приведет к тому, что в AL окажется 20h (а не 1Ah, как было бы после INC).

DAA выполняет следующие действия:

1. *Если младшие четыре бита AL больше 9 или флаг AF = 1, то AL увеличивается на 6, CF устанавливается, если при этом сложении произошел перенос, и AF устанавливается в 1.*
2. *Иначе AF = 0.*
3. *Если теперь старшие четыре бита AL больше 9 или флаг CF = 1, то AL увеличивается на 60h и CF устанавливается в 1.*
4. *Иначе CF = 0.*

Флаги AF и CF устанавливаются, если в ходе коррекции происходил перенос из первой или второй цифры. SF, ZF и PF устанавливаются в соответствии с результатом, флаг OF не определен.

Команда	Назначение	Процессор
DAS	BCD-коррекция после вычитания	8086

Если эта команда выполняется сразу после SUB (SBB или DEC) и в регистре AL находится разность двух упакованных двоично-десятичных чисел, то в AL записывается упакованное двоично-десятичное число, которое должно было быть результатом вычитания. Например, если AL содержит число 20h, последовательность команд

```
dec     al
das
```

приведет к тому, что в регистре окажется 19h (а не 1Fh, как было бы после DEC).

DAS выполняет следующие действия:

1. *Если младшие четыре бита AL больше 9 или флаг AF = 1, то AL уменьшается на 6, CF устанавливается, если при этом вычитании произошел заем, и AF устанавливается в 1.*
2. *Иначе AF = 0.*
3. *Если теперь старшие четыре бита AL больше 9 или флаг CF — 1, то AL уменьшается на 60h и CF устанавливается в 1.*
4. *Иначе CF = 0.*

Известный пример необычного использования этой команды ~ самый компактный вариант преобразования шестнадцатеричной цифры в ASCII-код соответствующего символа (более длинный и очевидный вариант этого преобразования рассматривался в описании команды XLAT):

```
cmp     al,10
sbb     al,69h
das
```

После SBB числа 0-9 превращаются в 96h - 9Fh, а числа 0Ah - 0Fh - в 0A1h - 0A6h. Затем DAS вычитает 66h из первой группы чисел, переводя их в 30h - 39h, и 60h из второй группы чисел, переводя их в 41h — 46h.

Флаги AF и CF устанавливаются, если в ходе коррекции происходил заем из первой или второй цифры. SF, ZF и PF устанавливаются в соответствии с результатом, флаг OF не определен.

Команда	Назначение	Процессор
AAA	ASCII-коррекция после сложения	8086

Корректирует сумму двух неупакованных двоично-десятичных чисел в AL. Если коррекция приводит к десятичному переносу, AH увеличивается на 1. Эту команду лучше использовать сразу после команды сложения двух таких чисел. Например, если при сложении 05 и 06 в AX окажется число 000Bh, то команда AAA скорректирует его в 0101h (неупакованное десятичное 11). Флаги CF и OF устанавливаются в 1, если произошел перенос из AL в AH, в противном случае они равны нулю. Значения флагов OF, SF, ZF и PF не определены.

Команда	Назначение	Процессор
AAS	ASCII-коррекция после вычитания	8086

Корректирует разность двух неупакованных двоично-десятичных чисел в AL сразу после команды SUB или SBB. Если операция приводит к займу, AH уменьшается на 1. Флаги CF и OF устанавливаются в 1, если произошел заем из AL в AH, и в ноль - в противном случае. Значения флагов OF, SF, ZF и PF не определены.

Команда	Назначение	Процессор
AAM	ASCII-коррекция после умножения	8086

Корректирует результат умножения неупакованных двоично-десятичных чисел, который находится в AX после выполнения команды MUL, преобразовывая полученное в пару неупакованных двоично-десятичных чисел (в AH и AL). Например:

```
mov     al,5
mov     bl,5            ; Умножить 5 на 5.
mul     bl              ; Результат в AX - 0019h.
aam                     ; Теперь AX содержит 0205h.
```

AAM устанавливает флаги SF, ZF и PF в соответствии с результатом и оставляет OF, AF и CF неопределенными.

Код команды AAM - D4h OAh, где OAh - основание системы счисления, по отношению к которой выполняется коррекция. Этот байт можно заменить на любое другое число (кроме нуля), и AAM преобразует AX к двум неупакованным цифрам любой системы счисления. Такая обобщенная форма AAM работает на всех процессорах (начиная с 8086), но появляется в документации Intel только с процессоров Pentium. Фактически действие, которое выполняет AAM, - целочисленное деление AL на OAh (или любое другое

число в общем случае), частное помещается в AH, и остаток - в AL, поэтому команду часто используют для быстрого деления в высокооптимизированных алгоритмах.

Команда	Назначение	Процессор
AAD	ASCII-коррекция перед делением	8086

Выполняет коррекцию неупакованного двоично-десятичного числа, находящегося в регистре AX, так, чтобы последующее деление привело к десятичному результату. Например, разделим десятичное 25 на 5:

```
mov     ax,0205h        ; 25 в неупакованном формате.
mov     bl,5
aad                     ; Теперь в AX находится 19h.
div     bl              ; AX = 0005.
```

Флаги SF, ZF и PF устанавливаются в соответствии с результатом, OF, AF и CF не определены.

Команда AAD, как и AAM, используется с любой системой счисления: ее код - D5h 0Ah, и второй байт можно заменить на любое другое число. Действие AAD заключается в том, что содержимое регистра AH умножается на второй байт команды (0Ah по умолчанию) и складывается с AL, после чего AH обнуляется, так что AAD можно использовать для быстрого умножения на любое число.

2.3.4. Логические операции

Команда	Назначение	Процессор
AND приемник,источник	Логическое И	8086

Команда выполняет побитовое «логическое И» над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и помещает результат в приемник. Любой бит результата равен 1, только если соответствующие биты обоих операндов были равны 1, и равен 0 в остальных случаях. Наиболее часто AND применяют для выборочного обнуления отдельных битов. Например, команда

```
and     al,00001111b
```

обнулит старшие четыре бита регистра AL, сохранив неизменными четыре младших.

Флаги OF и CF обнуляются, SF, ZF и PF устанавливаются в соответствии с результатом, AF не определен.

Команда	Назначение	Процессор
OR приемник,источник	Логическое ИЛИ	8086

Выполняет побитовое «логическое ИЛИ» над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не

могут быть переменными одновременно) и помещает результат в приемник. Любой бит результата равен 0, только если соответствующие биты обоих операндов были равны 0, и равен 1 в остальных случаях. Команду OR чаще всего используют для выборочной установки отдельных битов. Например, команда

```
or      al,00001111b
```

приведет к тому, что младшие четыре бита регистра AL будут установлены в 1.

При выполнении команды OR флаги OF и CF обнуляются, SF, ZF и PF устанавливаются в соответствии с результатом, AF не определен.

Команда	Назначение	Процессор
XOR приемник,источник	Логическое исключающее ИЛИ	8086

Выполняет побитовое «логическое исключающее ИЛИ» над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и помещает результат в приемник. Любой бит результата равен 1, если соответствующие биты операндов различны, и нулю - в противном случае. XOR используется для самых разных операций, например:

```
        xor     ax,ax           ; Обнуление регистра AX.
или
        xor     ax,bx
        xor     bx,ax
        xor     ax,bx           ; Меняет местами содержимое AX и BX.
```

Оба примера могут выполняться быстрее, чем соответствующие очевидные команды

```
        mov     ax,0
или
        xchg    ax,bx
```

Команда	Назначение	Процессор
NOT приемник	Инверсия	8086

Каждый бит приемника (регистр или переменная), равный нулю, устанавливается в 1, и каждый бит, равный 1, сбрасывается в 0. Флаги не затрагиваются.

Команда	Назначение	Процессор
TEST приемник, источник	Логическое сравнение	8086

Вычисляет результат действия побитового «логического И» над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и устанавливает флаги SF, ZF и PF в соответствии с полученным показателем, не сохраняя результата (флаги OF и CF обнуляются, значение AF не определено). TEST, так же как и CMP, используется в основном в сочетании с командами условного перехода (Jcc), условной пересылки данных (CMOVcc) и условной установки байтов (SETcc).

2.3.5. *Сдвиговые операции*

Команда	Назначение	Процессор
SAR приемник,счетчик	Арифметический сдвиг вправо	8086
SAL приемник,счетчик	Арифметический сдвиг влево	8086
SHR приемник,счетчик	Логический сдвиг вправо	8086
SHL приемник,счетчик	Логический сдвиг влево	8086

Эти четыре команды выполняют двоичный сдвиг приемника (регистр или переменная) вправо (в сторону младшего бита) или влево (в сторону старшего бита) на значение счетчика (число или регистр CL, из которого учитываются только младшие 5 бит, принимающие значения от 0 до 31). Операция сдвига на 1 эквивалентна умножению (сдвиг влево) или делению (сдвиг вправо) на 2. Так, число 0010b (2) после сдвига на 1 влево превращается в 0100b (4). Команды SAL и SHL выполняют одну и ту же операцию (на самом деле это одна и та же команда) - на каждый шаг сдвига старший бит заносится в CF, все биты сдвигаются влево на одну позицию, и младший бит обнуляется. Команда SHR осуществляет прямо противоположную операцию: младший бит заносится в CF, все биты сдвигаются на 1 вправо, старший бит обнуляется. Эта команда эквивалентна беззнаковому целочисленному делению на 2. Команда SAR действует по аналогии с SHR, только старший бит не обнуляется, а сохраняет предыдущее значение, вот почему, например, число 11111100b (-4) перейдет в 11111110b (-2). SAR, таким образом, эквивалентна знаковому делению на 2, но, в отличие от IDIV, округление происходит не в сторону нуля, а в сторону отрицательной бесконечности. Так, если разделить –9 на 4 с помощью IDIV, получится -2 (и остаток –1), а если выполнить арифметический сдвиг вправо числа -9 на 2, результатом будет -3. Сдвиги больше 1 эквивалентны соответствующим сдвигам на 1, выполненным последовательно. Схема всех сдвиговых операций приведена на рис. 7.

Сдвиги на 1 изменяют значение флага OF: SAL/SHL устанавливают его в 1, если после сдвига старший бит изменился (то есть старшие два бита исходного числа не были одинаковыми), и в 0, если старший бит остался тем же. SAR устанавливает OF в 0, а SHR -- в значение старшего бита исходного числа. Для сдвигов на несколько битов значение OF не определено. Флаги SF, ZF, PF назначаются

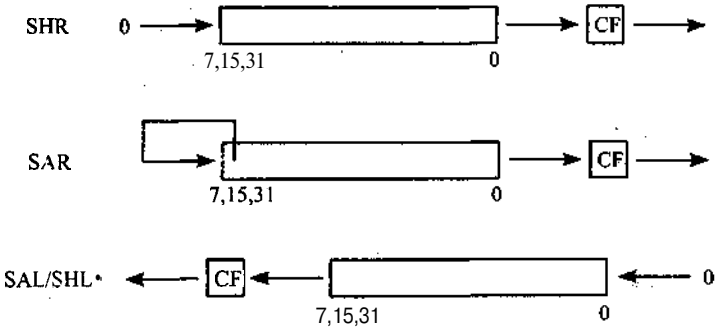

Рис. 7. Сдвиговые операции

всеми сдвигами в соответствии с результатом, параметр AF не определен (кроме случая, когда счетчик сдвига равен нулю: ничего не происходит и флаги не изменяются).

В процессорах 8086 в качестве второго операнда можно было задавать лишь число 1 и при использовании CL учитывать все биты, а не только младшие 5, но уже начиная с 80186 эти команды приняли свой окончательный вид.

Команда	Назначение	Процессор
SHRD приемник,источник,счетчик	Сдвиг повышенной точности вправо	80386
SHLD приемник,источник,счетчик	Сдвиг повышенной точности влево	80386

Приемник (регистр или переменная) сдвигается влево (в случае SHLD) или вправо (в случае SHRD) на число битов, указанное в счетчике (число или регистр CL, откуда используются только младшие 5 бит, принимающие значения от 0 до 31). Старший (для SHLD) или младший (в случае SHRD) бит не обнуляется, а считывается из источника (регистр), значение которого не изменяется. Например, если приемник содержит 00101001b, источник - 1010b, то счетчик равен 3, SHRD даст в результате 01000101b, а SHLD - 01001101b (см. рис. 8).

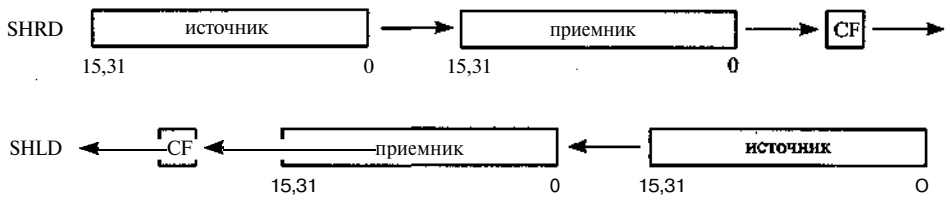

Рис. 8. Сдвиги двойной точности

Флаг OF устанавливается при сдвигах на 1 бит, если изменился знак приемника, и сбрасывается в противном случае; при сдвигах на несколько битов флаг OF не определен. Во всех случаях SF, ZF и PF устанавливаются в соответствии с результатом и AF не определен, кроме варианта со сдвигом на 0 бит, в котором значения флагов не изменяются. Если счетчик больше, чем разрядность приемника, - результат и все флаги не определены.

Команда	Назначение	Процессор
ROR приемник,счетчик	Циклический сдвиг вправо	8086
ROL приемник,счетчик	Циклический сдвиг влево	8086
RCR приемник,счетчик	Циклический сдвиг вправо через флаг переноса	8086
RCL приемник,счетчик	Циклический сдвиг влево через флаг переноса	8086

Эти команды осуществляют циклический сдвиг приемника (регистр или переменная) на число битов, указанное в счетчике (число или регистр CL, из которого учитываются только младшие 5 бит, принимающие значения от 0 до 31). При выполнении циклического сдвига на 1 команды ROR (ROL) перемещают каждый бит приемника вправо (влево) на одну позицию, за исключением самого младшего (старшего), который записывается в позицию самого старшего (младшего) бита.

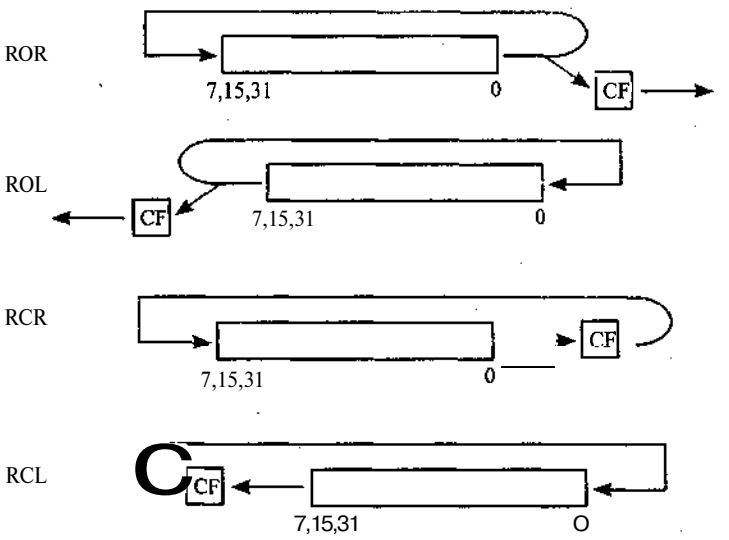

Рис. 9. Циклические сдвиги

Команды ,RCR и RCL выполняют аналогичное действие, но включают флаг CF в цикл, как если бы он был дополнительным битом в приемнике (см. рис. 9).

После выполнения команд циклического сдвига флаг CF всегда равен последнему вышедшему за пределы приемника биту, флаг OF определен только для сдвигов на 1 - он устанавливается, если изменилось значение самого старшего бита, и сбрасывается в противном случае. Флаги SF, ZF, AF и PF не изменяются.

2.3.6. Операции над битами и байтами

Команда	Назначение	Процессор
BT база,смещение	Проверка бита	80386

Команда BT считывает в флаг CF значение бита из битовой строки, определенной первым операндом - *битовой базой* (регистр или переменная), со смещением, указанным во втором операнде - *битовом смещении* (число или регистр). Когда первый операнд - регистр, то битовой базой считается бит 0 в названном регистре и смещение не может превышать 15 или 31 (в зависимости от размера регистра); если оно превышает эти границы, в качестве смещения будет использоваться остаток от деления на 16 или 32 соответственно. Если первый операнд - переменная, то в качестве битовой базы нужен бит 0 указанного байта в памяти, а смещение может принимать значения от 0 до 31, если оно установлено непосредственно (старшие биты процессором игнорируются), и от -2^{31} до $2^{31}-1$, если оно указано в регистре.

Несмотря на то что эта команда считывает единственный бит из памяти, а процессор - целое двойное слово по адресу База + (4 × (Смещение/32)) или слово по адресу База + (2 x (Смещение/16)),в зависимости от разрядности адреса, все равно не следует пользоваться BT вблизи от недоступных для чтения областей памяти.

После выполнения команды BT флаг CF равен значению считанного бита, флаги OF, SF, ZF, AF и PF не определены.

Команда	Назначение	Процессор
BTS база,смещение	Проверка и установка бита	80386
BTR база,смещение	Проверка и сброс бита	80386
BTC база,смещение	Проверка и инверсия бита	80386

Эти три команды соответственно устанавливают в 1 (BTS), сбрасывают в 0 (BTR) и инвертируют (BTC) значение бита, который находится в битовой строке с началом, определенным в базе (регистр или переменная), и смещением, указанным во втором операнде (число от 0 до 31 или регистр). Если битовая база - регистр, то смещение не может превышать 15 или 31 в зависимости от разрядности этого регистра. Если битовая база - переменная в памяти, то смещение может принимать значения от -2^{31} до $2^{31}-1$ (при условии, что оно указано в регистре).

После выполнения команд BTS, BTR и BTC флаг CF равен значению считанного бита до его изменения в результате действия команды, флаги OF, SF, ZF, AF и PF не определены.

Команда	Назначение	Процессор
BSF приемник,источник	Прямой поиск бита	80386
BSR приемник,источник	Обратный поиск бита	80386

BSF сканирует источник (регистр или переменная), начиная с самого младшего бита, и записывает в приемник (регистр) номер первого встретившегося бита, равного 1. Команда BSR сканирует источник, начиная с самого старшего бита, и возвращает номер первого встретившегося ненулевого бита, считая от нуля. То есть, если источник равен 0000 0000 0000 0010b, то BSF возвратит 1, а BSR - 14.

Если весь источник равен нулю, значение приемника не определено и флаг ZF устанавливается в 1, иначе ZF всегда сбрасывается. Флаги CF, OF, SF, AF и PF не определены.

Команда	Назначение	Процессор
SETcc приемник	Установка байта по условию	80386

Это набор команд, устанавливающих приемник (8-битный регистр или переменная размером в 1 байт) в 1 или 0, если удовлетворяется *или* не удовлетворяется определенное условие. Фактически в каждом случае проверяется состояние тех или иных флагов, но, когда команда из набора SETcc используется сразу после CMP, условия приобретают формулировки, соответствующие отношениям между операндами CMP (см. табл. 6). Скажем, если операнды CMP были неравны, то команда SETNE, выполненная сразу после CMP, установит значение своего операнда в 1.

Слова «выше» и «ниже» в таблице относятся к сравнению чисел без знака, слова «больше» и «меньше» учитывают знак.

Таблица 6. Команды SETcc

Код команды	Реальное условие	Условие для CMP
SETA SETNBE	CF = 0 и ZF = 0	Если выше Если не ниже и не равно
SETAE SETNB SETNC	CF = 0	Если выше или равно Если не ниже Если нет переноса
SETB SETNAE SETC	CF = 1	Если ниже Если не выше и не равно Если перенос
SETBE SETNA	CF = 1 или ZF = 1	Если ниже или равно Если не выше
SETE SETZ	ZF = 1	Если равно Если ноль
SETG SETNLE	ZF = 0 и SF = OF	Если больше Если не меньше и не равно
SETGE SETNL	SF = OF	Если больше или равно Если не меньше
SETL SETNGE	SF O OF	Если меньше Если не больше и не равно
SETLE SETNG	ZF = 1 или SF <> OF	Если меньше или равно Если не больше
SETNE SETNZ	ZF = 0	Если не равно Если не ноль
SETNO	OF = 0	Если нет переполнения
SETO	OF = 1	Если есть переполнение
SETNP SETPO	PF = 0	Если нет четности Если нечетное
SETP SETPE	PF = 1	Если есть четность Если четное
SETNS	SF = 0	Если нет знака
SETS	SF = 1	Если есть знак

2.3.7. *Команды передачи управления*

Команда	Назначение	Процессор
JMP операнд	Безусловный переход	8086

JMP передает управление в другую точку программы, не сохраняя какой-либо информации для возврата. Операндом может быть непосредственный адрес для перехода (в программах используют имя метки, установленной перед командой, на которую выполняется переход), а также регистр или переменная, содержащая адрес,

В зависимости от типа перехода различают:

а переход типа *short* (короткий переход) - если адрес перехода находится в пределах –128...+127 байт от команды JMP;

а переход типа *near* (ближний переход) - если адрес перехода находится в том же сегменте памяти, что и команда JMP;

- переход типа *far* (дальний переход) - если адрес перехода находится в другом сегменте. Дальний переход может выполняться и в тот же самый сегмент при условии, что в сегментной части операнда указано число, совпадающее с текущим значением CS;
- переход с переключением задачи - передача управления другой задаче в многозадачной среде. Этот вариант будет рассмотрен в разделе, посвященном защищенномурежиму.

При выполнении переходов типа short и near команда JMP фактически преобразовывает значение регистра EIP (или IP), изменяя тем самым смещение следующей исполняемой команды относительно начала сегмента кода. Если операнд - регистр или переменная в памяти, то его показатель просто копируется в EIP, как если бы это была команда MOV. Если операнд для JMP - непосредственно указанное число, то его значение суммируется с содержимым EIP, приводя к относительному переходу. В ассемблерных программах в качестве операнда обычно указывают имена меток, но на уровне исполняемого кода ассемблер вычисляет и записывает именно относительные смещения.

Выполняя дальний переход в реальном, виртуальном и защищенном режимах (при переходе в сегмент с теми же привилегиями), команда JMP просто загружает новое значение в EIP и новый селектор сегмента кода в CS, используя старшие 16 бит операнда как новое значение для CS и младшие 16 или 32 бит в качестве значений IP или EIP.

Команда	Назначение	Процессор
Jcc метка	Условный переход	8086

Это набор команд, выполняющих переход (типа short или near), если удовлетворяется соответствующее условие, которым в каждом случае реально является состояние тех или иных флагов. Но, когда команда из набора Jcc используется сразу после CMP, условия приобретают формулировки, соответствующие отношениям между операндами CMP (см. табл. 7). Например, если операнды CMP были равны, то команда JE, выполненная сразу после CMP, осуществит переход. Операнд для всех команд из набора Jcc - 8-битное или 32-битное смещение относительно текущей команды.

Слова «выше» и «ниже» в таблице относятся к сравнению чисел без знака; слова «больше» и «меньше» учитывают знак.

Команды Jcc не поддерживают дальних переходов, поэтому, если требуется выполнить условный переход на дальнюю метку, необходимо использовать команду из набора Jcc с обратным условием и дальний JMP, как, например:

```
        cmp     ax,0
        jne     local_1
        jmp     far_label       ; Переход, если AX = 0.
local_1:
```

Таблица 7. Варианты команды Jcc

Код команды	Реальное условие	Условие для CMP
JA JBE	CF = 0 и ZF = 0	Если выше Если не ниже и не равно
JAE JNB JNC	CF = 0	Если выше или равно Если не ниже Если нет переноса
JB JNAE JC	CF = 1	Если ниже Если не выше и но равно Если перенос
JBE JNA	CF = 1 или ZF = 1	Если ниже или равно Если не выше
JE JZ	ZF = 1	Если равно Если ноль
JG JNLE	ZF = 0 и SF = OF	Если больше Если не меньше и не равно
JGE JNL	SF = OF	Если больше или равно Если не меньше
JL JNGE	SF O OF	Если меньше Если не больше и не равно
JLE JNG	ZF = 1 или SF 0 OF	Если меньше или равно Если не больше
JNE JNZ	ZF = 0	Если не равно Если не ноль
JNO	OF = 0	Если нет переполнения
JO	OF = 1	Если есть переполнение
JNP JPO	PF = 0	Если нет четности Если нечетное
JP JPE	PF = 1	Если есть четность Если четное
JNS	SF = 0	Если нет знака
JS	SF = 1	Если есть знак

Команда	Назначение	Процессор
JCXZ метка	Переход, если CX = 0	8086
JECXZ метка	Переход, если ECX = 0	80386

Выполняет ближний переход на указанную метку, если регистр CX или ECX (для JCXZ и JECXZ соответственно) равен нулю. Так же как и команды из серии Jcc, JCXZ и JECXZ не могут выполнять дальних переходов. Проверка равенства CX нулю, например, может потребоваться в начале цикла, организованного командой LOOPNE, - если в него войти с CX = 0, то он будет выполнен 65 535 раз.

Команда	**Назначение**	**Процессор**
LOOP метка	Цикл	8086

Уменьшает регистр ECX на 1 и выполняет переход типа short на метку (которая не может быть дальше расстояния -128...+ 127 байт от команды ЮОР), если ECX не равен нулю. Эта команда используется для организации циклов, в которых регистр ECX (или CX при 16-битной адресации) играет роль счетчика. Так, в следующем фрагменте команда ADD выполнится 10 раз:

```
        mov     cx,0Ah
loop_start:
        add     ax,cx
        loop    loop_start
```

Команда LOOP полностью эквивалентна паре команд

```
        dec     ecx
        jnz     метка
```

Но LOOP короче этих двух команд на один байт и не изменяет значения флагов.

Команда	Назначение	Процессор
LOOPE метка	Цикл, пока равно	8086
LOOPZ метка	Цикл, пока ноль	8086
LOOPNE метка	Цикл, пока не равно	8086
LOOPNZ метка	Цикл, пока не ноль	8086

Все перечисленные команды уменьшают регистр ECX на один, после чего выполняют переход типа short, если ECX не равен нулю и если выполняется условие. Для команд LOOPE и LOOPZ условием является равенство единице флага ZF, для команд LOOPNE и LOOPNZ - равенство флага ZF нулю. Сами команды LOOPcc не изменяют значений флагов, так что ZF должен быть установлен (или сброшен) предшествующей командой. Например, следующий фрагмент копирует строку из DS:SI в строку в ES:DI (см. описание команд работы со строками), пока не кончится строка (CX = 0) или пока не встретится символ с ASCII-кодом 13 (конец строки):

```
        mov     cx,str_length
move_loop:
        lodsb
        stosb
        cmp     al,13
        loopnz  move_loop
```

Команда	Назначение	Процессор
CALL операнд	Вызов процедуры	8086

Сохраняет текущий адрес в стеке и передает управление по адресу, указанному в операнде. Операндом может быть непосредственное значение адреса (метка в ассемблерных программах), регистр или переменная, содержащие адрес перехода. Если в качестве*адреса перехода указано только смещение, считается, что адрес расположен в том же сегменте, что и команда CALL. При этом, так же как и в случае с JMP, выполняется ближний вызов процедуры. Процессор помещает значение регистра EIP (IP при 16-битной адресации), соответствующее следующей за CALL команде, в стек и загружает ь EIP новое значение, осуществляя тем

самым передачу управления. Если операнд CALL - регистр или переменная, то его значение рассматривается как абсолютное смещение, если операнд - ближняя метка в программе, то ассемблер указывает ее относительное смещение. Чтобы выполнить дальний CALL в реальном режиме, режиме V86 или в защищенном режиме при переходе в сегмент с теми же привилегиями, процессор помещает в стек значения регистров CS и EIP (IP при 16-битной адресации) и осуществляет дальний переход аналогично команде JMP.

Команда	Назначение	Процессор
RET число	Возврат из процедуры	8086
RETN число		
RETF число		

RETN считывает из стека слово (или двойное слово, в зависимости от режима адресации) и загружает его в IP (или EIP), выполняя тем самым действия, обратные ближнему вызову процедуры командой CALL. Команда RETF загружает из стека IP (EIP) и CS, возвращаясь из дальней процедуры. Если в программе указана команда RET, ассемблер заменит ее на RETN или RETF в зависимости от того, как была описана процедура, которую эта команда завершает. Операнд для RET необязателен, но, если он присутствует, после считывания адреса возврата из стека будет удалено указанное количество байтов - это нужно, если при вызове процедуры ей передавались параметры через стек.

Команда	Назначение	Процессор
INT число	Вызов прерывания	8086

INT аналогично команде CALL помещает в стек содержимое регистров EFLAGS, CS и EIP, после чего передает управление программе, называемой *обработчиком прерываний с* указанным в качестве операнда номером (число от 0 до 0FFh). В реальном режиме адреса обработчиков прерываний считываются из таблицы, начинающейся в памяти по адресу 0000h:0000h. Адрес каждого обработчика занимает 4 байта, вот почему, например, обработчик прерывания 10h находится в памяти по адресу 0000h:0040h. В защищенном режиме адреса обработчиков прерываний находятся в таблице IDT и обычно недоступны для прямого чтения или записи, так что для установки собственного обработчика программа должна обращаться к операционной системе. В DOS вызовы прерываний используются для выполнения большинства системных функций - работы с файлами, вводом/выводом и т. д. Например, следующий фрагмент кода завершает выполнение программы и возвращает управление DOS:

```
mov     ax,4C01h
int     21h
```

Команда	Назначение	Процессор
IRET	Возврат из обработчика прерывания	8086
IRETD		

Возврат управления из обработчика прерывания или исключения. IRET загружает из стека значения IP, CS и FLAGS, а IRETD - EIP, CS и EFLAGS соответственно. Единственное отличие IRET от RETF состоит в том, что значение регистра флагов восстанавливается, из-за чего многим обработчикам прерываний приходится изменять величину EFLAGS, находящегося в стеке, чтобы, например, вернуть флаг CF, установленный в случае ошибки.

Команда	Назначение	Процессор
INT3	Вызов прерывания 3	8086

Размер этой команды - один байт (код 0CCh), что делает ее удобной для доводки программ отладчиками, работающими в реальном режиме. Они записывают этот байт вместо первого байта команды, перед которой требуется точка останова, и переопределяют адрес обработчика прерывания 3 на соответствующую процедуру внутри отладчика.

Команда	Назначение	Процессор
INTO	Вызов прерывания 4 при переполнении	8086

INTO - еще одна специальная форма команды INT. Она вызывает обработчик прерывания 4, если флаг OF установлен в 1.

Команда	Назначение	Процессор
BOUND индекс,границы	Проверка выхода за границы массива	80186

BOUND проверяет, не выходит ли значение первого операнда (регистр), взятое как число со знаком, за границы, указанные во втором операнде (переменная). Границы - два слова или двойных слова (в зависимости от разрядности операндов), рассматриваемые как целые со знаком и расположенные в памяти подряд. Первая граница считается нижней, вторая - верхней. Если индекс меньше нижней границы или больше верхней, вызывается прерывание 5 (или исключение #BR), причем адрес возврата указывает не на следующую команду, а на BOUND, так что обработчик должен исправить значение индекса или границ, прежде чем выполнять команду IRET.

Команда	Назначение	Процессор
ENTER размер,уровень	Вход в процедуру	80186

Команда ENTER создает стековый кадр заданного размера и уровня вложенности (оба операнда - числа; уровень вложенности может принимать значения только от 0 до 31) с целью вызова процедуры, использующей динамическое распределение памяти в стеке для своих локальных переменных. Так, команда

```
enter   2048,3
```

помещает в стек указатели на стековый кадр текущей процедуры и той, из которой вызывалась текущая, создает стековый кадр размером 2 Кб для вызываемой

процедуры и помещает в EBP адрес начала кадра. Пусть процедура MAIN имеет уровень вложенности 0, процедура PROCA запускается из MAIN и имеет уровень вложенности 1, и PROCB запускается из PROCA с уровнем вложенности 2. Тогда стек при входе в процедуру МАШ имеет вид, показанный на рис. 10.

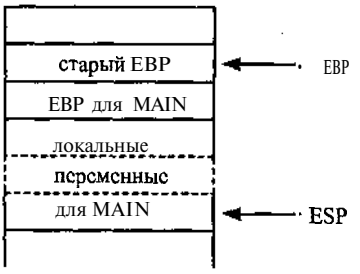

Рис. 10. Стековый кадр процедуры нулевого уровня (MAIN)

Теперь процедура МАШ может определять свои локальные переменные в памяти, используя текущее значение EBP.

На первом уровне вложенности процедура PROCA, как показано на рис. 11, может создавать свои локальные переменные, применяя текущее значение EBP, и получит доступ к локальным переменным процедуры МАШ, используя значение EBP для МАШ, помещенное в стек командой ENTER.

Процедура PROCB на втором уровне вложенности (см. рис. 12) получает доступ как к локальным переменным процедуры PROCA, применяя значение EBP для PROCA, так и к локальным переменным процедуры МАШ, используя значение EBP для МАШ.

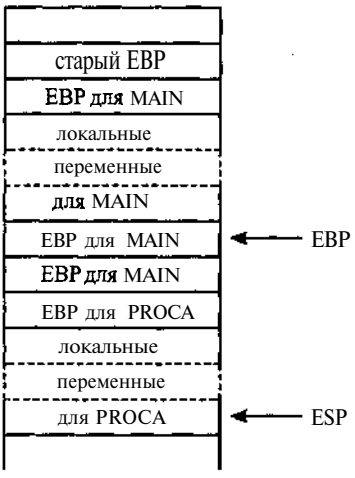

Рис. 11. Стековый кадр процедуры первого уровня (PROCA)

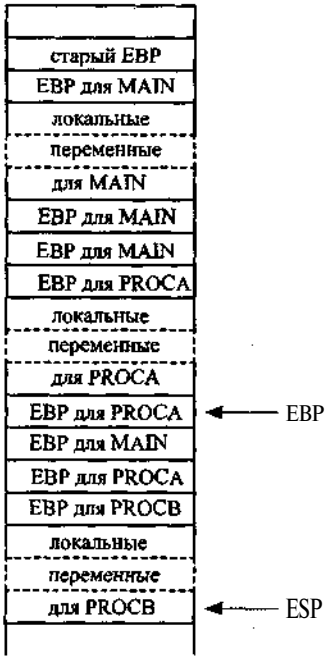

Рис. 12. Стековый кадр процедуры второго уровня (PROCB)

Команда	Назначение	Процессор
LEAVE	Выход из процедуры	80186

Команда выполняет действия, противоположные команде ENTER. Фактически LEAVE только копирует содержимое EBP в ESP, тем самым выбрасывая из стека

весь кадр, созданный последней выполненной командой ENTER, и считывает из стека EBP для предыдущей процедуры, что одновременно восстанавливает и значение, которое имел ESP до вызова последней команды ENTER.

2.3.8. Строковые операции

Все команды для работы со строками считают, что строка-источник находится по адресу DS:SI (или DS:ESI), то есть в сегменте памяти, указанном в DS Со смещением в SI, а строка-приемник - соответственно в ES:DI (или ES:EDI). Кроме того, все строковые команды работают только с одним элементом строки (байтом, словом или двойным словом) за один раз. Для того чтобы команда выполнялась над всей строкой, необходим один из префиксов повторения операций.

Команда	Назначение	Процессор
REP	Повторять	8086
REPE	Повторять, пока равно	8086
REPNE	Повторять, пока не равно	8086
REPZ	Повторять, пока ноль	8086
REPNZ	Повторять, пока не ноль	8086

Все перечисленные команды являются префиксами для операций над строками. Любой из префиксов выполняет следующую за ним команду строковой обработки столько раз, сколько указано в регистре ECX (или CX, в зависимости от разрядности адреса), уменьшая его при каждом выполнении команды на 1. Кроме того, REPZ и REPE прекращают повторения команды, если флаг ZF сброшен в О, а REPNZ и REPNE прекращают повторения, если флаг ZF установлен в 1. Префикс REP обычно используется с командами INS, OUTS, MOVS, LODS, $TOS, а префиксы REPE, REPNE, REPZ и REPNZ - с командами CMPS и SCAS. Поведение префиксов в других случаях не определено.

Команда	Назначение	Процессор
MOVS приемник,источник	Копирование строки	8086
MOVSB	Копирование строки байтов	8086
MOVSW	Копирование строки слов	8086
MOVSD	Копирование строки двойных слов	80386

Копирует один байт (MOVSB), слово (MOVSW) или двойное слово (MOVSD) из памяти по адресу DS:ESI (или DS:SI, в зависимости от разрядности адреса) в память по адресу ES:EDI (или ES:DI). При использовании формы записи MOVS ассемблер сам определяет по типу указанных операндов (принято указывать имена копируемых строк, но можно применять любые два операнда подходящего типа), какую из трех форм этой команды (MOVSB, MOVSW или MOVSD) выбрать. Используя MOVS с операндами, разрешается заменить регистр DS другим с помощью префикса замены сегмента (ES:, GS:, FS:, CS:, SS:), регистр ES заменить нельзя. После выполнения команды регистры ESI (или SI) и EDI (или DI) увеличиваются на 1, 2 или 4 (если копируются байты, слова или двойные слова), когда флаг DF = О,

и уменьшаются, когда DF = 1. Команда MOVS с префиксом REP выполняет копирование строки длиной в ECX (или CX) байтов, слов или двойных слов.

Команда	Назначение	Процессор
CMPS приемник,источник	Сравнение строк	8086
CMPSB	Сравнение строк байтов	8086
CMPSW	Сравнение строк слов	8086
CMPSD	Сравнение строк двойных слов	80386

Сравнивает один байт (CMPSB), слово (CMPSW) или двойное слово (CMPSD) из памяти по адресу DS:ESI (или DS:SI, в зависимости от разрядности адреса) с байтом, словом или двойным словом по адресу ES:EDI (или ES:DI) и устанавливает флаги аналогично команде CMP. При использовании формы записи CMPS ассемблер сам определяет по типу указанных операндов (принято указывать имена сравниваемых строк, но можно использовать любые два операнда подходящего типа), какую из трех форм этой команды (CMPSB, CMPSW или CMPSD) выбрать. Применяя CMPS с операндами, можно заменить регистр DS другим, воспользовавшись префиксом замены сегмента (ES:, GS:, FS:, CS:, SS:), регистр ES заменить нельзя. После выполнения команды регистры ESI (или SI) и EDI (или DI) увеличиваются на 1, 2 или 4 (если сравниваются байты, слова или двойные слова), когда флаг DF = 0, и уменьшаются, когда DF = 1. Команда CMPS с префиксами REPNE/REPNZ или REPE/REPZ выполняет сравнение строки длиной в ECX (или CX) байтов, слов или двойных слов. В первом случае сравнение продолжается до первого совпадения в строках, а во втором — до первого несовпадения.

Команда	Назначение	Процессор
SCAS приемник	Сканирование строки	8086
SCASB	Сканирование строки байтов	8086
SCASW	Сканирование строки слов	8086
SCASD	Сканирование строки двойных слов	80386

Сравнивает содержимое регистра AL (SCASB), AX (SCASW) или EAX (SCASD) с байтом, словом или двойным словом из памяти по адресу ES:EDI (или ES:DI, в зависимости от разрядности адреса) и устанавливает флаги аналогично команде CMP. При использовании формы записи SCAS ассемблер сам определяет по типу указанного операнда (принято указывать имя сканируемой строки, но можно использовать любой операнд подходящего типа), какую из трех форм этой команды (SCASE, SCASW или SCASD) выбрать. После выполнения команды регистр EDI (или DI) увеличивается на 1, 2 или 4 (если сканируются байты, слова или двойные слова), когда флаг DF = 0, и уменьшается, когда DF = 1. Команда SCAS с префиксами REPNE/REPNZ или REPE/REPZ выполняет сканирование строки длиной в ECX (или CX) байтов, слов или двойных слов. В первом случае сканирование продолжается до первого элемента строки, совпадающего с содержимым аккумулятора, а во втором - до первого отличного.

Команда	Назначение	Процессор
LODS источник	Чтение из строки	8086
LODSB	Чтение байта из строки	8086
LODSW	Чтение слова из строки	8086
LODSD	Чтение двойного слова из строки	80386

Копирует один байт (LODSB), слово (LODSW) или двойное слово (LODSD) из памяти по адресу DS:ESI (или DS:SI, в зависимости от разрядности адреса) в регистр AL, AX или EAX соответственно. При использовании формы записи LODS ассемблер сам определяет по типу указанного операнда (принято указывать имя строки, но можно использовать любой операнд подходящего типа), какую из трех форм этой команды (LODSB, LODSW или LODSD) выбрать. Применяя LODS с операндом, можно заменить регистр DS на другой с помощью префикса замены сегмента (ES:, GS:, FS:, CS:, SS:). После выполнения команды регистр ESI (или SI) увеличивается на 1, 2 или 4 (если считывается байт, слово или двойное слово), когда флаг DF = 0, и уменьшается, когда DF = 1. Команда LODS с префиксом REP выполнит копирование строки длиной в ECX (или CX), и в аккумуляторе окажется последний элемент строки. На самом деле LODS используют без префиксов, часто внутри цикла в паре с командой STOS, так что LODS считывает число, другие команды выполняют над ним какие-нибудь действия, а затем STOS записывает измененное число на прежнее место в памяти.

Команда	Назначение	Процессор
STOS приемник	Запись в строку	8086
STOSB	Запись байта в строку	8086
STOSW	Запись слова в строку	8086
STOSD	Запись двойного слова в строку	80386

Копирует регистр AL (STOSB), AX (STOSW) или EAX (STOSD) в память по адресу ES:EDI (или ES:DI, в зависимости от разрядности адреса). При использовании формы записи STOS ассемблер сам определяет по типу указанного операнда (принято указывать имя строки, но можно использовать любой операнд подходящего типа), какую из трех форм этой команды (STOSB, STOSW или STOSD) выбрать. После выполнения команды регистр EDI (или DI) увеличивается на 1, 2 или 4 (если копируется байт, слово или двойное слово), когда флаг DF = 0, и уменьшается, когда DF = 1. Команда STOS с префиксом REP заполнит строку длиной в ECX (или CX) числом, находящимся в аккумуляторе.

Команда	Назначение	Процессор
INS источник,DX	Чтение строки из порта	80186
INSB	Чтение строки байт из порта	80186
INSW	Чтение строки слов из порта	80186
INSD	Чтение строки двойных слов из порта	80386

Считывает из порта ввода-вывода, номер которого указан в регистре DX, байт (INSB), слово (INSW) или двойное слово (INSD) в память по адресу ES:EDI (или ES:DI, в зависимости от разрядности адреса). При использовании формы записи INS ассемблер определяет по типу указанного операнда, какую из трех форм этой команды (INSB, INSW или INSD) употребить. После выполнения команды регистр EDI (или DI) увеличивается на 1, 2 или 4 (если считывается байт, слово или двойное слово), когда флаг DF = 0, и уменьшается, когда DF = 1. Команда INS с префиксом REP считывает блок данных из порта длиной в ECX (или CX) байтов, слов или двойных слов.

Команда	Назначение	Процессор
OUTS DX,приемник	Запись строки в порт	80186
OUTSB	Запись строки байтов в порт	80186
OUTSW	Запись строки слов в порт	80186
OUTSD	Запись строки двойных слов в порт	80386

Записывает в порт ввода-вывода, номер которого указан в регистре DX, байт (OUTSB), слово (OUTSW) или двойное слово (OUTSD) из памяти по адресу DS:ESI (или DS:SI, в зависимости от разрядности адреса). При использовании формы записи OUTS ассемблер определяет по типу указанного операнда, какую из трех форм этой команды (OUTSB, OUTSW или OUTSD) употребить. Применяя OUTS с операндами, также можно заменить регистр DS другим с помощью префикса замены сегмента (ES:, GS:, FS:, CS:, SS:). После выполнения команды регистр ESI (или SI) увеличивается на 1, 2 или 4 (если считывается байт, слово или двойное слово), когда флаг DF = 0, и уменьшается, когда DF = 1. Команда OUTS с префиксом REP записывает блок данных размером в ECX (или CX) байтов, слов или двойных слов в указанный порт. Все процессоры до Pentium не проверяли готовность порта принять новые данные в ходе выполнения команды REP OUTS, так что, если порт не успевал обрабатывать информацию с той скоростью, с которой ее поставляла эта команда, часть данных терялась.

2.3.9. *Управление флагами*

Команда	**Назначение**	**Процессор**
STC	Установить флаг переноса	8086

Устанавливает флаг CF в 1.

Команда	**Назначение**	**Процессор**
CLC	Сбросить флаг переноса	8086

Сбрасывает флаг CF в 0.

Команда	**Назначение**	**Процессор**
CMC	Инвертировать флаг переноса	8086

Инвертирует флаг CE

Команда	Назначение	Процессор
STD	Установить флаг направления	8086

Устанавливает флаг DF в 1, так что при последующих строковых операциях регистры DI и SI будут уменьшаться.

Команда	Назначение	Процессор
CLD	Сбросить флаг направления	8086

Сбрасывает флаг DF в 0, так что при последующих строковых операциях регистры DI и SI будут увеличиваться.

Команда	Назначение	Процессор
LAHF	Загрузить флаги состояния в AH	8086

Копирует младший байт регистра FLAGS в AH, включая флаги SF (бит 7), ZF (бит 6), AF (бит 4), PF (бит 2) и CF (бит 0). Бит 1 устанавливается в 1, биты 3 и 5 - в 0.

Команда	Назначение	Процессор
SAHF	Загрузить флаги состояния из AH	8086

Загружает флаги SF, ZF, AF, PF и CF из регистра AH значениями битов 7, 6, 4, 2 и 0 соответственно. Зарезервированные биты 1, 3 и 5 регистра флагов не изменяются.

Команда	Назначение	Процессор
PUSHF	Поместить FLAGS в стек	8086
PUSHFD	Поместить EFLAGS в стек	80386

Эти команды копируют содержимое регистра FLAGS или EFLAGS в стек (уменьшая SP или ESP на 2 или 4 соответственно). При копировании регистра EFLAGS флаги VM и RF (биты 16 и 17) не копируются, а соответствующие биты в двойном слове, помещенном в стек, обнуляются.

Команда	Назначение	Процессор
POPF	Загрузить FLAGS из стека	8086
POPFD	Загрузить EFLAGS из стека	80386

Считывает из вершины стека слово (POPF) или двойное слово (POPFD) и помещает в регистр FLAGS или EFLAGS. Эффект этих команд зависит от режима, в котором выполняется программа: в реальном и защищенном режимах с уровнем привилегий 0 модифицируются все незарезервированные флаги в EFLAGS, кроме VIP, VIF и VM. VIP и VIF обнуляются, и VM не изменяется. В защищенном режиме с уровнем привилегий, большим нуля, но меньшим или равным

IOPL, модифицируются все флаги, кроме VIP, VIF, VM и IOPL. В режиме V86 не модифицируются флаги VIF, VIP, VM, IOPL и RF.

Команда	Назначение	Процессор
CLI	Запретить прерывания	8086

Сбрасывает флаг IF в 0. После выполнения этой команды процессор игнорирует все прерывания от внешних устройств (кроме NMI). В защищенном режиме эта команда, так же как и все другие команды, модифицирующие флаг IF (POPF или IRET), выполняется, только если программе даны соответствующие привилегии (CPL ⩽ IOPL).

Команда	Назначение	Процессор
STI	Разрешить прерывания	8086

Устанавливает флаг IF в 1, отменяя тем самым действие команды CLI.

Команда	Назначение	Процессор
SALC	Установить AL в соответствии с CF	8086

Устанавливает AL в 0FFh, если флаг CF = 1, и сбрасывает в 00h, если CF = 0. Это недокументированная команда с кодом OD6h, присутствующая во всех процессорах Intel и совместимых с ними (начиная с 8086). В документации на Pentium Pro команда SALC упоминается в общем списке команд, но ее действие не описывается (оно аналогично SBB AL,AL, однако значения флагов не изменяются).

2.3.10. Загрузка сегментных регистров

Команда	Назначение	Процессор
LDS приемник,источник	Загрузить адрес, используя DS	8086
LES приемник,источник	Загрузить адрес, используя ES	8086
LFS приемник,источник	Загрузить адрес, используя FS	80386
LGS приемник,источник	Загрузить адрес, используя GS	80386
LSS приемник,источник	Загрузить адрес, используя SS	8086

Второй операнд (источник) для всех этих команд - переменная в памяти размером в 32 или 48 бит (в зависимости от разрядности операндов). Первые 16 или 32 бита из этой переменной загружаются в регистр общего назначения, указанный в качестве первого операнда, а следующие 16 бит - в соответствующий сегментный регистр (DS для LDS, ES для LES и т. д.). В защищенном режиме значение, загружаемое в сегментный регистр, всегда должно быть правильным селектором сегмента (в реальном режиме любое число может использоваться как селектор).

2.3.11. Другие команды

Команда	Назначение	Процессор
NOP	Отсутствие операции	8086

NOP - однобайтная команда (код 90h), которая не выполняет ничего, только занимает место и время. Код этой команды фактически соответствует XCHG AL,AL. Многие команды разрешается записать так, что они не будут приводить ни к каким действиям, например:

```
mov     ax, ax          ; 2 байта.
xchg    ax, ax          ; 2 байта.
lea     bx, [bx+0]      ; 3 байта (8Dh, 5Fh, 00h, но многие ассемблеры,
                        ; встретив такую команду, реально используют более
                        ; короткую lea bx,[bx] с кодом 8Dh 1Fh).
shl     eax ,0          ; 4 байта.
shrd    eax ,eax, 0     ; 5 байт.
```

Команда	Назначение	Процессор
LOCK	Префикс блокировки шины данных	8086

На все время выполнения команды, снабженной таким префиксом, будет заблокирована шина данных, и если в системе присутствует другой процессор, он не сможет обращаться к памяти, пока не закончится выполнение команды с префиксом LOCK. Команда XCHG всегда выполняется автоматически с блокировкой доступа к памяти, даже если префикс LOCK не указан. Этот префикс можно использовать только с командами ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD и XCHG.

Команда	Назначение	Процессор
UD2	Неопределенная операция	P6

Эта команда всегда вызывает ошибку «неопределенная операция» (исключение #UD). Впервые она описана как таковая для Pentium Pro, но во всех предыдущих процессорах UD2 (код OFh 0Bh) не была определена и, естественно, приводила к такой же ошибке. Команда предназначена для тестирования программного обеспечения, в частности операционных систем, которые должны уметь корректно обрабатывать такую ошибку. Название команды происходит от команды UD (код OFh 0FFh), которая была определена AMD для процессоров AMD K5.

Команда	Назначение	Процессор
CPUID	Идентификация процессора	80486

CPUID сообщает информацию о производителе, типе и модификации процессора и о наличии различных расширений. Команда CPUID поддерживается Intel, начиная с процессоров Intel 80486DX/SX/DX2 SL, UMC U5S, Cyrix M1, AMD 80486DX4. Попробуйте установить флаг ID в 1 (бит 21 в регистре EFLAGS) - если это получается, значит, команда CPUID поддерживается.

Результат работы CPUID зависит от значения регистра EAX. Если EAX = 0, CPUID возвращает в EAX максимальное значение, с которым ее можно вызывать (2 для P6, 1 для P5), а регистры EBX:ECX:EDX содержат 12-байтную строку - идентификатор производителя (см. табл. 8).

Таблица 8. Строки производителей в CPUID

Производитель	Строка в EBX:ECX:EDX
Intel	GenuineIntel
UMC	UMC UMC UMC
Cyrix	CyrixInstead
AMD	AuthenticAMD
NexGen	NexGenDriven
CentaurTechnology	CentaurHalls

Например, для процессоров Intel регистр EBX содержит **Genu** (756E6547h), ECX - **inel** (49656E69h), а EDX - **ntel** (6C65746Eh).

Если EAX = 1, CPUID возвращает в EAX информацию о версии процессора, а в EDX - сведения о поддерживаемых расширениях. Многие понятия в этом описании относятся к работе процессора в защищенном режиме и рассмотрены ниже.

Информация о версии процессора:

Биты 3-0 - модификация.

Биты 7-4 - модель.

Биты 11–8 - семейство (3 для 386, 4 для 486, 5 для Pentium, 6 для Pentium Pro).

Биты 13-12 - тип (0 - OEM, 1 - Overdrive, 2 - Dual).

Биты 31–14 — зарезервированы и равны нулю.

Поддерживаемые расширения (регистр EDX):

Бит 0: FPU - процессор содержит FPU и может выполнять весь набор команд 80387.

Бит 1: VME - процессор поддерживает усовершенствованный режим V86 (флаги VIF и VIP в EFLAGS, биты VME и PVI в CRO).

Бит 2: DE - процессор поддерживает точки останова по вводу/выводу, бит DE в CRO.

Бит 3: PSE - процессор поддерживает страницы до 4 Мб, бит PSE в CR4, модифицированные биты в элементах списков страниц (PDE) и таблиц страниц (PTE).

Бит 4: TSC - процессор поддерживает команду RDTSC и бит TSC в CR4.

Бит 5: MSR - процессор поддерживает команды RDMSR и WRMSR и машинно-специфичные регистры, совместимые с Pentium.

Бит 6: PAE - процессор поддерживает физические адреса больше 32 бит, дополнительный уровень *в* таблицах трансляции страниц, страницы по 2 Мб и бит PAE в CR4. Число битов для физических адресов зависит от модели процессора. Так, Pentium Pro поддерживает 36 бит.

Бит 6: PTE (только для Cyrix).

Бит 7: MCE - процессор поддерживает бит MCE в CR4.

Бит 8: CX8 - процессор поддерживает команду CMPXCHG8B.

Бит 9: APIC - процессор содержит встроенный контроллер прерываний (APIC), который активизирован и доступен.

Бит 9: PGE (только для AMD).

Бит 10: зарезервирован.
Бит 11: SEP - процессор поддерживает быстрые системные вызовы, команды SYSENTER и SYSEXIT (Pentium II).
Бит 12: MTRR - процессор поддерживает машинно-специфичные регистры MTRR.
Бит 13: PGE - процессор поддерживает бит PGE в CR4 и глобальные флаги в PTDE и PTE, указывающие элементы TLB, которые принадлежат сразу нескольким задачам.
Бит 14: MCA - процессор поддерживает машинно-специфичный регистр MCG_CAP.
Бит 15: CMOV - процессор поддерживает команды CMOVcc и (если бит О EDX установлен) FCMOVcc (Pentium Pro).
Бит 16: PAT - процессор поддерживает таблицу атрибутов страниц.
Биты 17-22: зарезервированы.
Бит 23: MMX - процессор поддерживает набор команд MMX.
Бит 24: FXSR - процессор поддерживает команды быстрого чтения/записи (MMX2).
Бит 25: SSE - процессор поддерживает расширения SSE (Pentium III).
Биты 31-26: зарезервированы.

Если EAX = 2, CPUID на процессорах семейства P6 возвращает в регистрах EAX, EBX, ECX и EDX информацию о кэшах и TLB. Самый младший байт EAX (регистр AL) определяет, сколько раз надо вызвать CPUID с EAX = 2, чтобы получить информацию обо всех кэшах (1 для Pentium Pro и Pentium II). Самый старший бит (бит 31) каждого регистра указывает, содержит ли этот регистр правильную информацию (бит 31 = 0) или он зарезервирован (бит 31 = 1). В первом случае регистр содержит информацию в 1-байтных дескрипторах со следующими значениями:

00h - пустой дескриптор;
01h - TLB команд, 4-килобайтные страницы, 4-сторонняя ассоциативность, 32 элемента;
02h - TLB команд, 4-мегабайтные страницы, 4-сторонняя ассоциативность, 4 элемента;
03h - TLB данных, 4-килобайтные страницы, 4-сторонняя ассоциативность, 64 элемента;
04h - TLB данных, 4-мегабайтные страницы, 4-сторонняя ассоциативность, 8 элементов;
06h - кэш команд, 8 Кб, 4-сторонняя ассоциативность, 32 байта в строке;
08h - кэш команд, 16 Кб, 4-сторонняя ассоциативность, 32 байта в строке;
0Ah - кэш данных, 8 Кб, 2-сторонняя ассоциативность, 32 байта в строке;
0Ch – кэш данных, 16 Кб, 2-сторонняя ассоциативность, 32 байта в строке;
41h – унифицированный кэш, 128 Кб, 4-сторонняя ассоциативность, 32 байта в строке;
42h - унифицированный кэш, 256 Кб, 4-сторонняя ассоциативность, 32 байта в строке;
43h - унифицированный кэш, 512 Кб, 4-сторонняя ассоциативность, 32 байта в строке;

44h - унифицированный кэш, 1 Мб, 4-сторонняя ассоциативность, 32 байта в строке.

Совместимые с Intel процессоры AMD и Cyrix поддерживают вызов «расширенных функций» CPUID со значениями EAX, в которых самый старший бит всегда установлен в 1.

EAX = 80000000h: возвращает в EAX максимальный номер расширенной функции CPUID, поддерживаемой данным процессором.

EAX = 80000001h: возвращает в EAX 051Xh для AMD K5 (X - номер модификации) или 061Xh для AMD K6. В EDX эта функция возвращает информацию о поддерживаемых расширениях (указаны только флаги, отличающиеся от CPUID с EAX = 1).

Бит 5: MSR — процессор поддерживает машинно-специфичные регистры, совместимые с K5.

Бит 10: процессор поддерживает команды SYSCALL и SYSRET.

Бит 16: процессор поддерживает команды FCMOVcc.

Бит 24: процессор поддерживает MMX с расширениями от Cyrix.

Бит 25: процессор поддерживает набор команд AMD 3D.

EAX - 80000002h, 80000003h и 80000004h - последовательный вызов CPUID с этими значениями в EAX возвращает в EAX:EBX:ECX:EDX последовательно четыре 16-байтные части строки - имени процессора. Например: AMD-K5(tm) Processor.

EAX = 80000005h - команда возвращает информацию о TLB в регистре EBX (старшее слово — TLB данных, младшее слово — TLB команд, старший байт — ассоциативность, младший байт - число элементов), о кэше данных в регистре ECX и о кэше команд в регистре EDX (биты 31–24 - размер в килобайтах, биты 23–16 - ассоциативность, биты 15-8 - число линий на тэг, биты 7-0 - число байтов на линию).

2.4. Числа с плавающей запятой

В процессорах Intel все операции с плавающей запятой выполняет специальное устройство - FPU (Floating Point Unit) - с собственными регистрами и набором команд, поставлявшееся сначала в виде сопроцессора (8087, 80287, 80387, 80487), а начиная с 80486DX - встраивающееся в основной процессор. FPU полностью соответствует стандартам IEEE 754 и IEEE 854 (с 80486).

2.4.1. Типы данных FPU

Числовой процессор может выполнять операции с семью разными типами данных, представленными в табл. 9: три целых двоичных, один целый десятичный и три с плавающей запятой.

Вещественные числа хранятся, как и все данные, в форме двоичных чисел. Двоичная запись числа с плавающей запятой аналогична десятичной, только позиции справа от запятой соответствуют не делению на 10 в соответствующей степени, а делению на 2. Переведем для примера в двоичный вид число 0,625:

Таблица 9. Типы данных FPU

Тип данных	Бит	Количество значащих цифр	Пределы
Целое слово	16	4	–32 768 ... 32 767
Короткое целое	32	9	-2×10^{9} ... 2×10^{9}
Длинное слово	64	18	-9×10^{18} ... 9×10^{18}
Упакованное десятичное	80	18	–99..99 ... +99..99 (18 цифр)
Короткое вещественное	32	7	$1{,}18\times10^{-38}$ ... $3{,}40\times10^{38}$
Длинное вещественное	64	15-16	$2{,}23\times10^{-308}$ ... $1{,}79\times10^{308}$
Расширенное вещественное	80	19	$3{,}37\times10^{-4932}$ ... $1{,}18\times10^{4932}$

0,625 - 1/2 - 0,125
1/4 больше, чем 0,125
0,125 – 1/8 = 0

Итак, 0,625 = 0,101b. При записи вещественных чисел всегда выполняют нормализацию - умножают число на такую степень двойки, чтобы перед десятичной точкой стояла единица, в нашем случае

$0{,}625 = 0{,}101b - 1{,}01b \times 2^{-1}$.

Говорят, что число имеет мантиссу 1,01 и экспоненту –1. Как можно заметить, при использовании этого алгоритма первая цифра мантиссы всегда равна 1, поэтому ее можно не писать, увеличивая тем самым точность представления числа дополнительно на 1 бит. Кроме того, значение экспоненты хранят не в виде целого со знаком, а в виде суммы с некоторым числом так, чтобы всегда было только положительное число и чтобы вещественные числа легко сопоставлялись - в большинстве случаев достаточно сравнить экспоненту. Теперь мы можем рассмотреть вещественные форматы IEEE, применяемые в процессорах Intel:

- *короткое вещественное:* бит 31 - знак мантиссы, биты 30-23 - 8-битная экспонента +127, биты 22-0 - 23-битная мантисса без первой цифры;
- *длинное вещественное:* бит 63 - знак мантиссы, биты 62-52 - 11-битная экспонента +1024, биты 51-0 - 52-битная мантисса без первой цифры;
- *расширенное вещественное:* бит 79 - знак мантиссы, биты 78-64 - 15-битная экспонента +16 383, биты 63-0 - 64-битная мантисса с первой цифрой (то есть бит 63 равен 1).

FPU выполняет все вычисления в 80-битном расширенном формате, а 32- и 64-битные числа используются для обмена данными с основным процессором и памятью.

Кроме обычных чисел формат IEEE предусматривает несколько специальных случаев, которые могут получаться в результате математических операций и над которыми также можно выполнять отдельные операции:
- *положительный ноль: все биты числа сброшены в ноль;*

а *отрицательный ноль: знаковый бит - 1, все остальные биты - нули;*

- *положительная бесконечность: знаковый бит - 0, все биты мантиссы — 0, все биты экспоненты - 1;*
- *отрицательная бесконечность: знаковый бит — 1, все биты мантиссы — 0, все биты экспоненты - 1;*
- *денормализованные числа: все биты экспоненты - 0 (используются для работы с очень маленькими числами – до 10^{-16445} для расширенной точности);*
- *неопределенность: знаковый бит - 1, первый бит мантиссы (первые два для 80-битных чисел) - 1, а остальные - 0, все биты экспоненты - 1;*
- *не-число типа SNAN (сигнальное): все биты экспоненты - 1, первый бит мантиссы - 0 (для 80-битных чисел первые два бита мантиссы - 10), а среди остальных битов есть единицы;*
- *не-число типа QNAN (тихое): все биты экспоненты - 1, первый бит мантиссы (первые два для 80-битных чисел) - 1, среди остальных битов есть единицы. Неопределенность - один из вариантов QNAN;*
- *неподдерживаемое число: все остальные ситуации.*

Остальные форматы данных FPU также допускают неопределенность — единица в старшем бите и нули в остальных для целых чисел, и старшие 16 бит - единицы для упакованных десятичных чисел.

2.4.2. Регистры FPU

FPU предоставляет восемь регистров для хранения данных и пять вспомогательных регистров.

Регистры данных (RO - R7) не адресуются по именам, как регистры основного процессора, а рассматриваются в качестве стека, вершина которого называется ST; более глубокие элементы - ST(1), ST(2) и так далее до ST(7). Если, например, в какой-то момент времени регистр R5 называется ST (см. рис. 13), то после записи в этот стек числа оно будет записано в регистр R4, который станет называться ST, R5 станет называться ST(1) и т. д.

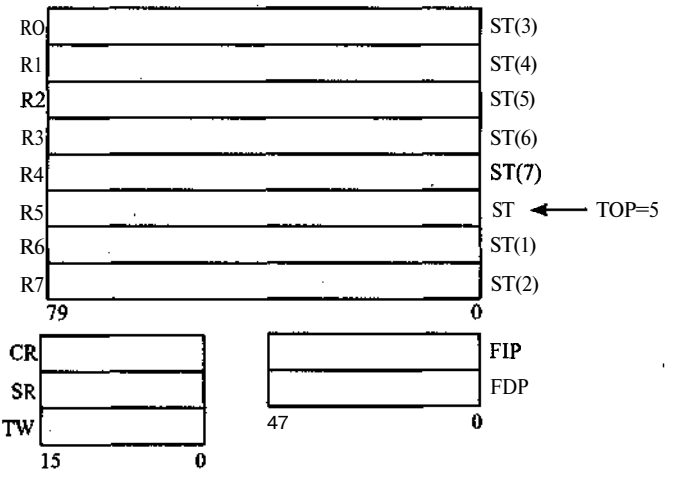

Рис. 13. Регистры FPU

Крегистрам RO - R7 нельзя обращаться напрямую, по именам, но если процессор поддерживает расширение MMX, то мантиссы, находящиеся в этих регистрах, становятся доступны, как MM0 - MM7.

Регистр состояний SR содержит слово состояния FPU:

бит 15: B - занятость FPU - этот флаг существует для совместимости с 8087, и его значение всегда совпадает с ES

бит 14: C3 - условный флаг 3

биты 13-11: TOP - число от 0 до 7, показывающее, какой из регистров данных RO - R7 в настоящий момент является вершиной стека

бит 10: C2 – условный флаг 2

бит 9: C1 - условный флаг 1

бит 8: CO - условный флаг O

бит 7: ES - общий флаг ошибки - равен 1, если произошло хотя бы одно немаскированное исключение

бит 6: SF - ошибка стека. Если C1 = 1, произошло переполнение (команда пыталась писать в непустую позицию в стеке), если C1 = 0, произошло антипереполнение (команда пыталась считать число из пустой позиции в стеке)

бит 5: PE - флаг неточного результата - результат не может быть представлен точно

бит 4: UE - флаг антипереполнения - результат слишком маленький

бит 3: OE - флаг переполнения - результат слишком большой

бит 2: ZE - флаг деления на ноль '- выполнено деление на ноль

бит 1: DE — флаг денормализованного операнда — выполнена операция над денормализованным числом

бит 0: IE - флаг недопустимой операции - произошла ошибка стека (SF = 1) или выполнена недопустимая операция

Биты CO - C3 применяются так же, как и биты состояния в основном процессоре, - их значения отражают результат выполнения предыдущей команды и используются для условных переходов; команды

```
fstsw ax
sahf
```

копируют значения битов в регистр FLAGS так, что флаг CO переходит в CF, C2 - в PF, а C3 - в ZF (флаг C2 теряется).

Биты 0-5 отражают различные ошибочные ситуации, которые могут возникать при выполнении команд FPU. Они рассмотрены в описании управляющих регистров.

Регистр управления CR:

биты 15-13: зарезервированы;

бит 12: IC - управление бесконечностью (поддерживается для совместимости с 8087 и 80287 - вне зависимости от значения этого бита $+\infty > -\infty$);

биты 11-10: RC - управление округлением;

биты 9-8: PC - управление точностью;

биты 7-6: зарезервированы;
бит 5: PM — маска неточного результата;
бит 4: UM – маска антипереполнения;
бит 3: OM - маска переполнения;
бит 2: ZM - маска деления на ноль;
бит 1: DM – маска денормализованного операнда;
бит 0: IM - маска недействительной операции.

Биты RC определяют способ округления результатов команд FPU до заданной точности (см. табл. 10).

Таблица 10. Способы округления

Значение RC	Способ округления
0	К ближайшему числу
1	К отрицательной бесконечности
2	К положительной бесконечности
3	К нулю

Биты PC определяют точность результатов команд FADD, FSUB, FSUBR, FMUL, FDIV, FDIVR и FSQRT (см. табл. И).

Таблица 11. Точность результатов

Значение PC	Точность результатов
0	Одинарная точность (32-битные числа)
1	Зарезервировано
2	Двойная точность (64-битные числа)
3	Расширенная точность (80-битные числа)

Биты 0-5 регистра CR маскируют соответствующие исключения - если маскирующий бит установлен, исключения не происходит, а результат вызвавшей его команды определяется правилами для каждого исключения специально.

Регистр тэгов TW содержит восемь пар битов, описывающих содержание каждого регистра данных: биты 15-14 описывают регистр R7, 13-12 - R6 и т. д. Если пара битов (тэгов) равна 11, соответствующий регистр пуст. 00 означает, что регистр содержит число, 01 — ноль, 10 — не-число, бесконечность, денормализованное число, неподдерживаемое число.

Регистры FIP и FDP содержат адрес последней выполненной команды (кроме FINIT, FCLEX, FLDCW, FSTCW, FSTSW, FSTSWAX, FSTENV, FLDENV, FSAVE, FRSTOR и FWAIT) и адрес ее операнда соответственно и используются в обработчиках исключений для анализа вызвавшей его команды.

2.4.3. *Исключения FPU*

При выполнении команд FPU могут возникать шесть типов особых ситуаций, называемых исключениями. При возникновении исключения соответствующий

флаг в регистре SR устанавливается в 1 и, если маска этого исключения в регистре CR не установлена, вызывается обычное прерывание INT 10h (если бит NE в регистре центрального процессора CRO установлен в 1) или IRQ13 (INT 75h), обработчик которого может прочитать регистр SR, чтобы определить тип исключения (и FIP, и FDP) и команду, которая его породила, а затем попытаться исправить ситуацию. Если бит маски наступившего исключения в регистре CR установлен в 1, по умолчанию выполняются следующие действия:

- *неточный результат*: результат округляется в соответствии с битами RC (на самом деле это исключение происходит очень часто; например: дробь 1/6 не может быть представлена десятичным вещественным числом любой точности и округляется). При этом флаг C1 показывает, в какую сторону произошло округление: 0 - вниз, 1 - вверх;
- *антипереполнение:* результат слишком мал, чтобы быть представленным обычным числом, - он преобразуется в денормализованное число;
- *переполнение:* результат преобразуется в бесконечность соответствующего знака;
- *деление на ноль*: результат преобразуется в бесконечность соответствующего знака (учитывается и знак нуля);
- *денормализованный операнд:* вычисление продолжается, как обычно;
- *недействительная операция:* результат определяется из табл. 12.

Таблица 12. Результаты операций, приводящих к исключениям

Операция	Результат
Ошибка стека	Неопределенность
Операция с неподдерживаемым числом	Неопределенность
Операция с SNAN	QNAN
Сравнение числа с NAN	CO = C2 = C3 = 1
Сложение бесконечностей с одним знаком или вычитание - с разным	Неопределенность
Умножение нуля на бесконечность	Неопределенность
Деление бесконечности на бесконечность или О/О	Неопределенность
Команды FPREM и FPREM1, если делитель - 0 или делимое - бесконечность	Неопределенность и C2 = 0
Тригонометрическая операция над бесконечностью	Неопределенность и C2 = 0
Корень или логарифм, если x < 0, log(x+1), если x < -1	Неопределенность
FBSTP, если регистр-источник пуст, содержит NAN, бесконечность или превышает 18 десятичных знаков	Десятичная неопределенность
FXCH, если один из операндов пуст	Неопределенность

2.4.4. *Команды пересылки данных FPU*

Команда	Назначение	Процессор
FLD источник	Загрузить вещественное число в стек	8087

Команда помещает содержимое источника (32-, 64- или 80-битная переменная или ST(n)) в стек и уменьшает TOP на 1. Команда FLD ST(0) делает копию вершины стека.

Команда	Назначение	Процессор
FST приемник	Скопировать вещественное число из стека	8087
FSTP приемник	Считать вещественное число из стека	8087

Копирует ST(0) в приемник (32- или 64-битную переменную или пустой ST(n) в случае FST; 32-, 64- или 80-битную переменную или пустой ST(n) в случае FSTP). FSTP после этого выталкивает число из стека (помечает ST(0) как пустой и увеличивает TOP на один).

Команда	Назначение	Процессор
FILD источник	Загрузитьцелое число в стек	8087

Преобразовывает целое число со знаком из источника (16-, 32- или 64-битная переменная) в вещественный формат, помещает в вершину стека и уменьшает TOP на 1.

Команда	Назначение	Процессор
FIST приемник	Скопировать целое число из стека	8087
FISTP приемник	Считать целое число из стека	8087

Преобразовывает число из вершины стека в целое со знаком и записывает его в приемник (16- или 32-битная переменная для FIST; 16-, 32- или 64-битная переменная для FISTP). FISTP после этого выталкивает число из стека (помечает ST(0) как пустой и увеличивает TOP на один). Попытка записи слишком большого числа, бесконечности или не-числа приводит к исключению «недопустимая операция» (и записи целой неопределенности, если IM = 1).

Команда	Назначение	Процессор
FBLD источник	Загрузить десятичное число в стек	8087

Преобразовывает BCD число из источника (80-битная переменная в памяти), помещает в вершину стека и уменьшает TOP на 1.

Команда	Назначение	Процессор
FBSTP приемник	Считать десятичное число из стека	8087

Преобразовывает число из вершины стека в 80-битное упакованное десятичное, записывает его в приемник (80-битная переменная) и выталкивает это число из стека (помечает ST(0) как пустой и увеличивает TOP на один). Попытка записи слишком большого числа, бесконечности или не-числа приводит к исключению «недопустимая операция» (и записи десятичной неопределенности, если IM = 1).

Команда	Назначение	Процессор
FXCH источник	Обменять местами два регистра стека	8087

Обмен местами содержимого регистра ST(0) и источника (регистр ST(n)). Если операнд не указан, обменивается содержимое ST(0) и ST(1).

Команда	Назначение	Процессор
FCMOVcc приемник,источник	Условная пересылка данных	P6

Это набор команд, каждая из которых копирует содержимое источника (регистр ST(n)) в приемник (только ST(0)), если выполняется необходимое условие. Реально каждое условие соответствует тем или иным значениям флагов регистра FLAGS, но после команд

```
fcom    (или другие команды сравнения)
fstsw   ax
sahf
```

в регистр FLAGS загружаются флаги CO, C1 и C3, и последующая команда из набора FCMOVcc приобретает смысл обработки результата предыдущего сравнения (см. табл. 13).

Таблица 13. Команды FCMOVcc

Команда	Значения флагов	Действие после FCOM
FCMOVE	ZF = 1	Если равно
FCMOVNE	ZF = 0	Если не равно
FCMOVB	CF = 1	Если меньше
FCMOVBE	CF = 1 и ZF = 1	Если меньше или равно
FCMOVNB	CF = 0	Если не меньше
FCMOVNBE	CF = 0 и ZF = 0	Если не меньше или равно
FCMOVU	PF = 1	Если несравнимы
FCMOVNU	PF = 0	Если сравнимы

2.4.5. *Базовая арифметика FPU*

Команда	Назначение	Процессор
FADD приемник,источник	Сложение вещественных чисел	8087
FADDP приемник,источник	Сложение с выталкиванием из стека	8087
FIADD источник	Сложение целых чисел	8087

Команда выполняет сложение источника и приемника и помещает результат в приемник. Команда FADDP после этого выталкивает ST(0) из стека (помечает ST(0) как пустой и увеличивает TOP на один). Команды сложения могут принимать следующие формы:

- *FADD источник,* когда источником является 32- или 64-битная переменная, а приемником - ST(0);
- *FADD ST(0),ST(n), FADD ST(n),ST(0), FADDP ST(n),ST(0),* когда источник и приемник заданы в виде регистров FPU;
- *FADD без* операндов эквивалентна FADD ST(0),ST(1); *FADDP* без операндов эквивалентна FADDP ST(i),ST(0);
- *FIADDисточник,* когда источником является 16- или 32-битная переменная, содержащая целое число, а приемником - ST(0).

Команда	Назначение	Процессор
FSUB приемник,источник	Вычитание вещественных чисел	8087
FSUBP приемник,источник	Вычитание с выталкиванием из стека	8087
FISUB источник	Вычитание целых чисел	8087

Выполняет вычитание источника из приемника и сохраняет результат в приемнике. Команда FSUBP после этого выталкивает ST(0) из стека (помечает ST(0) как пустой и увеличивает TOP на один). Команды вычитания могут принимать следующие формы:

- *FSUB источник,* когда источником является 32- или 64-битная переменная, содержащая вещественное число, а приемником - ST(0);
- *FSUB ST(0),ST(n), FSUB ST(n),ST(0), FSUBP ST(n),ST(0),* когда источник и приемник заданы явно в виде регистров FPU;
- *FSUB* без операндов эквивалентна FSUB ST(0),ST(1); *FSUBP без* операндов эквивалентна FSUBP ST(1),ST(0);
- *О FISUB источник,* когда источником является 16- или 32-битная переменная, содержащая целое число, а приемником - ST(0).

Если один из операндов - бесконечность, то результат - бесконечность соответствующего знака. Если оба операнда — бесконечности одного знака, результат не определен (происходит исключение «недопустимая операция»).

Команда	Назначение	Процессор
FSUBR приемник,источник	Обратное вычитание вещественных чисел	8087
FSUBRP приемник,источник	Обратное вычитание с выталкиванием	8087
FISUBR источник	Обратное вычитание целых чисел	8087

Эти команды эквивалентны FSUB/FSUBP/FISUB, но при этом они выполняют вычитание приемника из источника, а не источника из приемника.

Команда	Назначение	Процессор
FMUL приемник,источник	Умножение вещественных чисел	8087
FMULP приемник,источник	Умножение с выталкиванием из стека	8087
FIMUL источник	Умножение целых чисел	8087

Выполняет умножение источника и приемника и помещает результат в приемник. Команда FMULP после этого выталкивает ST(0) из стека (помечает ST(0) как пустой и увеличивает TOP на один). Так же как и остальные команды базовой арифметики, команды умножения могут принимать следующие формы:

- *FMUL источник,* когда источником является 32- или 64-битная переменная, а приемником - ST(0);
- *FMUL ST(0),ST(n),FMUL ST(n),ST(0),FMULP ST(n),ST(0),* когда источник и приемник заданы явно в виде регистров FPU;
- *FMUL* без операндов эквивалентна FMUL ST(0),ST(1); *FMULP* без операндов эквивалентна FMULP ST(1),ST(0);
- *FIMULисточник,* когда источником является 16- или 32-битная переменная, содержащая целое число, а приемником - ST(0).

Команда	Назначение	Процессор
FDIV приемник,источник	Деление вещественных чисел	8087
FDIVP приемник,источник	Деление с выталкиванием из стека	8087
FIDIV источник	Деление целых чисел	8087

Выполняет деление приемника на источник и сохраняет результат в приемнике. Команда FDIVP после этого выталкивает ST(0) из стека (помечает ST(0) как пустой и увеличивает TOP на один). Команды вычитания могут принимать следующие формы:

- *FDIV источник,* когда источником является 32- или 64-битная переменная, содержащая вещественное число, а приемником - ST(0);
- *FDIV ST(0),ST(n)FDIV ST(n),ST(0),FDIVP ST(n),ST(0),*когда источник и приемник заданы явно в виде регистров FPU;
- *FDIV без* операндов эквивалентна FDIV ST(0),ST(1); *FDIVP без* операндов эквивалентна FDIVP ST(1),ST(0);
- *O FIDIV источник,* когда источником является 16- или 32-битная переменная, содержащая целое число, а приемником - ST(0).

При делении бесконечности на ноль (так же как и на любое число) результат - бесконечность, при делении нуля на бесконечность (так же как и на любое число) результат - ноль. При делении на ноль нормального числа происходит исключение деления на ноль, а если флаг ZM = 1, в качестве результата записывается бесконечность соответствующего знака.

Команда	Назначение	Процессор
FDIVR приемник,источник	Обратное деление вещественных чисел	8087
FDIVRP приемник,источник	Обратное деление с выталкиванием	8087
FIDIVR источник	Обратное деление целых чисел	8087

Эти команды эквивалентны FDIV/FDIVP/FIDIV, но при этом они выполняют деление источника на приемник, а не приемника на источник.

Команда	Назначение	Процессор
FPREM	Найти частичный остаток от деления	8087
FPREM1	Найти частичный остаток в стандарте IEEE	80387

Эти команды выполняют деление ST(0) на ST(1) и помещают остаток от деления в ST(0). Деление осуществляется при помощи последовательных вычитаний ST(1) из ST(0), но за один раз выполняется не более 64 таких вычитаний. Если ST(0) не стал меньше ST(1) за это время, говорят, что в ST(0) находится частичный остаток от деления. Если был получен точный остаток, флаг C2 сбрасывается в 0, если частичный - устанавливается в 1, так что можно повторять эту команду до обнуления C2. Если вычисление привело к точному остатку, три младших бита частного (то есть числа потребовавшихся вычитаний) сохраняются в C0, C3, C1 (биты 2, 1, 0 соответственно). Например, используя FPREM1, можно уменьшить аргумент тангенса, вычислив его остаток от деления на я/4, тогда потребуются младшие три бита частного, чтобы определить, не поменялся ли при этой операции знак тангенса.

Различие между FPREM и FPREM1 заключается в разном определении значения частного. Сначала эти команды выполняют вещественное деление ST(0) на ST(1), округляют результат (FPREM1 - к ближайшему целому, FPREM - к нулю), а затем, если частное меньше 64, вычисляют точный остаток, а если больше - частичный.

Команда	Назначение	Процессор
FABS	Найти абсолютное значение	8087

Если ST(0) был отрицательным числом - переводит его в положительное.

Команда	Назначение	Процессор
FCHS	Изменить знак	8087

Изменяет знак ST(0), превращая положительное число в отрицательное, и наоборот.

Команда	Назначение	Процессор
FRNDINT	Округлить до целого	8087

Округляет значение ST(0) до целого числа в соответствии с режимом округления, заданным битами RC.

Команда	Назначение	Процессор
FSCALE	Масштабировать по степеням двойки	8087

Умножает ST(0) на два в степени ST(1) и записывает результат в ST(0). Значение ST(1) предварительно округляется в сторону нуля до целого числа. Эта команда выполняет действие, обратное FXTRACT.

Команда	Назначение	Процессор
FXTRACT	Извлечь экспоненту и мантиссу	8087

Разделяет число в ST(0) на мантиссу и экспоненту, сохраняет экспоненту в ST(0) и помещает мантиссу в стек, так что после этого TOP уменьшается на 1, мантисса оказывается в ST(0), а экспонента - в ST(1).

Команда	Назначение	Процессор
FSQRT	Извлечь квадратный корень	8087

Вычисляет квадратный корень из ST(0) и сохраняет результат в ST(0).

2.4.6. Команды сравнения FPU

Команда	Назначение	Процессор
FCOM источник	Сравнить вещественные числа	8087
FCOMP источник	Сравнить и вытолкнуть из стека	8087
FCOMPP	Сравнить и вытолкнуть из стека два числа	8087

Команды выполняют сравнение содержимого регистра ST(0) с источником (32- или 64-битная переменная или регистр ST(n), если операнд не указан - ST(1)) и устанавливают флаги CO, C2 и C3 в соответствии с табл. 14.

Таблица 14. Флаги сравнения FPU

Условие	C3	C2	C0
ST(0) > источник	0	0	0
ST(0) < источник	0	0	1
ST(0) = источник	1	0	0
Несравнимы	1	1	1

Если один из операндов - не-число или неподдерживаемое число, происходит исключение «недопустимая операция», а если оно замаскировано (флаг IM = 1), все три флага устанавливаются в 1. После команд сравнения посредством FSTSW и SAHF можно перевести флаги C3, C2 и CO в ZF, PF и CF соответственно, затем все условные команды (Jcc, CMOVcc, FCMOVcc, SETcc) используют результат сравнения, как после команды CMP

Команда FCOMP после выполнения сравнения выталкивает из стека содержимое ST(0) (помечает его как пустой и увеличивает TOP на 1), а команда FCOMPP выталкивает из стека и ST(0), и ST(1).

Команда	Назначение	Процессор
FUCOM источник	Сравнить вещественные числа без учета порядков	80387
FUCOMP источник	Сравнить без учета порядков и вытолкнуть из стека	80387
FUCOMPP	Сравнить без учета порядков и вытолкнуть из стека два числа	80387

Эти команды аналогичны FCOM/FCOMP/FCOMPP во всем, но в роли источника могут выступать только регистры ST(n), и если один из операндов - QNAN («тихое» не-число), флаги C3, C2, CO устанавливаются в единицы, однако исключение «недопустимая операция» не вызывается. Если один из операндов - SNAN или неподдерживаемое число, эти команды ведут себя так же, как и обычное сравнение.

Команда	Назначение	Процессор
FICOM источник	Сравнить целые числа	8087
FICOMP источник	Сравнить целые и вытолкнуть из стека	8087

Эти команды сравнивают содержимое регистра ST(0) и источника (16- или 32-битная переменная), причем считается, что источник содержит целое число. В остальном действие FICOM/FICOMP полностью эквивалентно FCOM/FCOMP.

Команда	Назначение	Процессор
FCOMI источник	Сравнить и установить EFLAGS	P6
FCOMIP источник	Сравнить, установить EFLAGS и вытолкнуть	P6
FUCOMI источник	Сравнить без учета порядков и установить EFLAGS	P6
FUCOMIP источник	Сравнить без учета порядков, установить EFLAGS и вытолкнуть из стека	P6

Выполняет сравнение регистра ST(0) и источника (регистр ST(n)) и устанавливает флаги регистра EFLAGS соответственно табл. 15.

Таблица 15. Флаги после команд FUCOM

Условие	ZF	PF	CF
ST(0) > источник	0	0	0
ST(0) < источник	0	0	1
ST(0) = источник	1	0	0
Несравнимы	1	1	1

Эти команды эквивалентны командам FCOM/FCOMP/FUCOM/FUCOMP, вслед за которыми исполняются FSMSW AX и SAHF, но они не изменяют содержимого регистра AX и выполняются быстрее.

Команда	Назначение	Процессор
FTST	Проверить, не содержит ли SP(0) ноль	8087

Сравнивает содержимое ST(0) с нулем и выставляет флаги C3, C2 и CO аналогично другим командам сравнения.

Команда	Назначение	Процессор
FXAM	Проанализировать содержимое ST(0)	8087

Устанавливает флаги C3, C2 и CO в зависимости от типа числа, находящегося в ST(0), в соответствии с правилами, приведенными в табл. 16.

Таблица 16. Результаты действия команды FXAM

Тип числа	C3	C2	C0
Неподдерживаемое	0	0	0
Не-число	0	0	1
Нормальное конечное число	0	1	0
Бесконечность	0	1	1
Ноль	1	0	0
Регистр пуст	1	0	1
Денормализованное число	1	1	0

Флаг C1 устанавливается равным знаку числа в ST(0) независимо от типа числа (на самом деле он устанавливается, даже если регистр помечен как пустой).

2.4.7. *Трансцендентные операции FPU*

Команда	Назначение	Процессор
FSIN	Синус	80387

Вычисляет синус числа, находящегося в ST(0), и сохраняет результат в этом же регистре. Операнд считается заданным в радианах и не может быть больше 2^{63} или меньше -2^{63} (можно воспользоваться FPREM с делителем 2л, если операнд слишком велик). Если операнд выходит за эти пределы, флаг C2 устанавливается в 1 и значение ST(0) не изменяется.

Команда	Назначение	Процессор
FCOS	Косинус	80387

Вычисляет косинус числа, находящегося в ST(0), и сохраняет результат в этом же регистре. Операнд считается заданным в радианах и не может быть больше 2^{63} или меньше -2^{63} (так же, как и в случае синуса, можно воспользоваться FPREM с делителем 2я, если операнд слишком велик). Если операнд выходит за эти пределы, флаг C2 устанавливается в 1 и значение ST(0) не изменяется.

Команда	Назначение	Процессор
FSINCOS	Синус и косинус	80387

Вычисляет синус и косинус числа, находящегося в ST(0), помещает синус в ST(0), а затем косинус в стек (так что синус оказывается в ST(1), косинус - в ST(0), и TOP уменьшается на 1). Операнд считается заданным в радианах и не может быть больше 2^{63} или меньше -2^{63}. Если операнд выходит за эти пределы, флаг C2 устанавливается в 1 и значение ST(0) и стек не изменяются.

Команда	Назначение	Процессор
FPTAN	Тангенс	8087

Вычисляет тангенс числа, находящегося в регистре ST(0), заменяет его на вычисленное значение и затем помещает 1 в стек, так что результат оказывается в ST(1), ST(0) содержит 1, а TOP уменьшается на единицу. Как и для остальных тригонометрических команд, операнд считается заданным в радианах и не может быть больше 2^{63} или меньше -2^{63}. Если операнд выходит за эти пределы, флаг C2 устанавливается в 1 и значение ST(0) и стек не изменяются. Единица помещается в стек для того, чтобы можно было получить котангенс вызовом команды FDIVR сразу после FPTAN.

Команда	Назначение	Процессор
FPATAN	Арктангенс	8087

Вычисляет арктангенс числа, получаемого при делении ST(1) на ST(0), сохраняет результат в ST(1) и выталкивает ST(0) из стека (помечает ST(0) как пустой и увеличивает TOP на 1). Результат всегда имеет тот же знак, что и ST(1), и меньше π по абсолютной величине. Смысл этой операции в том, что FPATAN вычисляет угол между осью абсцисс и линией, проведенной из центра координат в точку ST(1), ST(0).

FPATAN может выполняться над любыми операндами (кроме не-чисел), давая результаты для различных нулей и бесконечностей, определенные в соответствии со стандартом IEEE (как показано в табл. 17).

Таблица 17. Результаты работы команды FPATAN[1]

ST(1) \ ST(0)	−∞	-F	-0	+0	+F	+∞
−∞	**−3π/4**	-я/2	-я/2	-я/2	-я/2	-я/4
-F	-я	от -я до -я/2	-я/2	-я/2	от -я/2 до -0	**-0**
-0	-я	**−π**	-я	**-0**	**-0**	**-0**
+0	+я	**+π**	**+π**	+0	+0	+0
+F	**+π**	от +я до +я/2	+я/2	+я/2	от +я/2 до +0	+0
+∞	+3я/4	+я/2	+я/2	+я/2	+я/2	+я/4

Команда	Назначение	Процессор
F2XM1	Вычисление 2x-1	8087

Возводит 2 в степень, равную ST(0), и вычитает 1. Результат сохраняется в ST(0). Значение ST(0) должно лежать в пределах от -1 до +1, иначе результат не определен.

[1] F в этой таблице - конечное вещественное число.

Команда	Назначение	Процессор
FYL2X	Вычисление y x $\log_2(x)$	8087

Вычисляет ST(1) x $\log_2(ST(0))$, помещает результат в ST(1) и выталкивает ST(0) из стека, так что после этой операции результат оказывается в ST(0). Первоначальное значение ST(0) должно быть неотрицательным. Если регистр ST(0) содержал ноль, результат (если ZM = 1) будет равен бесконечности со знаком, обратным ST(1).

Команда	Назначение	Процессор
FYL2XP1	Вычисление y x $\log_2(x+1)$	8087

Вычисляет ST(1) x $\log_2(ST(0) + 1)$, помещает результат в ST(1) и выталкивает ST(0) из стека, так что после этой операции результат оказывается в ST(0). Первоначальное значение ST(0) должно быть в пределах от $-(1 - \sqrt{2}/2)$ до $(1 + \sqrt{2}/2)$, в противном случае результат не определен. Команда FYL2XP1 дает большую точность для ST(0), близких к нулю, чем FYL2X для суммы того же ST(0) и 1.

2.4.8. Константы FPU

Команда	Назначение	Процессор
FLD1	Поместить в стек 1,0	8087
FLDZ	Поместить в стек +0,0	8087
FLDPI	Поместить в стек число я	8087
FLDL2E	Поместить в стек $\log_2(e)$	8087
FLDL2T	Поместить в стек $\log_2(10)$	8087
FLDLN2	Поместить в стек ln(2)	8087
FLDLG2	Поместить в стек lg(2)	8087

Все эти команды помещают в стек (то есть уменьшают TOP на один и помещают в ST(0)) соответствующую часто используемую константу. Начиная с сопроцессора 80387, все константы хранятся в более точном формате, чем 80-битный формат, используемый в регистрах данных, и при загрузке в стек происходит округление в соответствии с полем RC.

2.4.9. Команды управления FPU

Команда	Назначение	Процессор
FINCSTP	Увеличить указатель вершины стека	8087

Поле TOP регистра состояния FPU увеличивается на 1. Если TOP было равно семи, оно обнуляется. Эта команда не эквивалентна выталкиванию ST(0) из стека, потому что регистр данных, который назывался ST(0) и стал ST(7), не помечается как пустой.

Команда	Назначение	Процессор
FDECSTP	Уменьшить указатель вершины стека	8087

Поле TOP регистра состояния FPU уменьшается на 1. Если TOP было равно нулю, оно устанавливается в 7. Содержимое регистров данных и TW не изменяется.

Команда	Назначение	Процессор
FFREE операнд	Освободить регистр данных	8087

Команда отмечает в регистре TW, что операнд (регистр данных ST(n)) пустой. Содержимое регистра и TOP не изменяется.

Команда	Назначение	Процессор
FINIT	Инициализировать FPU	8087
FNINIT	Инициализировать FPU без ожидания	8087

Команды FINIT и FNINIT восстанавливают значения по умолчанию в регистрах CR, SR, TW, а начиная с 80387 - FIP и FDP. Управляющий регистр инициализируется значением 037Fh (округление к ближайшему, 64-битная точность, все исключения замаскированы). Регистр состояния обнуляется (TOP = 0, флаги исключений не установлены). Регистры данных никак не изменяются, но все они помечаются пустыми в регистре TW. Регистры FIP и FDP обнуляются. Команда FINIT, в отличие от FNINIT, проверяет наличие произошедших и необработанных исключений и обрабатывает их до инициализации. Команда FINIT полностью эквивалентна (и на самом деле является) WAIT FNINIT.

Команда	Назначение	Процессор
FCLEX	Обнулить флаги исключений	8087
FNCLEX	Обнулить флаги исключений без ожидания	8087

Команды обнуляют флаги исключений (PE, UE, OE, ZE, DE, IE), а также флаги ES, SF и B в регистре состояния FPU. Команда FCLEX, в отличие от FNCLEX, проверяет наличие произошедших и необработанных исключений и обрабатывает их до выполнения. Команда FCLEX полностью эквивалентна (и на самом деле является) WAIT FNCLEX.

Команда	Назначение	Процессор
FSTCW приемник	Сохранить регистр CR	8087
FNSTCW приемник	Сохранить регистр CR без ожидания	8087

Команды копируют содержимое CR в приемник (16-битная переменная). Команда FSTCW, в отличие от FNSTCW, проверяет наличие произошедших и необработанных исключений и обрабатывает их до выполнения. Команда FSTCW полностью эквивалентна (и на самом деле является) WAIT FNSTCW.

Команда	Назначение	Процессор
FLDCW источник	Загрузить регистр CR	8087

Копирует содержимое источника (16-битная переменная) в регистр CR. Если один или несколько флагов исключений установлены в регистре SR и замаскированы в CR, а команда FLDCW эти маски удалила, исключения будут обработаны перед началом выполнения следующей команды FPU (кроме команд без ожидания). Чтобы этого не происходило, обычно перед FLDCW выполняют команду FCLEX.

Команда	Назначение	Процессор
FSTENV приемник	Сохранить вспомогательные регистры	8087
FNSTENV приемник	Сохранить вспомогательные регистры без ожидания	8087

Сохраняет все вспомогательные регистры FPU в приемник (14 или 28 байт в памяти, в зависимости от разрядности операндов) и маскирует все исключения, а также сохраняет содержимое регистров CR, SR, TW, FIP, FDP и последнюю команду в формате, зависящем от текущей разрядности операндов и адресов (7 двойных слов для 32-битных операндов и 7 слов для 16-битных операндов). Первое слово (или младшая половина первого двойного слова в 32-битном случае) всегда содержит CR, второе слово - SR, третье слово - TW, четвертое - FIP. Использование последних трех слов варьируется в зависимости от текущей разрядности адресации и операндов.

- 32-битные операнды и 16-битная адресация:
  двойное слово 5: биты 10-0 старшего слова - код последней команды, младшее слово - селектор для FIP;
  двойное слово 6: FDP (32-битный);
  двойное слово 7: младшее слово содержит селектор для FDP;
- 32-битные операнды и 16-битная адресация:
  двойное слово 5: биты 31-16 - FIP, биты 10-0 - код последней команды;
  двойное слово 6: биты 15-0 - FDP;
  двойное слово 7: биты 31–16 - FDP;
- 16-битные операнды и 32-битная адресация:
  слово 5: селектор для FIP;
  слово 6: FDP;
  слово 7: селектор для FDP;
- 16-битные операнды и 16-битная адресация:
  слово 5: биты 15-12 - биты 19-16 20-битного FIP, биты 10-0 - код последней команды;
  слово 6: FDP;
  слово 7: биты 15-12 - биты 19-16 20-битного FDP.

Из кода последней выполненной FPU-команды сохраняются первые два байта без префиксов и без первых пяти бит, которые одинаковы для всех команд! FPU, то есть всего 11 бит. Команда FSTENV, в отличие от FNSTENV, проверяет наличие

произошедших и необработанных исключений и обрабатывает их до выполнения. Команда FSTENV полностью эквивалентна (и на самом деле является) WAIT FNSTENV.

Команда	Назначение	Процессор
FLDENV источник	Загрузить вспомогательные регистры	8087

Команда загружает все вспомогательные регистры FPU (регистры CR, SR, TW, FIP, FDP) из источника (область памяти в 14 или 28 байт, в зависимости от разрядности операндов), сохраненные ранее командой FSTENV/FNSTENV. Если в загружаемом SW установлены несколько (или один) флагов исключений, которые одновременно не замаскированы флагами CR, то эти исключения будут выполнены перед следующей командой FPU (кроме команд без ожидания).

Команда	Назначение	Процессор
FSAVE приемник	Сохранить состояние FPU	8087
FNSAVE приемник	Сохранить состояние FPU без ожидания	8087

Сохраняет состояние FPU (регистры данных и вспомогательные регистры) в приемник (область памяти размером 94 или 108 байт, в зависимости от разрядности операндов) и инициализирует FPU аналогично командам FINIT/FNINIT. Команда FSAVE, в отличие от FNSAVE, проверяет наличие произошедших и необработанных исключений и обрабатывает их до выполнения. Она полностью эквивалентна (и на самом деле является) WAIT FNSAVE. Эта команда обычно используется операционной системой при переключении задач или программами, которые должны передавать вызываемым процедурам чистый FPU.

Команда	Назначение	Процессор
FXSAVE приемник	Быстрое сохранение состояния FPU	PII

Команда FXSAVE сохраняет текущее состояние FPU, включая все регистры, в приемник (512-байтную область памяти с адресом, кратным 16), не проверяя на необработанные исключения, аналогично команде FNSAVE. Кроме того, в отличие от FSAVE/FNSAVE, эта команда не переинициализирует FPU после сохранения состояния. Она несовместима с FSAVE/FRSTOR.

Команда	Назначение	Процессор
FRSTOR источник	Восстановить состояние FPU	8087

Загружает состояние FPU (вспомогательные регистры и регистры данных) из источника (область в памяти размером в 94 или 108 байт, в зависимости от разрядности операндов).

Команда	Назначение	Процессор
FXRSTOR источник	Быстрое восстановление состояния FPU	PII

Команда FXRSTOR восстанавливает текущее состояние FPU, включая все регистры, из источника (512-байтной области памяти с адресом, кратным 16), который был заполнен командой FXSAVE.

Команда	Назначение	Процессор
FSTSW приемник	Сохранить регистр SR	80287
FNSTSW приемник	Сохранить регистр SR без ожидания	80287

Сохраняет текущее значение регистра SR в приемник (регистр AX или 16-битная переменная). Команда FSTSW AX обычно используется после команд сравнения и FPREM/FPREM1/FXAM, чтобы выполнять условные переходы.

Команда	Назначение	Процессор
WAIT **FWAIT**	Ожидание готовности FPU	8087

Процессор проверяет, присутствуют ли необработанные и незамаскированные исключения FPU, и обрабатывает их. Эту команду можно указывать в критических ситуациях после команд FPU, чтобы убедиться, что возможные исключения будут обработаны. WAIT и FWAIT - разные названия для одной и той же команды.

Команда	Назначение	Процессор
FNOP	Отсутствие операции	8087

Эта команда занимает место и время, но не выполняет никакого действия. Устаревшие команды FPU – FENI (разрешить исключения, 8087), FDISI (запретить исключения, 8087) и FSETPM (80287) выполняются как FNOP всеми более старшими процессорами.

2.5. Расширение IA MMX

Начиная с модификации процессора Pentium P54C, все процессоры Intel содержат расширение MMX, предназначенное для увеличения эффективности программ, работающих с большими потоками данных (обработка изображений, звука, видео, синтез), то есть для всех тех случаев, когда нужно выполнить несложные операции над массивами однотипных чисел. MMX предоставляет несколько новых типов данных, регистров и команд, позволяющих осуществлять арифметические и логические операции над несколькими числами одновременно.

2.5.1. Регистры MMX

Расширение MMX включает в себя восемь 64-битных регистров общего пользования MM0 - MM7, показанных на рис. 14.

Физически никаких новых регистров с введением MMX не появилось, MM0 - MM7 - это в точности мантиссы восьми регистров FPU, от R0 до R7. При записи числа в регистр MMX оно оказывается в битах 63-0 соответствующего регистра

FPU, а экспонента (биты 78-64) и ее знаковый бит (бит 79) заполняются единицами. Запись числа в регистр FPU также приводит к изменению соответствующего регистра MMX. Любая команда MMX, кроме EMMS, приводит к тому, что поле TOP регистра SR и весь регистр TW в FPU обнуляются. Команда EMMS заполняет регистр TW единицами. Таким образом, нельзя одновременно пользоваться командами для работы с числами с плавающей запятой и командами MMX, а если это необходимо - следует применять команды FSAVE/FRSTOR каждый раз перед переходом от FPU к MMX и обратно (эти команды сохраняют состояние регистров MMX точно так же, как и FPU).

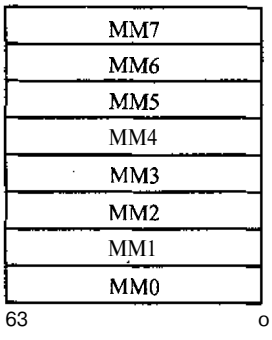

Рис. 14. Регистры MMX

2.5.2. *Типы данных MMX*

MMX использует четыре новых типа данных:

- учетверенное слово — простое 64-битное число;
- упакованные двойные слова - два 32-битных двойных слова, упакованные в 64-битный тип данных. Двойное слово 1 занимает биты 63–32, и двойное слово 0 - биты 31-0;
- упакованные слова - четыре 16-битных слова, упакованные в 64-битный тип данных. Слово 3 занимает биты 63-48, слово 0 - биты 15-0;
- упакованные байты - восемь байт, упакованных в 64-битный тип данных. Байт 7 занимает биты 63-56, байт 0 - биты 7-0.

Команды MMX перемещают упакованные данные в память или в обычные регистры как целое, но выполняют арифметические и логические операции над каждым элементом по отдельности.

Арифметические операции в MMX могут использовать специальный способ обработки переполнений и антипереполнений - *насыщение*. Если результат операции больше, чем максимальное значение для его типа данных (+127 для байта со знаком), то результат подразумевают равным этому максимальному значению. Если он меньше минимального значения - соответственно его считают равным минимально допустимому значению. Например, при операциях с цветом насыщение позволяет ему превращаться в чисто белый при переполнении и в чисто черный при антипереполнении, в то время как обычная арифметика привела бы к нежелательной инверсии цвета.

2.5.3. *Команды пересылки данных MMX*

Команда	Назначение	Процессор
MOVD приемник,источник	Пересылка двойных слов	MMX

Команда копирует двойное слово из источника (регистр MMX, обычный регистр или переменная) в приемник (регистр MMX, обычный регистр или переменная, но один из операндов обязательно должен быть регистром MMX). Если

приемник - регистр MMX, двойное слово записывается в его младшую половину (биты 31-0), а старшая заполняется нулями. Если источник - регистр MMX, в приемник записывается младшее двойное слово этого регистра.

Команда	Назначение	Процессор
MOVQ приемник,источник	Пересылка учетверенных слов	MMX

Копирует учетверенное слово (64 бита) из источника (регистр MMX или переменная) в приемник (регистр MMX или переменная, оба операнда не могут быть переменными).

2.5.4. *Команды преобразования типов MMX*

Команда	Назначение	Процессор
PACKSSWB приемник, источник	Упаковка со знаковым насыщением	MMX
PACKSSDW приемник, источник		

Команды упаковывают и насыщает слова со знаком в байты (PACKSSWB) или двойные слова со знаком в слова (PACKSSDW). Команда PACKSSWB копирует четыре слова (со знаком), находящиеся в приемнике (регистр MMX), в 4 младших байта (со знаком) приемника и копирует четыре слова (со знаком) из источника (регистр MMX или переменная) в старшие четыре байта (со знаком) приемника. Если значение какого-нибудь слова больше +127 (7Fh) или меньше -128 (80h), в байты помещаются числа +127 и -128 соответственно. Команда PACKSSDW аналогично копирует два двойных слова из приемника в два младших слова приемника и два двойных слова из источника в два старших слова приемника. Если значение какого-нибудь двойного слова больше +32 767 (7FFFh) или меньше -32 768 (8000h), в слова помещаются числа +32 767 и -32 768 соответственно.

Команда	Назначение	Процессор
PACKUSWB приемник,источник	Упаковка с беззнаковым насыщением	MMX

Копирует четыре слова (со знаком), находящиеся в приемнике (регистр MMX), в 4 младших байта (без знака) приемника и копирует четыре слора (со знаком) из источника (регистр MMX или переменная) в старшие четыре байта (без знака) приемника. Если значение какого-нибудь слова больше 255 (0FFh) или меньше О (00h), в байты помещаются числа 255 и 0 соответственно.

Команда	Назначение	Процессор
PUNPCKHBW приемник,источник	Распаковка и объединение старших элементов	MMX
PUNPCKHWD приемник,источник	Распаковка и объединение старших элементов	MMX
PUNPCKHDQ приемник,источник	Распаковка и объединение старших элементов	MMX

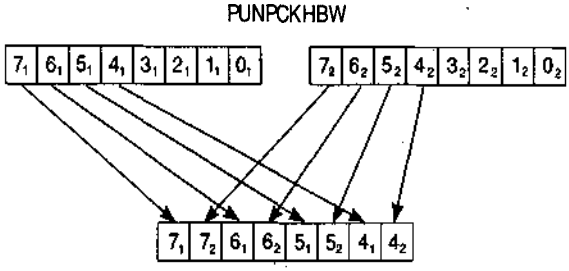

Рис. 15. Действие команды PUNPCKHBW

Команды распаковывают старшие элементы источника (регистр MMX или переменная) и приемника (регистр MMX) и записывают их в приемник через один (см. рис. 15).

Команда PUNPCKHBW объединяет по 4 старших байта источника и приемника, команда PUNPCKHWD - по 2 старших слова, а команда PUNPCKHDQ копирует в приемник по одному старшему двойному слову из источника и приемника.

Если источник содержит нули, эти команды фактически переводят старшую половину приемника из одного формата данных в другой, дополняя увеличиваемые элементы нулями. PUNPCKHBW переводит упакованные байты в упакованные слова, PUNPCKHWD - слова в двойные слова, а PUNPCKHDQ - единственное старшее двойное слово приемника в учетверенное.

Команда	Назначение	Процессор
PUNPCKLBW приемник,источник	Распаковка и объединение младших элементов	MMX
PUNPCKLWD приемник,источник	Распаковка и объединение младших элементов	MMX
PUNPCKLDQ приемник,источник	Распаковка и объединение младших элементов	MMX

Команды распаковывают младшие элементы источника (регистр MMX или переменная) и приемника (регистр MMX) и записывают их в приемник через один аналогично предыдущим командам. Команда PUNPCKLBW объединяет по 4 младших байта источника и приемника, команда PUNPCKLWD объединяет по 2 младших слова, а команда PUNPCKLDQ копирует в приемник по одному младшему двойному слову из источника и приемника. Если источник содержит только нули, эти команды, аналогично PUNPCKH*, фактически переводят младшую половину приемника из одного формата данных в другой, дополняя увеличиваемые элементы нулями.

2.5.5. *Арифметические операции MMX*

Команда	**Назначение**	**Процессор**
PADDB приемник,источник	Сложение	MMX
PADDW приемник,источник	Сложение	MMX
PADDD приемник,источник	Сложение	MMX

Команды выполняют сложение отдельных элементов данных (байтов - для PADDB, слов - для PADDW, двойных слов - для PADDD) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если при сложении возникает перенос, он не влияет ни на следующие элементы, ни на флаг переноса, а просто игнорируется (так что, например, для PADDB 255 + 1 = 0, если это числа без знака, или -128 + -1 = +127, если со зна-" ком).

Команда	Назначение	Процессор
PADDSB приемник,источник	Сложение с насыщением	MMX
PADDSW приемник,источник	Сложение с насыщением	MMX

Команды выполняют сложение отдельных элементов данных (байтов - для PADDSB и слов - для PADDSW) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если при сложении сумма выходит за пределы байта со знаком для PADDSB (больше +127 или меньше -128) или слова со знаком для PADDSW (больше +32 767 или меньше -32 768), в качестве результата используется соответствующее максимальное или минимальное число; так что, например, для PADDSB -128 + -1 = -128.

Команда	Назначение	Процессор
PADDUSB приемник, источник	Беззнаковое сложение с насыщением	MMX
PADDUSW приемник,источник	Беззнаковое сложение с насыщением	MMX

Команды выполняют сложение отдельных элементов данных (байтов - для PADDUSB и слов - для PADDUSW) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если при сложении сумма выходит за пределы байта без знака для PADDUSB (больше 255 или меньше 0) или слова без знака для PADDUSW (больше 65 535 или меньше 0), то в качестве результата используется соответствующее максимальное или минимальное число; так что, например, для PADDUSB 255 + 1 = 255.

Команда	Назначение	Процессор
PSUBB приемник,источник	Вычитание	MMX
PSUBW приемник,источник	Вычитание	MMX
PSUBD приемник,источник	Вычитание	MMX

Команды выполняют вычитание отдельных элементов данных (байтов - для PSUBB, слов - для PSUBW, двойных слов - для PSUBD) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если при вычитании возникает заем, он игнорируется (так что, например, для PSUBB -128 - 1 = +127 - применительно к числам со знаком или 0 - 1 = 255 - применительно к числам без знака).

Команда	Назначение	Процессор
PSUBSB приемник,источник	Вычитание с насыщением	MMX
PSUBSW приемник,источник	Вычитание с насыщением	MMX

Команды выполняют вычитание отдельных элементов данных (байтов - для PSUBSB и слов - для PSUBSW) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если при вычитании разность выходит за пределы байта или слова со знаком, в качестве результата используется соответствующее максимальное или минимальное число; так что, например, для PSUBSB -128 - 1 = -128.

Команда	Назначение	Процессор
PSUBUSB приемник, источник	Беззнаковое вычитание с насыщением	MMX
PSUBUSW приемник, источник	Беззнаковое вычитание с насыщением	MMX

Команды выполняют вычитание отдельных элементов данных (байтов — для PSUBUSB и слов - для PSUBUSW) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если при вычитании разность выходит за пределы байта или слова без знака, в качестве результата используется соответствующее максимальное или минимальное число; так что, например, для PSUBUSB 0 – 1 = 0.

Команда	Назначение	Процессор
PMULHW приемник,источник	Старшее умножение	MMX

Команда умножает каждое из четырех слов со знаком из источника (регистр MMX или переменная) на соответствующее слово со знаком из приемника (регистр MMX). Старшее слово каждого из результатов записывается в соответствующую позицию приемника.

Команда	Назначение	Процессор
PMULLW приемник,источник	Младшее умножение	MMX

Умножает каждое из четырех слов со знаком из источника (регистр MMX или переменная) на соответствующее слово со знаком из приемника (регистр MMX). Младшее слово каждого из результатов записывается в соответствующую позицию приемника.

Команда	Назначение	Процессор
PMADDWD приемник,источник	Умножение и сложение	MMX

Умножает каждое из четырех слов со знаком из источника (регистр MMX или переменная) на соответствующее слово со знаком из приемника (регистр MMX). Произведения двух старших пар слов складываются между собой, и их сумма записывается в старшее двойное слово приемника. Сумма произведений двух младших пар слов записывается в младшее двойное слово.

2.5.6. *Команды сравнения MMX*

Команда	Назначение	Процессор
PCMPEQB приемник,источник	Проверка на равенство	MMX
PCMPEQW приемник,источник	Проверка на равенство	MMX
PCMPEQD приемник,источник	Проверка на равенство	MMX

Команды сравнивают индивидуальные элементы данных (байты - в случае PCMPEQB, слова - в случае PCMPEQW, двойные слова - в случае PCMPEQD) источника (регистр MMX или переменная) с элементами приемника (регистр MMX). Если пара сравниваемых элементов равна, соответствующий элемент приемника заполняется единицами, если они не равны - элемент заполняется нулями.

Команда	Назначение	Процессор
PCMPGTB приемник,источник	Сравнение	MMX
PCMPGTW приемник,источник	Сравнение	MMX;
PCMPGTD приемник,источник	Сравнение	MMX

Команды сравнивают индивидуальные элементы данных (байты - в Случае PCMPGTB, слова - в случае PCMPGTW, двойные слова - в случае PCMPGTD) источника (регистр MMX или переменная) с элементами приемника (регистр MMX). Если элемент приемника больше, чем соответствующий элемент Источника, все биты в этом элементе приемника устанавливаются в единицы. Если элемент приемника меньше или равен элементу источника, он обнуляется.

2.5.7. *Логические операции MMX*

Команда	Назначение	Процессор
PAND приемник,источник	Логическое И	MMX

Команда выполняет побитовое «логическое И» над источником (регистр MMX или переменная) и приемником (регистр MMX) и сохраняет результат в приемнике. Каждый бит результата устанавливается в 1, если соответствующие биты в обоих операндах равны 1, в противном случае бит сбрасывается в 0.

Команда	Назначение	Процессор
PANDN приемник,источник	Логическое НЕ-И (штрих Шеффера)	MMX

Выполняет побитовое «логическое НЕ» (то есть инверсию битов) над приемником (регистр MMX) и затем побитовое «логическое И» над приемником и источником (регистр MMX или переменная). Результат сохраняется в приемнике. Каждый бит результата устанавливается в 1, только если соответствующий бит источника был равен 1, а приемника - 0, иначе бит сбрасывается в 0. Эта логическая операция называется также *штрихом Шеффера.*

Команда	Назначение	Процессор
POR приемник,источник	Логическое ИЛИ	MMX

Выполняет побитовое «логическое ИЛИ» над источником (регистр MMX или переменная) и приемником (регистр MMX) и сохраняет результат в приемнике. Каждый бит результата сбрасывается в 0, если соответствующие биты в обоих операндах равны 0, в противном случае бит устанавливается в 1.

Команда	Назначение	Процессор
PXOR приемник,источник	Логическое исключающее ИЛИ	MMX

Выполняет побитовое «логическое исключающее ИЛИ» над источником (регистр MMX или переменная) и приемником (регистр MMX) и сохраняет результат в приемнике. Каждый бит результата устанавливается в 1, если соответствующие биты в обоих операндах равны, иначе бит сбрасывается в 0.

2.5.8. Сдвиговые операции MMX

Команда	Назначение	Процессор
PSLLW приемник,источник	Логический сдвиг влево	MMX
PSLLD приемник,источник	Логический сдвиг влево	MMX
PSLLQ приемник,источник	Логический сдвиг влево	MMX

Команды сдвигают влево биты в каждом элементе (в словах - для PSLLW, в двойных словах - для PSLLD, во всем регистре - для PSLLQ) приемника (регистр MMX) на число битов, указанное в источнике (8-битное число, регистр MMX или переменная). При сдвиге младшие биты заполняются нулями, так что, например, команды

```
psllw   mm0,15
pslld   mm0,31
psllq   mm0,63
```

обнуляют регистр MMO.

Команда	Назначение	Процессор
PSRLW приемник,источник	Логический сдвиг вправо	MMX
PSRLD приемник,источник	Логический сдвиг вправо	MMX
PSRLQ приемник,источник	Логический сдвиг вправо	MMX

Команды сдвигают вправо биты в каждом элементе (в словах - для PSRLW, в двойных словах - для PSRLD, во всем регистре - для PSRLQ) приемника (регистр MMX) на число битов, указанное в источнике (8-битное число, регистр MMX или переменная). При сдвиге старшие биты заполняются нулями.

Команда	Назначение	Процессор
PSRAW приемник,источник	Арифметический сдвиг вправо	MMX
PSRAD приемник,источник	Арифметический сдвиг вправо	MMX

Команды сдвигают вправо биты в каждом элементе (в словах - для PSRAW и в двойных словах - для PSRAD) приемника (регистр MMX) на число битов,

указанное в источнике (8-битное число, регистр MMX или переменная). При сдвиге самый старший (знаковый) бит используется для заполнения пустеющих старших битов, так что фактически происходит знаковое деление на 2 в степени, равной содержимому источника.

2.5.9. *Команды управления состоянием MMX*

Команда	Назначение	Процессор
EMMS	Освободить регистры MMX	MMX

Если выполнялись какие-нибудь команды MMX (кроме EMMS), все регистры FPU помечаются как занятые (в регистре TW). Команда EMMS помечает все регистры FPU как пустые для того, чтобы после завершения работы с MMX можно было передать управление процедуре, использующей FPU.

2.5.10. *Расширение AMD 3D*

Процессоры AMD, начиная с AMD K6 3D, поддерживают дополнительное расширение набора команд MMX. В AMD 3D вводится новый тип данных - упакованные 32-битные вещественные числа, определяются новые команды (начинающиеся с PF) и несколько дополнительных команд для работы с обычными MMX-типами данных:

- *PI2FD приемник,источник* - преобразовывает упакованные 32-битные целые со знаком (двойные слова) в упакованные вещественные числа;
- *PF2IDприемник,источник* - преобразовывает упакованные вещественные в упакованные целые числа со знаком (преобразование с насыщением);
- *PAVGUSB приемник,источник* - вычисляет средние арифметические для упакованных 8-битных целых чисел без знака;
- *PMULHRWприемник,источник* — перемножает упакованные 16-битные целые со знаком и сохраняет результаты как 16-битные целые в приемнике (при переполнениях выполняется насыщение);
- а *PFACC приемник,источник* - сумма вещественных чисел в приемнике помещается в младшую половину приемника, сумма вещественных чисел из источника помещается в старшую половину приемника;
- *PFADDприемник,источник* - сложение упакованных вещественных чисел;
- *PFSUBприемник,источник* - вычитание упакованных вещественных чисел;
- *PFSUBRприемник,источник* - обратное вычитание (приемник из источника) упакованных вещественных чисел;
- *PFMULприемник,источник* - умножение упакованных вещественных чисел.

Набор команд для быстрого вычисления по итерационным формулам:
Быстрое деление:

$x_{i+1} - x_i(2 - bx_i)$

$x_0 = PFRCP(b)$

$x_1 = PFRCPIT1(b,x_0)$

$x_2 = PFRCPIT2(x_1,x_0)$
$x_{i+1} = PFMUL(b,x_3)$
Быстрое вычисление квадратного корня:
$x_{i+1} - x_i(3 - bx_i^2)/2$
$x_0 = PFRSQRT(b)$
$x_1 - PFMUL(x_0,x_0)$
$x_2 = PFRSQIT(b,x_1)$
$x_3 = PFRCPIT2(x_2,x_0)$
$x_{i+1} - PFMUL(b,x_3)$

- *PFCMPEQприемник,источник* - проверка равенства для упакованных вещественных чисел (полностью аналогично PCMPEQW);
- *PFCMPGE приемник,источник* - сравнение упакованных вещественных чисел: если число в приемнике больше или равно числу в источнике, все его биты устанавливаются в 1;
- *PFCMPGT приемник,источник* - сравнение упакованных вещественных чисел: если число в приемнике больше числа в источнике, все его биты устанавливаются в 1;
- *PFMAXприемник,источник* - сохраняет в приемнике максимальное из каждой пары сравниваемых вещественных чисел;
- *PFMIN приемник,источник* - сохраняет в приемнике минимальное из каждой пары сравниваемых вещественных чисел;
- *FEMMS* - более быстрая версия команды EMMS;
- *PREFETCH источник* - заполняет строку кэша L1 из памяти по адресу, указанному источником;
- *PREFETCHW источник* - заполняет строку кэша L1 из памяти по адресу, указанному источником, и помечает как модифицированную.

2.6. Расширение SSE

2.6.1. Регистры *SSE*

Со времени процессора Pentium III (Katmai) появилось новое расширение SSE (Streaming SIMD Extensions - потоковые SIMD-расширения), где SIMD (Single Instruction - Multiple Data) - общий для SSE и MMX подход к обработке большого количества данных одной командой. Расширение предназначается для современных приложений, работающих с двумерной и трехмерной графикой, видео-, аудио- и другими видами потоковых данных.

В отличие от MMX, это расширение не использует уже существующие ресурсы процессора, а вводит 8 новых независимых 128-битных регистров данных: XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 и XMM7. Таким образом решаются проблемы технологии MMX - не требуется команд типа EMMS для переключения режимов и можно пользоваться другими расширениями, работая с SSE.

Кроме *восьми* регистров данных, вводится дополнительный 32-битный регистр управления/состояния MXCSR, который используется для маскирования исключений, выбора режимов и определения состояния флагов:

бит 0: произошло исключение IE
бит 1: произошло исключение DE
бит 2: произошло исключение ZE
бит 3: произошло исключение OE
бит 4: произошло исключение UE
бит 5: произошло исключение PE
бит 6: зарезервирован (всегда 0)
бит 7: IM - маска исключения IE
бит 8: DM - маска исключения DE
бит 9: ZM - маска исключения ZE
бит 10: OM - маска исключения OE
бит 11: UM - маска исключения UE
бит 12: PM - маска исключения PE
биты 14–13: RC - управление округлением
бит 15: FZ - режим сброса в ноль (flush-to-zero)
биты 31-16: зарезервированы (всегда 0)

Все маскирующие биты по умолчанию (при включении процессора) устанавливаются в 1, так что никакие исключения не обрабатываются.

Поле RC определяет режим округления: 00 - к ближайшему числу, 01 - к отрицательной бесконечности, 10 - к положительной бесконечности, 11 - к нулю. По умолчанию устанавливается в режим округления к ближайшему числу.

Бит FZ включает режим сброса в ноль (по умолчанию выключен). В этом режиме команды SSE не превращают слишком маленькое число с плавающей запятой в денормализованное (как этого требует стандарт IEEE), а возвращают ноль. Знак нуля соответствует знаку получившегося бы денормализованного числа, и, кроме того, устанавливаются флаги PE и UE.

2.6.2. *Типы данных SSE*

Основной тип данных, с которым работают команды SSE, -упакованные числа с плавающей запятой одинарной точности. В одном 128-битном регистре размещаются сразу четыре таких числа - в битах 127-96 (число 3), 95-64 (число 2), 63-32 (число 1) и 31-0 (число 0). Это стандартные 32-битные числа с плавающей запятой, используемые числовым сопроцессором. Целочисленные команды SSE могут работать с упакованными байтами, словами или двойными словами. Однако эти команды оперируют данными, находящимися в регистрах MMX.

2.6.3. *Команды SSE*

Все команды SSE доступны из любых режимов процессора — реального, защищенного и режима V86.

*Команды **пересылки** данных*

Команда	Назначение	Процессор
MOVAPS приемник,источник	Переслать выравненные упакованные числа	PHI

Копирует 128 бит из источника в приемник. Каждый из аргументов может быть либо регистром SSE, либо переменной в памяти, но пересылки типа память-память запрещены. Если адрес переменной некратен 16 байтам (128 битам), вызывается исключение #GP.

Команда	Назначение	Процессор
MOVUPS приемник,источник	Переслать невыравненные упакованные числа	PIII

Копирует 128 бит из источника в приемник. Каждый из аргументов может быть либо регистром SSE, либо переменной в памяти, но пересылки типа память-память запрещены. В тех случаях, когда легко достичь выравнивания всех данных по адресам, кратным 16 байт, рекомендуется пользоваться командой MOVAPS, так как она более эффективна.

Команда	Назначение	Процессор
MOVHPS приемник,источник	Переслать старшие упакованные числа	PHI

Копирует старшие 64 бита из источника в приемник. Младшие 64 бита приемника не изменяются. Каждый из аргументов может быть либо регистром SSE, либо переменной в памяти, но пересылки типа память-память запрещены.

Команда	Назначение	Процессор
MOVLPS приемник,источник	Переслать младшие упакованные числа	PIII

Копирует младшие 64 бита из источника в приемник. Старшие 64 бита приемника не изменяются. Один из аргументов должен быть регистром SSE, другой - переменной в памяти.

Команда	Назначение	Процессор
MOVHLPS приемник,источник	Переслать старшие упакованные числа в младшие	PHI

Копирует старшие 64 бита источника в младшие 64 бита приемника. Старшие 64 бита приемника не изменяются. И приемником, и источником могут быть только регистры SSE.

Команда	Назначение	Процессор
MOVLHPS приемник,источник	Переслать младшие упакованные числа в старшие	PIII

Копирует младшие 64 бита источника в старшие 64 бита приемника. Младшие 64 бита приемника не изменяются. И приемником, и источником могут быть только регистры SSE.

Команда	Назначение	Процессор
MOVMSKPS приемник,источник	Переслать маску в переменную	PHI

В приемник (32-битный регистр центрального процессора) записывается 4-битная маска, отвечающая знакам четырех вещественных чисел, находящихся в источнике (128-битный регистр SSE). Фактически бит 0 приемника устанавливается равным биту 31 источника, бит 1 - биту 63, бит 2 - биту 95, бит 3 - биту 127, а биты 4-31 приемника обнуляются.

Команда	Назначение	Процессор
MOVSS приемник,источник	Переслать одно вещественное число	PIII

Копирует младшие 64 бита из источника в приемник. Если приемник - регистр, его старшие 96 бит обнуляются. Если приемник — переменная в памяти, старшие 96 бит не изменяются. Каждый из аргументов может быть либо регистром SSE, либо переменной в памяти, но пересылки типа память-память запрещены.

Арифметические команды

Команда	Назначение	Процессор
ADDPS приемник,источник	Сложение упакованных вещественных чисел	PHI

Выполняет параллельное сложение четырех пар чисел с плавающей запятой, находящихся в источнике (переменная или регистр SSE) и приемнике (регистр SSE). Результат записывается в приемник.

Команда	Назначение	Процессор
ADDSS приемник,источник	Сложение одного вещественного числа	PIII

Выполняет сложение нулевых (занимающих биты 31-0) чисел с плавающей запятой в источнике (переменная или регистр SSE) и приемнике (регистр SSE). Результат записывается в биты 31-0 приемника, биты 127-32 остаются без изменений.

Команда	Назначение	Процессор
SUBPS приемник,источник	Вычитание упакованных вещественных чисел	PHI

Выполняет параллельное вычитание чисел с плавающей запятой, находящихся в источнике (переменная или регистр SSE), из чисел, находящихся в приемнике (регистр SSE). Результат записывается в приемник.

Команда	Назначение	Процессор
SUBSS приемник,источник	Вычитание одного вещественного числа	PIII

Выполняет вычитание нулевого (занимающего биты 31–0) числа с плавающей запятой в источнике (переменная или регистр SSE) из числа, находящегося в приемнике (регистр SSE). Результат записывается в биты 31–0 приемника, биты 127–32 остаются без изменений.

Команда	Назначение	Процессор
MULPS приемник,источник	Умножение упакованных вещественных чисел	PHI

Выполняет параллельное умножение четырех пар чисел с плавающей запятой, находящихся в источнике (переменная или регистр SSE) и приемнике (регистр SSE). Результат записывается в приемник.

Команда	Назначение	Процессор
MULSS приемник,источник	Умножение одного вещественного числа	PIII

Выполняет умножение нулевых (занимающих биты 31-0) чисел с плавающей запятой в источнике (переменная или регистр SSE) и приемнике (регистр SSE). Результат записывается в биты 31-0 приемника, биты 127-32 остаются без изменений.

Команда	Назначение	Процессор
DIVPS приемник,источник	Деление упакованных вещественных чисел	PHI

Выполняет параллельное деление четырех пар чисел с плавающей запятой, находящихся в приемнике (регистр SSE), на числа, находящиеся в источнике (переменная или регистр SSE). Результат записывается в приемник.

Команда	Назначение	Процессор
DIVSS приемник,источник	Деление одного вещественного числа	PHI

Выполняет деление нулевого (занимающего биты 31-0) числа с плавающей запятой в приемнике (регистр SSE) на нулевое число, находящееся в источнике (переменная или регистр SSE). Результат записывается в биты 31-0 приемника, биты 127-32 остаются без изменений.

Команда	Назначение	Процессор
SQRTPS приемник,источник	Корень из упакованных вещественных чисел	PHI

Определяет значение квадратных корней от каждого из четырех чисел с плавающей запятой, находящихся в источнике (регистр SSE или переменная), и записывает их в приемник (регистр SSE).

Команда	Назначение	Процессор
SQRTSS приемник,источник	Корень из одного вещественного числа	PIII

Определяет значение квадратного корня из нулевого (занимающего биты 31–0) числа с плавающей запятой из источника (регистр SSE или переменная) и записывает результат в биты 31-0 приемника (регистр SSE).

Команда	Назначение	Процессор
RCPPS приемник,источник	Обратная величина для упакованных чисел	PIII

Выполняет деление единицы на каждое из четырех чисел с плавающей запятой, находящихся в источнике (регистр SSE или переменная), и записывает результаты в приемник (регистр SSE). Максимальное значение ошибки - $1{,}5\times2^{-12}$.

Команда	Назначение	Процессор
RCPSS приемник,источник	Обратная величина для одного числа	PIII

Выполняет деление единицы на нулевое (занимающее биты 31-0) число с плавающей запятой из источника (регистр SSE или переменная) и записывает результат в биты 31-0 приемника (регистр SSE). Максимальное значение ошибки - $1{,}5\times2^{-12}$.

Команда	Назначение	Процессор
RSQRTPS приемник,источник	Обратный корень из упакованных чисел	PHI

Определяет обратные величины от квадратных корней (1/sqrt()) каждого из четырех чисел с плавающей запятой, находящихся в источнике (регистр $SE или переменная), и записывает их в приемник (регистр SSE). Максимальное значение ошибки - $1{,}5\times2^{-12}$.

Команда	Назначение	Процессор
RSQRTSS приемник,источник	Обратный корень из одного числа	PIII

Определяет обратную величину от квадратного корня (1/sqrt()) нулевого числа (занимающего биты 31–0) числа с плавающей запятой из источника (регистр SSE или переменная) и записывает результат в биты 31-0 приемника (регистр SSE). Максимальное значение ошибки - $1{,}5\times2^{-12}$.

Команда	Назначение	Процессор
MAXPS приемник,источник	Максимум для упакованных вещественных чисел	PIII

Определяет максимальные числа с плавающей запятой в каждой из четырех пар чисел, находящихся в источнике (переменная или регистр SSE) и приемнике (регистр SSE). Результат записывается в приемник. Если источник или приемник содержит не-число (SNAN), оно возвращается в приемник без изменений. При сравнении двух нулей возвращается нуль из источника. Если не-число сравнивается с другим не-числом, то возвращается не-число из приемника.

Команда	Назначение	Процессор
MAXSS приемник,источник	Максимум для одной пары вещественных чисел	PIII

Определяет максимальные числа с плавающей запятой в нулевой паре чисел (биты 31–0), находящихся в источнике (переменная или регистр SSE) и приемнике (регистр SSE). Результат записывается в приемник. Биты 127-32 приемника

не изменяются. Если источник или приемник содержит не-число (SNAN), оно возвращается в приемник без изменений. При сравнении двух нулей возвращается нуль из источника. Если не-число сравнивается с другим не-числом, возвращается не-число из приемника.

Команда	Назначение	Процессор
MINPS приемник,источник	Минимум для упакованных вещественных чисел	PHI

Определяет минимальные числа с плавающей запятой в каждой из четырех пар чисел, находящихся в источнике (переменная или регистр SSE) и приемнике (регистр SSE). Результат записывается в приемник. Если источник или приемник включает не-число (SNAN), возвращается содержимое другого не-числа из аргументов. При сравнении двух нулей возвращается нуль из источника. Если не-число сравнивается с другим не-числом, возвращается не-число из источника.

Команда	Назначение	Процессор
MINSS приемник,источник	Минимум для одной пары вещественных чисел	PHI

Определяет минимальные числа с плавающей запятой в нулевой паре чисел (биты 31-0), находящихся в источнике (переменная или регистр SSE) и приемнике (регистр SSE). Результат записывается в приемник. Биты 127-32 приемника не изменяются. Если источник или приемник включает не-число (SNAN), то возвращается содержимое другого аргумента. При сравнении двух нулей возвращается нуль из источника. Если не-число сравнивается с другим не-числом, возвращается не-число из источника.

Команды сравнения

Команда	Назначение	Процессор
CMPPS приемник,источник,предикат	Сравнение упакованных вещественных чисел	PIII

Для каждой из четырех пар вещественных чисел, находящихся в источнике (переменная или регистр SSE) и приемнике (регистр SSE), возвращает либо О (ложь), либо OFFFFFFFh (истина), в зависимости от результата сравнения. Тип сравнения определяется предикатом (число):

Предикат	Проверяемое утверждение
0 (eq)	Приемник равен источнику
1 (lt)	Приемник строго меньше источника
2 (le)	Приемник меньше или равен источнику
3 (unord)	Приемник или источник являются не-числом
4 (neq)	Приемник не равен источнику
5 (nlt)	Приемник больше или равен источнику

Предикат	Проверяемое утверждение
6 (nle)	Приемник строго больше источника
7 (ord)	Ни приемник, ни источник не являются не-числом

Если один из операндов - не-число, результатом сравнения является 0 для предикатов О, 1, 2, 7 и истина для предикатов 3, 4, 5, 6.

Команда	Назначение	Процессор
CMPSS приемник,источник, предикат	Сравнение одной пары упакованных чисел	PIII

Выполняет сравнение нулевых (занимающих биты 31–0) вещественных чисел из источника и приемника аналогично команде CMPPS. Биты 127-32 приемника не изменяются.

Команда	Назначение	Процессор
COMISS приемик,источник	Сравнение одной пары чисел с установкой флагов	PIII

Выполняет сравнение нулевых (занимающих биты 31–0) вещественных чисел из источника (переменная или регистр SSE) и приемника (регистр SSE) и устанавливает флаги ZF, PF, CF регистра EFLAGS в соответствии с результатом. Флаги OF, SF, AF обнуляются. Если одно из сравниваемых чисел - не-число, все три флага (ZF, PF, CF) устанавливаются в 1. Если сравниваемые числа равны, то ZF = 1, PF = CF = 0. Если приемник меньше источника, то CF = 1, ZF = PF = 0. Если приемник больше источника - CF = ZF = PF = 0.

Команда	Назначение	Процессор
UCOMISS приемик,источник	Сравнение одной пары неупорядоченных чисел с установкой флагов	PHI

Эта команда полностью аналогична COMISS, но она приводит к исключению #I, если один из операндов SNAN или QNAN; UCOMISS только если один из операндов - SNAN.

Команды преобразования типов

Команда	Назначение	Процессор
CVTPI2PS приемник,источник	Преобразовать упакованные целые в вещественные	PIII

Преобразует два 32-битных целых числа со знаком из источника (регистр MMX или 64-битная переменная) в два упакованных вещественных числа в приемнике (регистр SSE). Если преобразование нельзя выполнить точно, результат округляется в соответствии с MXCSR. Биты 127-64 приемника не изменяются.

Команда	Назначение	Процессор
CVTPS2PI приемник,источник	Преобразовать упакованные вещественные в целые	PIII

Преобразует младшие два 32-битных вещественных числа из источника (регистр SSE или 64-битная переменная) *в* два упакованных целых числа со знаком в приемнике (регистр MMX). Если преобразование нельзя выполнить точно, результат округляется в соответствии с MXCSR. Если результат больше максимального 32-битного числа со знаком, возвращается целая неопределенность (80000000h).

Команда	Назначение	Процессор
CVTSI2SS приемник,источник	Преобразовать целое в вещественное	PIII

Преобразует 32-битное целое число со знаком из источника (переменная или 32-битный регистр) в вещественное число в приемнике (регистр SSE). Если преобразование нельзя выполнить точно, результат округляется в соответствии с MXCSR. Биты 127-32 приемника не изменяются.

Команда	Назначение	Процессор
CVTSS2SI приемник,источник	Преобразовать вещественное в целое	PIII

Преобразует нулевое (младшее) вещественное число из источника (регистр SSE или 32-битная переменная) в 32-битное целое число со знаком в приемнике (32-битный регистр). Если преобразование нельзя выполнить точно, результат округляется в соответствии с MXCSR. Если результат больше максимального 32-битного числа со знаком, возвращается целая неопределенность (80000000h).

Команда	Назначение	Процессор
CVTTPS2PI приемник,источник	Преобразование вещественных в целые с обрезанием	PIII

Выполняется аналогично CVTPS2PI, но, если результат не может быть представлен точно, он всегда округляется в сторону нуля (обрезается).

Команда	Назначение	Процессор
CVTTSS2SI приемник,источник	Преобразование вещественного в целое с обрезанием	PHI

Выполняется аналогично CVTSS2SI, но, если результат не может быть представлен точно, он всегда округляется в сторону нуля (обрезается).

Логические операции

Команда	Назначение	Процессор
ANDPS приемник,источник	Логическое И для SSE	PIII

Выполняет операцию побитового «логического И» для источника (регистр SSE или 128-битная переменная) и приемника (регистр SSE) и помещает результат в приемник.

Команда	Назначение	Процессор
ANDNPS приемник,источник	Логическое НЕ-И для SSE	PHI

Выполняет операцию НЕ над содержимым приемника (регистр SSE), затем выполняет операцию И над результатом и содержимым источника (регистр SSE или 128-битная переменная) и записывает результат в приемник.

Команда	Назначение	Процессор
ORPS приемник,источник	Логическое ИЛИ для SSE	PHI

Выполняет операцию побитового «логического ИЛИ» для источника (регистр SSE или 128-битная переменная) и приемника (регистр SSE) и помещает результат в приемник.

Команда	Назначение	Процессор
XORPS приемник,источник	Логическое исключающее ИЛИ для SSE	PIII

Выполняет операцию побитового «логического исключающего ИЛИ» для источника (регистр SSE или 128-битная переменная) и приемника (регистр SSE) и помещает результат в приемник.

Целочисленные SIMD-команды

Помимо расширения для работы с упакованными вещественными числами в SSE входит расширение набора команд для работы с упакованными целыми числами, которые размещаются в регистрах MMX.

Команда	Назначение	Процессор
PAVGB приемник,источник	Усреднение байтов с округлением	PIII
PAVGW приемник,источник	Усреднение слов с округлением	PIII

Каждый элемент (байт или слово) источника (регистр MMX или 64-битная переменная) добавляется к соответствующему элементу приемника (регистр MMX) как беззнаковое целое. Каждый из результатов сдвигается вправо на один бит (делится на два). Затем в старший бит каждого элемента записывается бит переноса от соответствующего сложения. В результате этих действий получаются средние арифметические целых чисел со знаками.

Команда	Назначение	Процессор
PEXTRW приемник,источник,индекс	Распаковать одно слово	PIII

Выделяет 16-битное слово из источника (регистр MMX) с номером, определяемым как младшие два бита индекса (непосредственно заданное число), и помещает его в младшую половину 32-битного регистра-приемника.

Команда	Назначение	Процессор
PINSRW приемник,источник,индекс	Запаковать одно слово	PHI

Считывает слово из источника (16-битная переменная или 32-битный регистр, во втором случае используется младшая половина регистра) и помещает

его в приемник (регистр MMX) в положение, задаваемое младшими двумя битами индекса (непосредственно заданное число). Другие три слова в приемнике не изменяются.

Команда	Назначение	Процессор
PMAXUB приемник,источник	Максимум для упакованных байтов	PHI

Для каждой из восьми пар упакованных байтов из источника (регистр MMX или 64-битная переменная) или приемника (регистр MMX) в приемник записывается максимальный байт в паре. Сравнение выполняется без учета знака.

Команда	Назначение	Процессор
PMAXSW приемник,источник	Максимум для упакованных слов	PHI

Для каждой из четырех пар упакованных слов из источника (регистр MMX или 64-битная переменная) или приемника (регистр MMX) в приемник записывается максимальное слово в паре. Сравнение выполняется с учетом знака.

Команда	Назначение	Процессор
PMINUB приемник,источник	Минимум для упакованных байтов	PIII

Для каждой из восьми пар упакованных байтов из источника (регистр MMX или 64-битная переменная) или приемника (регистр MMX) в приемник записывается минимальный байт в паре. Сравнение выполняется без учета знака.

Команда	Назначение	Процессор
PMINSW приемник,источник	Минимум для упакованных слов	PIII

Для каждой из четырех пар упакованных слов из источника (регистр MMX или 64-битная переменная) или приемника (регистр MMX) в приемник записывается минимальное слово в паре. Сравнение выполняется с учетом знака.

Команда	Назначение	Процессор
PMOVMSKB приемник,источник	Считать байтовую маску	PIII

В приемнике (32-битный регистр) каждый из младших 8 бит устанавливается равным старшему (знаковому) биту соответствующего байта источника (регистр MMX). Биты 31-8 приемника обнуляются.

Команда	Назначение	Процессор
PMULHUW приемник,источник	Старшее умножение без знака	PHI

Умножить упакованные беззнаковые слова из источника (регистр MMX или 64-битная переменная) и из приемника (регистр MMX) и поместить старшие 16 бит 32-битного результата в соответствующее слово в приемнике.

Команда	Назначение	Процессор
PSADBW приемник,источник	Сумма абсолютных разностей	PIII

Вычисляет абсолютные разности восьми пар байтов из источника (регистр MMX или 64-битная переменная) и приемника (регистр MMX) как целых чисел без знака, затем суммирует результаты и помещает их в младшее (нулевое) слово в приемнике. Старшие три слова обнуляются.

Команда	Назначение	Процессор
SHUFW приемник,источник,индекс	Переставить упакованные слова	PIII

Вместо каждого из четырех слов приемника (регистр MMX) размещается слово из источника (регистр MMX или 64-битная переменная) с номером, указанным в соответствующей паре битов индекса (непосредственно заданное 8-битное число). Так, вместо слова 0 (биты 15–0) приемника будет записано слово из источника с номером, равным значению битов 1 и 0 индекса. Например, если индекс равен 10101010b, второе слово источника будет скопировано во все четыре слова приемника.

Команды упаковки

Команда	Назначение	Процессор
SHUFPS приемник,источник,индекс	Переставить упакованные вещественные	PIII

Помещает в старшие два вещественных числа приемника (регистр SSE) любые из четырех чисел, находившихся в источнике (регистр SSE или 128-битная переменная). В младшие два числа приемника помещает любые из четырех чисел, находившихся в приемнике. По индексу (непосредственный операнд) определяется, какие именно числа упаковываются подобным образом. Биты 1 и 0 указывают номер числа из приемника, которое будет записано в нулевую позицию приемника; биты 3 и 2 - номер числа из приемника, которое будет записано в первую позицию приемника. Биты 5 и 4 устанавливают номер числа из источника, которое будет записано в третью позицию, а биты 7 и 6 - номер числа из источника, которое будет записано в четвертую позицию.

Команда	Назначение	Процессор
UNPCKHPS приемник,источник	Распаковать старшие вещественные числа	PIII

В нулевую позицию приемника (регистр SSE) записывается второе число из приемника, в первую позицию - второе число из источника (регистр SSE или 128-битная переменная), во вторую позицию - третье (старшее) число приемника, в третью (старшую) позицию - третье (старшее) число источника.

Команда	Назначение	Процессор
UNPCKLPS приемник,источник	Распаковать младшие вещественные числа	PHI

В нулевую позицию приемника (регистр SSE) записывается нулевое (младшее) число из приемника, в первую позицию - нулевое (младшее) число из источника (регистр SSE или 128-битная переменная), во вторую позицию — первое число приемника, в третью (старшую) позицию - первое число источника.

Команды управления состоянием

Команда	Назначение	Процессор
LDMXCSR источник	Загрузить регистр MXCSR	PIII

Помещает значение источника (32-битная переменная) в регистр управления и состояния SSE MXCSR.

Команда	Назначение	Процессор
STMXCSR приемник	Сохранить регистр MXCSR	PIII

Помещает значение регистра MXCSR в приемник (32-битная переменная).

Команда	Назначение	Процессор
FXSAVE приемник	Сохранить состояние FPL), MMX, SSE	PIII

Сохраняет содержимое всех регистров FPU, MMX и SSE в приемнике (512-байтовая область памяти).

Команда	Назначение	Процессор
FXRSTOR источник	Восстановить состояние FPU, MMX, SSE	PIII

Восстанавливает содержимое всех регистров FPU, MMX и SSE из источника (512-байтовой области памяти, заполненной командой FXSAVE).

Формат области памяти, используемой командами FXSAVE/FXRSTOR для Pentium III, имеет следующий вид:

байты 1–0: FCW
байты 3-2: FSW
байты 5-4: FTW
байты 7–6: FOP
байты 11–8: FIP
байты 13-12: FCS
байты 19-16: FDP
байты 21-20: FDS
байты 27-24: MXCSR
байты 41-32: STO или MMO
байты 57-48: ST1 или MM1
байты 73-64: ST2 или MM2
байты 89-80: ST3 или MM3
байты 105-96: ST4 или MM4
байты 121-112: ST5 или MM5

байты 137-128: ST6 или MM6
байты 153-144: ST7 или MM7
байты 175-160: XMMO
байты 191–176:XMM1
байты 207-192: XMM2
байты 223-208: XMM3
байты 239-224: XMM4
байты 255-240: XMM5
байты 271-256: XMM6
байты 287-272: XMM7

Остальные байты зарезервированы.

Команды управления кэшированием

Команда	Назначение	Процессор
MASKMOVQ источник,маска	Запись байтов минуя кэш	PHI

Данные из источника (регистр MMX) записываются в память по адресу DS:EDI (или DS:DI). При этом старший бит каждого байта в маске (регистр MMX) определяет, записывается ли соответствующий байт источника в память или нет. То есть бит 7 маски разрешает запись нулевого байта (битов 7-0) источника и т. д. Если байт не записывается, соответствующий байт в памяти обнуляется. Эта команда введена для того, чтобы по возможности уменьшить загрязнение кэша при работе с потоками данных, типичными для SSE, если основным типом данных является байт.

Команда	Назначение	Процессор
MOVNTQ приемник,источник	Запись 64 бит минуя кэш	PHI

Содержимое источника (регистр MMX) записывается в приемник (64-битная переменная в памяти), сводя к минимуму загрязнение кэша.

Команда	Назначение	Процессор
MOVNTPS приемник, источник	Запись 128 бит минуя кэш	PIII

Содержимое источника (регистр SSE) записывается в приемник (128-битная переменная в памяти), сводя к минимуму загрязнение кэша.

Команда	Назначение	Процессор
PREFETCHT0 адрес	Перенести данные в кэш TO	PIII
PREFETCHT1 адрес	Перенести данные в кэш T1	PIII
PREFETCHT2 адрес	Перенести данные в кэш T2	PIII
PREFETCHNTA адрес	Перенести данные в кэш NTA	PHI

Эти команды перемещают данные, располагающиеся по указанному адресу, в кэш. При этом возможны следующие варианты:

- TO - поместить данные в кэш всех уровней;
- T1 - пометить данные в кэш всех уровней, кроме нулевого;
- T2 — поместить данные в кэш всех уровней, кроме нулевого и первого;
- NTA - поместить данные в кэш для постоянных данных.

Реализация этих команд может отличаться для разных процессоров, и процессор не обязан их выполнять - команды рассматриваются только как подсказки. Объем данных, переносимых в кэш, также может различаться, но не должен быть меньше 32 байт.

Команда	Назначение	Процессор
SFENCE	Защита записи	PIII

При работе с памятью современные процессоры могут выполнить обращения к ней совсем не так и не в том порядке, в каком они указаны в программе. Команда SFENCE гарантирует, что все операции записи в память, расположенные в тексте программы до нее, будут выполнены раньше, чем процессор начнет выполнять операции, помещенные в тексте программы позднее.

2.6.4. *Определение поддержки SSE*

Перед тем как начинать работать с расширениями SSE (согласно документации Intel), нужно убедиться, что выполнены следующие три условия:

1. Бит 2 регистра CRO (эмуляция сопроцессора) должен быть равен нулю.
2. Бит 9 регистра CR4 (поддержка команд FXSAVE/FXRSTOR) должен быть равен 1.
3. Бит 25 регистра EDX после команды CPUID (поддержка SSE) должен быть равен 1.

2.6.5. Исключения

Особые ситуации при выполнении команд SSE вызывают новое системное исключение #XF (INT 19), обработчик которого может прочитать содержимое регистра MXCSR, чтобы определить тип исключения и выполнить соответствующие действия. Кроме того, команды SSE могут вызывать и обычные системные исключения - #UD (неопределенная команда), #NM (расширение отсутствует), #SS (переполнение стека), #GP (общая ошибка защиты), #PF (ошибка страничной защиты), #AC (невыровненное обращение к памяти).

Собственные исключения, вызываемые командами SSE и отраженные при помощи флагов в регистре MXCSR, - это:

- Q #I — невыполнимая команда (вызывается перед выполнением команды);
- Q #Z - деление на ноль (вызывается перед выполнением команды);
- #D - денормализованный операнд (вызывается перед выполнением команды);
- #O - переполнение (вызывается после выполнения команды);
- #U - антипереполнение (вызывается после выполнения команды);
- #P - потеря точности (вызывается после выполнения команды).

Глава 3. Директивы и операторы ассемблера

Каждая программа на языке ассемблера помимо команд процессора содержит еще и специальные инструкции, указывающие самому ассемблеру, как организовывать различные секции программы, где располагаются данные, а где команды, позволяющие создавать макроопределения, выбирать тип используемого процессора, налаживать связи между процедурами и т. д. К сожалению, пока нет единого стандарта на эти команды (он существует для UNIX, о чем рассказано в главе 11). Разные ассемблеры используют различные наборы директив, но TASM и MASM (два самых популярных ассемблера для DOS и Windows) поддерживают общий набор, или, точнее, TASM поддерживает набор директив MASM наряду с несовместимым собственным, известным как Ideal Mode. Все примеры программ в книге написаны так, чтобы для их компиляции можно было воспользоваться TASM, MASM или WASM - еще одним популярным ассемблером, поэтому в данной главе рассмотрены те предопределенные идентификаторы, операторы и директивы, которые поддерживаются этими тремя ассемблерами одновременно.

3.1. Структура программы

Программа на языке ассемблера состоит из строк, имеющих следующий вид:

```
метка    команда/директива операнды          ; комментарий
```

Причем все эти поля необязательны. Метка может быть любой комбинацией букв английского алфавита, цифр и символов _, $, @, ?, но цифра не может быть первым символом метки, а символы $ и ? иногда имеют специальные значения и обычно не рекомендуются к использованию. Большие и маленькие буквы по умолчанию не распознаются, но различие можно включить, задав ту или иную опцию в командной строке ассемблера. Во втором поле, поле команды, может располагаться команда процессора, которая транслируется в исполняемый код, или директива, которая не приводит к появлению нового кода, а управляет работой самого ассемблера. В поле операндов располагаются требуемые командой или директивой операнды (то есть нельзя указать операнды и не указать команду или директиву). И наконец, в поле комментариев, начало которого отмечается символом ; (точка с запятой), можно написать все что угодно - текст от символа ; до конца строки не анализируется ассемблером.

Для облегчения читаемости ассемблерных текстов принято, что метка начинается на первой позиции в строке, команда - на 17-й (две табуляции), операнды -

на 25-й (три табуляции) и комментарии - на 41-й или 49-й. Если строка состоит только из комментария, его начинают с первой позиции.

Если метка располагается перед командой процессора, сразу после нее всегда ставится оператор : (двоеточие), который указывает ассемблеру, что надо создать переменную с этим именем, содержащую адрес текущей команды:

```
some_loop:
        lodsw                   ; Считать слово из строки.
        cmp     ax,7            ; Если это 7 - выйти из цикла.
        loopne  some_loop
```

Когда метка стоит перед директивой ассемблера, она обычно оказывается одним из операндов этой директивы и двоеточие не ставится. Рассмотрим директивы, работающие напрямую с метками и их значениями, - LABEL, EQU и =.

метка **label** тип

Директива LABEL определяет метку и задает ее тип: BYTE (байт), WORD (слово), DWORD (двойное слово), FWORD (6 байт), QWORD (учетверенное слово), TBYTE (10 байт), NEAR (ближняя метка), FAR (дальняя метка). Метка получает значение, равное адресу следующей команды или следующих данных, и тип, указанный явно. В зависимости от типа команда

```
        mov     метка,0
```

запишет в память байт (слово, двойное слово и т. д.), заполненный нулями, а команда

```
        call    метка
```

выполнит ближний или дальний вызов подпрограммы.

С помощью директивы LABEL удобно организовывать доступ к одним и тем же данным, как к байтам, так и к словам, определив перед данными две метки с разными типами.

метка equ выражение

Директива EQU присваивает метке значение, которое определяется как результат целочисленного выражения в правой части. Результатом этого выражения может быть целое число, адрес или любая строка символов:

```
truth           equ     1
message1        equ     'Try again$'
var2            equ     4[si]
        cmp     ax, truth       ; cmp ax,1
        db      message1        ; db 'Try again$'
        mov     ax, var2        ; mov ax, 4[si]
```

Директива EQU чаще всего используется с целью введения параметров, общих для всей программы, аналогично команде #define препроцессора языка С.

метка =. выражение

Директива = эквивалентна EQU, но определяемая ею метка может принимать только целочисленные значения. Кроме того, метка, указанная этой директивой, может быть переопределена.

Каждый ассемблер предлагает целый набор специальных предопределенных меток - это может быть текущая дата (@date или ??date), тип процессора (@cpu) или имя того или иного сегмента программы, но единственная предопределенная метка, поддерживаемая всеми рассматриваемыми нами ассемблерами, — $. Она всегда соответствует текущему адресу. Например, команда

```
jmp     $
```

выполняет безусловный переход на саму себя, так что создается вечный цикл из одной команды.

3.2. Директивы распределения памяти

3.2.1. Псевдокоманды определения переменных

Псевдокоманда - это директива ассемблера, которая приводит к включению данных или кода в программу, хотя сама никакой команде процессора не соответствует. Псевдокоманды определения переменных указывают ассемблеру, что в соответствующем месте программы располагается переменная, устанавливают ее тип (байт, слово, вещественное число и т. д.), задают начальное значение и ставят в соответствие переменной метку, которая будет использоваться для обращения к этим данным. Псевдокоманды определения данных записываются в общем виде следующим образом:

```
имя_переменной    d*        значение
```

где D* — одна из нижеприведенных псевдокоманд:

DB - определить байт;
DW - определить слово (2 байта);
DD - определить двойное слово (4 байта);
DF - определить 6 байт (адрес в формате 16-битный селектор: 32-битное смещение);
DQ - определить учетверенное слово (8 байт);
DT - определить 10 байт (80-битные типы данных, используемые FPU).

Поле значения может содержать одно или несколько чисел, строк символов (взятых в одиночные или двойные кавычки), операторов ? и DUP, разделенных запятыми. Все установленные таким образом данные окажутся в выходном файле, а имя переменной будет соответствовать адресу первого из указанных Значений. Например, набор директив

```
text_string     db      'Hello world!'
number          dw      7
table           db      1,2,3,4,5,6,7,8,9,0Ah,0Bh,0Ch,0Dh,0Eh,0Fh
float_number    dd      3.5e7
```

заполняет данными 33 байта. Первые 12 байт содержат ASCII-коды символов строки **Hello world!**, и переменная text_string указывает на первую букву в этой строке, так что команда

```
        mov     al,text_string
```

считает в регистр AL число 48h (код латинской буквы **H).** Если вместо точного значения указан знак ?, переменная считается неинициализированной и ее значение на момент запуска программы может оказаться любым. Если нужно заполнить участок памяти повторяющимися данными, используется специальный оператор DUP, имеющий формат счетчик DUP (значение). Например, вот такое определение:

```
table_512w      dw      512 dup(?)
```

создает массив из 512 неинициализированных слов, на первое из которых указывает переменная table_512w. В качестве аргумента в операторе DUP могут выступать несколько значений, разделенных запятыми, и даже дополнительные вложенные операторы DUP.

3.2.2. *Структуры*

Директива STRUC позволяет определить структуру данных аналогично структурам в языках высокого уровня. Последовательность директив

```
имя             struc
                поля
имя             ends
```

где поля - любой набор псевдокоманд определения переменных или структур, устанавливает, но не инициализирует структуру данных. В дальнейшем для ее создания в памяти используют имя структуры как псевдокоманду:

```
метка           имя     <значения>
```

И наконец, для чтения или записи в элемент структуры используется оператор . (точка). Например:

```
point           struc                           ; Определение структуры.
x               dw      0                       ; Три слова со значениями
y               dw      0                       ; по умолчанию 0,0,0
z               dw      0
color           db      3 dup(?)                ; и три байта.
point           ends

cur_point       point   <1,1,1,255,255,255>     ; Инициализация.

        mov     ax,cur_point.x                  ; Обращение к слову "x".
```

Если была определена вложенная структура, доступ к ее элементам осуществляется через еще один оператор . (точка).

```
color           struc                          ; Определить структуру color.
red             db      ?
green           db      ?
blue            db      ?
color           ends

point           struc
x               dw      0
y               dw      0
z               dw      0
clr             color   0
point           ends

cur_point       point   <>

        mov     cur_point.clr.red,al    ; Обращение к красной компоненте
                                        ; цвета точки cur_point.
```

3.3. Организация программы

3.3.1. Сегменты

Каждая программа, написанная на любом языке программирования, состоит из одного или нескольких сегментов. Обычно область памяти, в которой находятся команды, называют *сегментом кода,* область памяти с данными - *сегментом данных* и область памяти, отведенную под стек, - *сегментом стека.* Разумеется, ассемблер позволяет изменять устройство программы как угодно - помещать данные в сегмент кода, разносить код на множество сегментов, помещать стек в один сегмент с данными или вообще использовать один сегмент для всего.

Сегмент программы описывается директивами SEGMENT и ENDS.

```
имя_сегмента    segment readonly выравн. тип разряд 'класс'
                ...
имя_сегмента    ends
```

Имя сегмента - метка, которая будет использоваться для получения сегментного адреса, а также для комбинирования сегментов в группы.

Все пять операндов директивы SEGMENT необязательны.

READONLY. Если этот операнд присутствует, MASM выдаст сообщение об ошибке на все команды, выполняющие запись в данный сегмент. Другие ассемблеры этот операнд игнорируют.

Выравнивание. Указывает ассемблеру и компоновщику, с какого адреса может начинаться сегмент. Значения этого операнда:

- BYTE - с любого адреса;
- WORD - с четного адреса;
- DWORD - с адреса, кратного 4;
- PARA - с адреса, кратного 16 (граница параграфа);
- PAGE - с адреса, кратного 256.

По умолчанию используется выравнивание по границе параграфа.

Тип. Выбирает один из возможных типов комбинирования сегментов:

- тип PUBLIC (иногда используется синоним MEMORY) означает, что все такие сегменты с одинаковым именем, но разными классами будут объединены в один;
- тип STACK - то же самое, что и PUBLIC, но должен использоваться для сегментов стека, потому что при загрузке программы сегмент, полученный объединением всех сегментов типа STACK, будет использоваться как стек;
- сегменты типа COMMON с одинаковым именем также объединяются в один, но не последовательно, а по одному и тому же адресу, следовательно, длина суммарного сегмента будет равна не сумме длин объединяемых сегментов, как в случае PUBLIC и STACK, а длине максимального. Таким способом иногда можно формировать оверлейные программы;
- тип AT - выражение указывает, что сегмент должен располагаться по фиксированному абсолютному адресу в памяти. Результат выражения, использующегося в качестве операнда для AT, равен этому адресу, деленному на 16. Например: segment at 40h - сегмент, начинающийся по абсолютному адресу 0400h. Такие сегменты обычно содержат только метки, указывающие на области памяти, которые могут потребоваться программе;
- PRIVATE (значение по умолчанию) - сегмент такого типа не объединяется с другими сегментами.

Разрядность. Этот операнд может принимать значения USE16 и USE32. Размер сегмента, описанного как USE16, не может превышать 64 Кб, и все команды и адреса в этом сегменте считаются 16-битными. В этих сегментах все равно можно применять команды, использующие 32-битные регистры или ссылающиеся на данные в 32-битных сегментах, но они будут использовать префикс изменения разрядности операнда или адреса и окажутся длиннее и медленнее. Сегменты USE32 могут занимать до 4 Гб, и все команды и адреса в них по умолчанию 32-битные. Если разрядность сегмента не указана, по умолчанию используется USE16 при условии, что перед .MODEL не применялась директива задания допустимого набора команд .386 или старше.

Класс сегмента - это любая метка, взятая в одинарные кавычки. Все сегменты с одинаковым классом, даже сегменты типа PRIVATE, будут расположены в исполняемом файле непосредственно друг за другом.

Для обращения к любому сегменту следует сначала загрузить его сегментный адрес (или селектор в защищенном режиме) в какой-нибудь сегментный регистр. Если в программе определено много сегментов, удобно объединить несколько сегментов в группу, адресуемую с помощью одного сегментного регистра:

```
имя_группы          group          имя_сегмента...
```

Операнды этой директивы - список имен сегментов (или выражений, использующих оператор SEG), которые объединяются в группу. Имя группы теперь можно применять вместо имен сегментов для получения сегментного адреса и для директивы ASSUME.

```
assume          регистр:связь,...
```

Директива ASSUME указывает ассемблеру, с каким сегментом или группой сегментов связан тот или иной сегментный регистр. В качестве операнда «связь» могут использоваться имена сегментов, имена групп, выражения с оператором SEG или слово «NOTHING», означающее отмену действия предыдущей ASSUME для данного регистра. Эта директива не изменяет значений сегментных регистров, а только позволяет ассемблеру проверять допустимость ссылок и самостоятельно вставлять при необходимости префиксы переопределения сегментов.

Перечисленные директивы удобны для создания больших программ на ассемблере, состоящих из разнообразных модулей и содержащих множество сегментов. В повседневном программировании обычно используется ограниченный набор простых вариантов организации программы, известных как модели памяти.

3.3.2. *Модели памяти и упрощенные директивы определения сегментов*

Модели памяти задаются директивой .MODEL

```
.model  модель,язык,модификатор
```

где модель - одно из следующих слов:

- TINY - код, данные и стек размещаются в одном и том же сегменте размером до 64 Кб. Эта модель памяти чаще всего используется при написании на ассемблере небольших программ;
- SMALL - код размещается в одном сегменте, а данные и стек - в другом (для их описания могут применяться разные сегменты, но объединенные в одну группу). Эту модель памяти также удобно использовать для создания программ на ассемблере;
- COMPACT - код размещается в одном сегменте, а для хранения данных могут использоваться несколько сегментов, так что для обращения к данным требуется указывать сегмент и смещение (данные дальнего типа);
- MEDIUM - код размещается в нескольких сегментах, а все данные - в одном, поэтому для доступа к данным используется только смещение, а вызовы подпрограмм применяют команды дальнего вызова процедуры;
- LARGE и HUGE — и код, и данные могут занимать несколько сегментов;
- FLAT - то же, что и TINY, но используются 32-битные сегменты, так что максимальный размер сегмента, содержащего и данные, и код, и стек, - 4 Мб.

Язык - необязательный операнд, принимающий значения C, PASCAL, BASIC, FORTRAN, SYSCALL и STDCALL. Если он указан, подразумевается, что Процедуры рассчитаны на вызов из программ на соответствующем языке высокого уровня, следовательно, если указан язык C, все имена ассемблерных процедур, объявленных как PUBLIC, будут изменены так, чтобы начинаться с символа подчеркивания, как это принято в C.

Модификатор - необязательный операнд, принимающий значения NEARSTACK (по умолчанию) или FARSTACK. Во втором случае сегмент стека не будет объединяться в одну группу с сегментами данных.

После того как модель памяти установлена, вступают в силу упрощенные директивы определения сегментов, объединяющие действия директив SEGMENT и ASSUME, Кроме того, сегменты, объявленные упрощенными директивами, не требуется закрывать директивой ENDS - они закрываются автоматически, как только ассемблер обнаруживает новую директиву определения сегмента или конец программы.

Директива .CODE описывает основной сегмент кода

```
.code имя_сегмента
```

эквивалентно

```
_TEXT          segment  word    public   'CODE'
```

для моделей TINY, SMALL и COMPACT и

```
name_TEXT      segment  word    public   'CODE'
```

для моделей MEDIUM, HUGE и LARGE (name - имя модуля, в котором описан данный сегмент). В этих моделях директива .CODE также допускает необязательный операнд - имя определяемого сегмента, но все сегменты кода, описанные так в одном и том же модуле, объединяются в один сегмент с именем NAME_TEXT.

```
.stack  размер
```

Директива .STACK описывает сегмент стека и эквивалентна директиве

```
STACK          segment  para    public   'stack'
```

Необязательный параметр указывает размер стека. По умолчанию он равен 1 Кб.

```
.data
```

Описывает обычный сегмент данных и соответствует директиве

```
_DATA          segment  word    public   'DATA'
```

```
.data?
```

Описывает сегмент неинициализированных данных:

```
_BSS           segment  word    public   'BSS'
```

Этот сегмент обычно не включается в программу, а располагается за концом памяти, так что все описанные в нем переменные на момент загрузки программы имеют неопределенные значения.

```
.const
```

Описывает сегмент неизменяемых данных:

```
CONST          segment  word    public   'CONST'
```

В некоторых операционных системах этот сегмент будет загружен так, что попытка записи в него может привести к ошибке.

```
.fardata имя_сегмента
```

Сегмент дальних данных:

```
имя_сегмента    segment  para    private  'FAR_DATA'
```

Доступ к данным, описанным в этом сегменте, потребует загрузки сегментного регистра. Если не указан операнд, в качестве имени сегмента используется FAR_DATA.

```
.fardata? имя_сегмента
```

Сегмент дальних неинициализированных данных:

```
имя_сегмента    segment  para    private  'FAR_BSS'
```

Как и в случае с FARDATA, доступ к данным из этого сегмента потребует загрузки сегментного регистра. Если имя сегмента не указано, используется FAR_BSS.

Во всех моделях памяти сегменты, представленные директивами .DATA, .DATA?, .CONST, .FARDATA и .FARDATA?, а также сегмент, описанный директивой .STACK, если не был указан модификатор FARSTACK, и сегмент .CODE в модели TINY автоматически объединяются в группу с именем FLAT - для модели памяти FLAT или DGROUP - для всех остальных моделей. При этом сегментный регистр DS (и SS, если не было FARSTACK, и CS в модели TINY) настраивается на всю эту группу, как если бы была выполнена команда ASSUME.

3.3.3. Порядок загрузки сегментов

Обычно сегменты загружаются в память в том порядке, в котором они описываются в тексте программы, причем, если несколько сегментов объединяются в один, порядок определяется по началу первого из объединяемых Сегментов. Этот порядок можно изменить с помощью одной из специальных директив.

```
.alpha
```

Эта директива устанавливает алфавитный порядок загрузки сегментов.

```
.dosseg                 ; для MASM и WASM
или
dosseg                  ; для MASM и TASM
```

Устанавливает порядок загрузки сегментов, существующий в MS DOS и часто требуемый для взаимодействия программ на ассемблере с программами на языках высокого уровня. DOSSEG устанавливает следующий порядок загрузки сегментов:

1. Все сегменты класса 'CODE'.
2. Все сегменты, не принадлежащие группе DGROUP и классу 'CODE'.
3. Группа сегментов DGROUP:
   - все сегменты класса 'BEGDATA';
   - все сегменты, кроме классов 'BEGDATA', 'BSS' и 'STACK';
   - все сегменты класса 'BSS';
   - все сегменты класса 'STACK'.

```
.seq
```

Устанавливает загрузку сегментов в том порядке, в котором они описаны в тексте программы. Этот режим устанавливается по умолчанию, так что директива .SEQ просто отменяет действие .ALPHA или .DOSSEG.

Знание порядка загрузки сегментов необходимо, например, для вычисления длины программы или адреса ее конца. Для этого надо знать, какой сегмент будет загружен последним, и смещение последнего байта в нем.

3.3.4. *Процедуры*

Процедурой в ассемблере является все то, что в других языках называют подпрограммами, функциями, процедурами и т. д. Ассемблер не накладывает на процедуры никаких ограничений - на любой адрес программы можно передать управление командой CALL, и оно вернется к вызвавшей процедуре, как только встретится команда RET. Такая свобода выражения легко может приводить к трудночитаемым программам, и в язык ассемблера были включены директивы логического оформления процедур.

```
метка           proc    язык тип USES регистры    ; TASM
или
метка           proc    тип язык USES регистры    ; MASM/WASM
...
                ret
метка           endp
```

Все операнды PROC необязательны.

Тип может принимать значения NEAR и FAR, и если он указан, все команды RET в теле процедуры будут заменены соответственно на RETN и RETF. По умолчанию подразумевается, что процедура имеет тип NEAR в моделях памяти TINY, SMALL и COMPACT.

Операнд «язык» действует аналогично такому же операнду директивы .MODEL, определяя взаимодействие процедуры с языками высокого уровня. В некоторых ассемблерах директива PROC позволяет также считать параметры, передаваемые вызывающей программой. В этом случае указание языка необходимо, так как различные языки высокого уровня используют разные способы передачи параметров.

USES - список регистров, значения которых изменяет процедура. Ассемблер помещает в начало процедуры набор команд PUSH, а перед командой RET - набор команд POP, так что значения перечисленных регистров будут восстановлены.

3.3.5. *Конец программы*

```
end     start_label
```

Этой директивой завершается любая программа на ассемблере. В роли необязательного операнда здесь выступает метка (или выражение), определяющая адрес, с которого начинается выполнение программы. Если программа состоит из нескольких модулей, только один файл может содержать начальный адрес, так же как в C только один файл может содержать функцию main().

3.5.5. *Директивы задания набора допустимых команд*

По умолчанию ассемблеры используют набор команд процессора 8086 и выдают сообщения об ошибках, если выбирается команда, которую этот процессор не поддерживал. Для того чтобы ассемблер разрешил использование команд, появившихся в более новых процессорах, и команд расширений, предлагаются следующие директивы:

- .8086 - используется по умолчанию. Разрешены только команды 8086;
- .186 - разрешены команды 80186;
- .286 и .286c - разрешены непривилегированные команды 80286;
- .286p - разрешены все команды 80286;
- .386 и .386c - разрешены непривилегированные команды 80386;
- .386p - разрешены все команды 80386;
- .486 и .486c - разрешены непривилегированные команды 80486;
- .486p - разрешены все команды 80486;
- .586 и .586c — разрешены непривилегированные команды P5 (Pentium);
- .586p - разрешены все команды P5 (Pentium);
- .686 - разрешены непривилегированные команды P6 (Pentium Pro, Pentium II);
- .686p - разрешены все команды P6 (Pentium Pro, Pentium II);
- .8087 - разрешены команды NPX 8087;
- .287 - разрешены команды NPX 80287;
- .387 - разрешены команды NPX 80387;
- .487 - разрешены команды FPU 80486;
- .587 - разрешены команды FPU 80586;
- .MMX - разрешены команды IA MM;
- .K3D - разрешены команды AMD 3D.

Не все ассемблеры поддерживают каждую директиву, например MASM и WASM не поддерживают .487 и .587, так как их действие не отличается от .387. Естественно, ассемблеры, вышедшие до появления последних процессоров и расширений, не в состоянии выполнять соответствующие им команды.

Если присутствует директива .386 или выше, ассемблер WASM всегда определяет все сегменты как 32-битные при условии, что не указан явно операнд USE16. MASM и TASM действуют так же, только если директива задания набора команд указана перед директивой .model.

3.3.7. *Директивы управления программным счетчиком*

Программный счетчик - внутренняя переменная ассемблера, равная смещению текущей команды или данных относительно начала сегмента. Для преобразования меток в адреса используется именно значение этого счетчика. Значением счетчика можно управлять с помощью следующих директив.

```
org     выражение
```

Устанавливает значение программного счетчика. Директива ORG с операндом 100h обязательно используется при написании файлов типа COM, которые загружаются в память после блока параметров размером 100h.

```
even
```

Директива EVEN делает текущее значение счетчика кратным двум, вставляя команду NOP, если оно было нечетным. Это увеличивает скорость работы программы, так как для доступа к слову, начинающемуся с нечетного адреса, процессор должен считать два слова из памяти. Если при описании сегмента не использовалось выравнивание типа BYTE, счетчик в начале сегмента всегда четный.

```
align значение
```

Округляет значение программного счетчика до кратного указанному значению. Оно. может быть любым четным числом. Если счетчик некратен указанному числу, эта директива вставляет необходимое количество команд NOP.

3.3.8. Глобальные объявления

```
public  язык метка...     ; Для TASM и MASM.
   или
public  метка язык...     ; Для WASM.
```

Метка, объявленная директивой PUBLIC, становится доступной для других модулей программы. Так, можно объявлять имена процедур, переменные и константы, определенные директивой EQU. Необязательный операнд языка (C, PASCAL, BASIC, FORTRAN, SYSCALL или STDCALL) указывает, что метка будет вызываться из модуля, написанного на соответствующем языке, и при необходимости изменяет ее (например, добавляет _ перед первым символом метки).

```
comm    расст язык      метка:тип...          ; Для TASM.
comm    язык расст      метка:тип...          ; Для TASM.
comm    расст           метка:тип язык...     ; Для WASM.
```

Директива COMM описывает общую переменную. Такие переменные доступны из всех модулей, и их размещение в программе определяется на этапе компоновки. Обязательные аргументы директивы COMM - метка (собственно имя общей переменной) и тип (BYTE, WORD, DWORD, FWORD, QWORD, TBYTE или имя структуры). Необязательный операнд «расстояние» (NEAR или FAR) указывает, находится ли переменная в группе сегментов DGROUP (ближняя переменная, для доступа достаточно смещения) или вне этих сегментов (дальняя переменная, для доступа потребуется сегментный адрес). Для моделей памяти TINY, SMALL и COMPACT по умолчанию значение этого операнда принимается за NEAR. И наконец, операнд «язык» действует аналогично такому же операнду для PUBLIC.

```
extrn   язык            метка:тип...          ; Для MASM и TASM.
extrn                   метка:тип язык...     ; Для WASM.
```

Описывает метку, определенную в другом модуле (с помощью PUBLIC). Тип (BYTE, WORD, DWORD, FWORD, QWORD, TBYTE, имя структуры, FAR, NEAR, ABS) должен соответствовать типу метки в том модуле, где она была установлена (тип ABS используется для констант из других модулей, определенных директивой EQU). Необязательный операнд языка действует так же, как и для директивы PUBLIC.

```
global          язык    метка:тип...      ; Для MASM и TASM.
global                  метка:тип язык...; Для WASM.
```

Директива GLOBAL действует, как PUBLIC и EXTRN одновременно. Когда указанная метка находится в этом же модуле, она становится доступной для других модулей, как если бы выполнилась директива PUBLIC. Если метка не описана - она считается внешней и выполняется действие, аналогичное действию директивы EXTRN.

3.3.9. *Условное ассемблирование*

В большинстве языков программирования присутствуют средства, позволяющие игнорировать тот или иной участок программы в зависимости от выполнения условий, например: в языке C это осуществляется командами препроцессора #if, #ifdef, #ifndef и т. д. Ассемблер тоже предоставляет такую возможность.

```
if          выражение
...
endif
```

Если значение выражения - ноль (ложь), весь участок программы между IF и ENDIF игнорируется. Директива IF может также сочетаться с ELSE и ELSEIF:

```
if          выражение
...
else
...
endif
```

Если значение выражения - ноль, ассемблируется участок программы от ELSE до ENDIF, в противном случае - от IF до ELSE.

```
if          выражение1
...
elseif      выражение2
...
elseif      выражение3
...
else
...
endif
```

Так, если, например, выражение 2 не равно нулю, будет ассемблироваться участок программы между первой и второй директивой ELSEIF Если все три выражения

равны нулю, ассемблируется фрагмент от ELSE до ENDIF. Данная структура директив может использоваться в частном случае аналогично операторам switch/case языков высокого уровня, если выражения - проверки некоторой константы на равенство.

Кроме общих директив IF и ELSEIF ассемблеры поддерживают набор специальных команд, каждая из которых проверяет специальное условие:

- IF1/ELSEIF1 - если ассемблер выполняет первый проход ассемблирования;
- IF2/ELSEIF2 - если ассемблер выполняет второй проход ассемблирования (часто не работает на современных ассемблерах);
- IFE выражение/ELSEIFE выражение - если выражение равно нулю (ложно);
- IFDEF метка/ELSEIFDEF метка - если метка определена;
- IFNDEF метка/ELSEIFNDEF метка - если метка не определена;
- IFB <аргумент>/ELSEIFB <аргумент> - если значение аргумента - пробел (эти и все следующие директивы используются в макроопределениях для проверки параметров);
- IFNB <аргумент>/ELSEIFNB <аргумент> - если значение аргумента - не пробел (используется в макроопределениях для проверки переданных параметров);
- IFDIF <арг1>,<арг2>/ELSEIFDIF <арг1>,<арг2> - если аргументы отличаются (с различием больших и маленьких букв);
- IFDIFI <арг1>,<арг2>/ELSEIFDIFI <арг1>,<арг2> - если аргументы отличаются (без различия больших и маленьких букв);
- IFIDN <арг1>,<арг2>/ELSEIFIDN <арг1>,<арг2> - если аргументы одинаковы (с различием больших и маленьких букв);
- IFIDNI <арг1>,<арг2>/ELSEIFIDNI <арг1>,<арг2> - если аргументы одинаковы (без различия больших и маленьких букв).

Иногда директивы условного ассемблирования используются для того, чтобы прервать ассемблирование программы, если обнаружилась какая-нибудь ошибка. Для таких случаев предназначены директивы условной генерации ошибок.

```
if      $ gt 65535      ; Если адрес вышел за пределы сегмента.
.err
endif
```

Встретив директиву .ERR, ассемблер прекратит работу с сообщением об ошибке. Аналогично: командам условного ассемблирования существуют модификации команды .ERR:

- .ERR1 - ошибка при первом проходе ассемблирования;
- .ERR2 - ошибка при втором проходе ассемблирования;
- .ERRE выражение - ошибка, если выражение равно нулю (ложно);
- .ERRNZ выражение - ошибка, если выражение не равно нулю (истинно);
- .ERRDEF метка - ошибка, если метка определена;
- .ERRNDEF метка - ошибка, если метка не определена;
- .ERRB <аргумент> - ошибка, если аргумент пуст (эта и все следующие директивы используются в макроопределениях для проверки параметров);

- .ERRNB <аргумент> - ошибка, если аргумент не пуст;
- .ERRDIF <арг1>,<арг2> - ошибка, если аргументы различны;
- .ERRDIFI <арг1>,<арг2> - ошибка, если аргументы отличаются (сравнение не различает большие и маленькие буквы);
- .ERRIDN <арг1>,<арг2> - ошибка, если аргументы совпадают;
- .ERRIDNI <арг1>,<арг2> - ошибка, если аргументы совпадают (сравнение не различает большие и маленькие буквы).

3.4. Выражения

Мы уже упоминали выражения при описании многих директив ассемблера. *Выражение* - это набор чисел, меток или строк, связанных друг с другом операторами. Например; 2 + 2 - выражение, состоящее из двух чисел (2 и 2) и оператора +. Каждое выражение имеет значение, которое определяется как результат действия операторов. Так, значение выражения 2 + 2 - число 4. Все выражения вычисляются в ходе ассемблирования программы, следовательно, в полученном коде используются только значения.

Оператор <> (угловые скобки). Часть выражения, заключенная в угловые скобки, не вычисляется, а применяется как строка символов, например:

```
message1 equ     <foobar>
```

Оператор () (круглые скобки). Часть выражения, заключенная в круглые скобки, вычисляется в первую очередь.

```
        mov     al, 2*(3+4)             ; mov al,14
```

Арифметические операторы: + (плюс), - (минус), * (умножение), / (целочисленное деление), MOD (остаток от деления). Они выполняют соответствующие арифметические действия.

```
        mov     al,90 mod 7             ; mov al,6
```

Кроме того, к арифметическим операторам относится унарный минус - минус, который ставят перед отрицательным числом.

Логические операторы: AND (И), NOT (НЕ), OR (ИЛИ), XOR (исключающее ИЛИ), SHL (сдвиг влево), SHR (сдвиг вправо). Эти операторы выполняют соответствующие логические действия.

```
        mov     ax,1234h AND 4321h      ; mov ax,0220h
```

Операторы сравнения: EQ (равно), GE (больше или равно), GT (больше), LE (меньше или равно), LT (меньше), NE (не равно). Результат действия каждого из этих операторов — единица, если условие выполняется, и ноль - если не выполняется.

```
.errnz  $ gt 65535              ; Если адрес больше 64 Кб - ошибка.
```

Операторы адресации:

- SEG выражение - сегментный адрес;
- OFFSET выражение - смещение;
- THIS тип - текущий адрес (MASM и TASM);

Qтип PTR выражение - переопределение типа;
- LARGE выражение - 32-битное смещение (TASM и WASM);
- SMALL выражение - 16-битное смещение (TASM и WASM);
- SHORT выражение - 8-битное смещение.

SEG и OFFSET возвращают соответствующую часть адреса своего аргумента:

```
mov     dx, offset msg          ; Занести в DX смещение переменной msg.
```

THIS создает операнд, адресом которого является текущее значение счетчика:

```
mov     al, this byte-1         ; Занести в AX последний байт кода
                                ; предыдущей команды,
```

PTR создает аргумент, адресом которого является значение выражения, а тип указан явно:

```
mov     dword ptr [si],0        ; Записать 4 байта нулей по адресу DS:SI.
```

LARGE, SMALL и SHORT используются с командами передачи управления, если возникают двусмысленности при косвенных переходах:

```
        jmp     large dword ptr old_address
; Переменная old_address содержит 32-битное смещение.
        jmp     small dword ptr old_address
; Переменная old_address содержит 16-битный сегментный адрес
; и 16-битное смещение.
        jmp     short short_label       ; Метка short_label находится
; ближе, чем +128/-127 байт от этой команды, так что можно
; использовать короткую форму команды JMP.
```

Другие операторы:

- . (точка) - ссылка на элемент структуры;
- : (двоеточие) - переопределение сегмента;
- a [] (угловые скобки) - косвенная адресация;
- ? - неинициализированное значение;
- число DUP (значение) — повторяющееся значение.

Эти пять операторов описаны ранее, когда говорилось о структурах данных, методах адресации и псевдокомандах определения данных.

LENGTH метка - число элементов данных

```
table     dw      0,1,2,3,4,5,6,7       ; Определить таблицу из 8 слов.
table_count = length table              ; table_count = 8
```

SIZE метка - размер данных

```
table_size = size table                 ; table_size = 16
```

3.5. Макроопределения

Одно из самых мощных языковых средств ассемблера - макроопределения. *Макроопределением* (или макросом) называется участок программы, которому присвоено имя и который ассемблируется всякий раз, когда ассемблер встречает

это имя в тексте программы. Макрос начинается директивой MACRO и заканчивается ENDM. Например: пусть описано макроопределение hex2ascii, переводящее шестнадцатеричное число, находящееся в регистре AL, в ASCII-код соответствующей шестнадцатеричной цифры:

```
hex2ascii       macro
        cmp     al,10
        sbb     al,69h
        das
                endm
```

Сейчас в программе можно использовать слово hex2ascii, как если *бы* это было имя команды, и ассемблер заменит каждое такое слово на три команды, содержащиеся в макроопределении. Разумеется, можно оформить этот же участок кода в виде процедуры и вызывать его командой CALL - если процедура вызывается больше одного раза, этот вариант программы займет меньше места, но вариант с макроопределением станет выполняться быстрее, так как в нем не будет лишних команд CALL и RET. Однако скорость выполнения - не главное преимущество макросов. В отличие от процедур макроопределения могут вызываться с параметрами, следовательно, в зависимости от ситуации, включаемый код будет немного различаться, например:

```
s_mov           macro   register1, register2
        push    register1
        pop     register2
                endm
```

Теперь можно использовать S_MOV вместо команды MOV для того, чтобы скопировать значение из одного сегментного регистра в другой.

Следующее важное средство, использующееся в макроопределениях, - директивы условного ассемблирования. Например: напишем макрос, выполняющий умножение регистра AX на число, причем, если множитель - степень двойки, то умножение будет выполняться более быстрой командой сдвига влево.

```
fast_mul        macro   number
        if      number  eq 2
                shl     ax,1            ; Умножение на 2.
        elseif  number  eq 4
                shl     ax,2            ; Умножение на 4.
        elseif  number  eq 8
                shl     ax,3            ; Умножение на 8.
                ...                     ; Аналогично вплоть до:
        elseif  number  eq 32768
                shl     ax,15           ; Умножение на 2^15.
        else
                fflov   dx,number       ; Умножение на число, не являющееся
                raul    dx              ; степенью двойки.
        endif
                endm
```

Можно, конечно, усложнить этот макрос, применяя особые свойства команды LEA и ее комбинации, сдвиги и сложения, однако в нынешнем виде он чрезмерно громоздкий. Проблема решается с помощью третьего средства, постоянно использующегося в макросах, - блоков повторений.

3.5.1. Блоки повторений

Простейший блок повторений REPT (не поддерживается WASM) выполняет ассемблирование участка программы заданное число раз. Например, если требуется создать массив байтов, проинициализированный значениями от 0 до 0FFh, это можно сделать путем повтора псевдокоманды DB следующим образом:

```
hexnumber       = 0
hextable        label byte              ; Имя массива.
                rept    256             ; Начало блока.
                db      hexnumber       ; Эти две строки ассемблируются
hexnumber       = hexnumber+1           ; 256 раз.
                endm
```

Блоки повторений, так же как макроопределения, могут вызываться с параметрами. Для этого используются директивы IRP и IRPC:

```
irp     параметр,<значение1,значение2...>
...
endm

irpc    параметр,строка
...
endm
```

Блок, описанный директивой IRP, будет вызываться столько раз, сколько значений указано в списке (в угловых скобках), и при каждом повторении будет определена метка с именем *параметр,* равная очередному значению из списка. Например, следующий блок повторений сохранит в стек регистры AX, BX, CX и DX:

```
irp     reg,<ax,bx,cx,dx>
push    reg
endm
```

Директива IRPC (FORC в WASM) описывает блок, который выполняется столько раз, сколько символов содержит указанная строка, и при каждом повторении будет определена метка с именем *параметр,* равная очередному символу из строки. Если строка содержит пробелы или другие символы, отличные от разрешенных для меток, она должна быть заключена в угловые скобки. Например, следующий блок задает строку в памяти, располагая после каждого символа строки атрибут 0Fh (белый символ на черном фоне), так что эту строку впоследствии можно будет скопировать прямо в видеопамять.

```
irpc    character,<строка символов>
db      '&character&',0Fh
endm
```

В этом примере используются амперсанды, чтобы вместо параметра character было подставлено его значение даже внутри кавычек. *Амперсанд* - это; один из макрооператоров - специальных операторов, которые действуют только внутри макроопределений и блоков повторений.

3.5.2. *Макрооператоры*

Макрооператор & (амперсанд) нужен для того, чтобы параметр, переданный в качестве операнда макроопределению или блоку повторений, заменялся значением до обработки строки ассемблером. Так, например, следующий макрос выполнит команду PUSH EAX, если его вызвать как PUSHREG A:

```
pushreg macro   letter
        push    e&letter&x
        endm
```

Иногда можно использовать только один амперсанд - в начале параметра, если не возникает неоднозначностей. Например, если передается номер, а требуется создать набор переменных с именами, оканчивающимися этим номером:

```
        irp     number, <1, 2, 3, 4>
msg&number      db      ?
        endm
```

Макрооператор <> (угловые скобки) действует так, что весь текст, заключенный в эти скобки, рассматривается как текстовая строка, даже если он содержит пробелы или другие разделители. Как мы уже видели, этот макрооператор используется при передаче текстовых строк в качестве параметров для макросов. Другое частое применение угловых скобок - передача списка параметров вложенному макроопределению или блоку повторений.

Макрооператор ! (восклицательный знак) используется аналогично угловым скобкам, но действует только на один следующий символ, так что, если этот символ - запятая или угловая скобка, он все равно будет передан макросу как часть параметра.

Макрооператор % (процент) указывает, что находящийся за ним текст является выражением и должен быть вычислен. Обычно это требуется для того, чтобы передавать в качестве параметра в макрос не само выражение, а его результат.

Макрооператор ;; (две точки с запятой) - начало макрокомментария. В отличие от обычных комментариев текст макрокомментария не попадает в листинг и в текст программы при подстановке макроса. Это сэкономит память при ассемблировании программы с большим количеством макроопределений.

3.5.3. Другие директивы, используемые в макроопределениях

Директива EXITM (не поддерживается WASM) выполняет преждевременный выход из макроопределения или блока повторений. Например, следующее макроопределение не выполнит никаких действий, то есть не будет расширено в команды процессора, если параметр не указан:

```
pushreg  macro    reg
         ifb      <reg>
         exitm
         endif
         push     reg
         endm
```

LOCAL метка... - перечисляет метки, которые будут применяться внутри макроопределения, чтобы не возникало ошибки «метка уже определена» при использовании макроса более одного раза или если та же метка присутствует в основном тексте программы (в WASM директива LOCAL позволяет использовать макрос с метками несколько раз, но не разрешает применять метку с тем же именем в программе). Операнд для LOCAL - метка или список меток, которые будут использоваться в макросе.

PURGE имя_макроса - отменяет определенный ранее макрос (не поддерживается WASM). Эта директива часто применяется сразу после INCLUDE, включившей в текст программы файл с большим количеством готовых макроопределений.

3.6. Другие директивы

3.6.1. Управление файлами

INCLUDE имя_файла - директива, вставляющая в текст программы текст файла аналогично команде препроцессора C #include. Обычно используется для включения файлов, содержащих определения констант, структур и макросов.

INCLUDELIB имя_файла - директива, указывающая компоновщику имя дополнительной библиотеки или объектного файла, который потребуется при составлении данной программы. Например, если используются вызовы процедур или обращение к данным, определенным в других модулях. Использование этой директивы позволяет не указывать имена дополнительных библиотек при вызове компоновщика.

3.6.2. Управление листингом

Обычно ассемблеры, помимо создания объектного файла, предоставляют возможность создания листинга программы (TASM /L - для TASM, ml /F1 — для MASM). *Листинг* - это файл, содержащий текст ассемблерной программы, код каждой ассемблированной команды, список определенных меток, перекрестных ссылок, сегментов и групп. Формат файла листинга отличается для разных ассемблеров, и директивы управления форматом этого файла также сильно различаются, но несколько наиболее общих директив все-таки поддерживаются всеми тремя ассемблерами, рассмотренными в этой книге.

- TITLE текст - определяет заголовок листинга. Заголовок появляется в начале каждой страницы;
- SUBTTL текст - определяет подзаголовок листинга. Подзаголовок появляется на следующей строке после заголовка;

- PAGE высота,ширина - устанавливает размеры страниц листинга (высота 10-255, ширина 59-255). Директива PAGE без аргументов начинает новую страницу, директива PAGE + начинает новую секцию, и нумерация страниц ведется с самого начала;
- NAME текст - определяет имя модуля программы. Если NAME не указан, в качестве имени используются первые 6 символов из TITLE; если нет ни NAME, ни TITLE, за имя берется название файла;
- .XLIST – отменить выдачу листинга;
- .LIST - разрешить выдачу листинга;
- .SALL - запретить листинг макроопределений;
- .SFCOND - запретить листинг неассемблированных условных блоков;
- .LFCOND - разрешить листинг неассемблированных условных блоков;
- .TFCOND - изменить режим листинга условных блоков на противоположный;
- .CREF - разрешить листинг перекрестных ссылок;
- .XCREF - запретить листинг перекрестных ссылок.

3.5.3. *Комментарии*

Кроме обычных комментариев, начинающихся с символа **;** (точка с запятой) и заканчивающихся в конце строки, возможны большие блоки комментариев, описываемых специальной директивой COMMENT.

```
comment @
любой текст
@
```

Операнд для COMMENT — любой символ, который будет считаться концом комментария. Весь участок текста, вплоть до следующего появления этого символа, ассемблером полностью игнорируется.

Глава 4. Основы программирования для MS DOS

Программа, написанная на ассемблере, так же как и программа, написанная на любом другом языке программирования, выполняется не сама по себе, а при помощи операционной системы. Операционная система выделяет области памяти для программы, загружает ее, передаёт ее управление и обеспечивает взаимодействие программы с устройствами ввода-вывода, файловыми системами и другими программами (разумеется, кроме тех случаев, когда эта программа сама является операционной системой или ее частью). Способы взаимодействия программы с внешним миром различны для разных операционных систем, поэтому программа, написанная для Windows, не будет работать в DOS, а программа для Linux - в Solaris/x86, хотя все указанные системы могут работать на одном и том же компьютере.

Самая простая и распространенная операционная система для компьютеров, основанных на процессорах Intel, - DOS (дисковая операционная система). Она распространяется как сама по себе несколькими производителями - Microsoft (MS DOS), IBM (PC-DOS), Novell (Novell DOS), Caldera (Open DOS) и др., так и в виде части систем Microsoft Windows 95 и старше. DOS предоставляет программам полную свободу действий, никак не ограничивая доступ к памяти и внешним устройствам, позволяя им самим управлять процессором и распределением памяти. По этой причине система лучше всего подходит для того, чтобы основательно познакомиться с устройством компьютера и возможностями программы на ассемблере, но которые часто скрываются компиляторами с языков высокого уровня и более совершенными операционными системами.

Итак, чтобы программа выполнилась любой ОС, она должна быть скомпилирована в исполняемый файл. Основные два формата исполняемых файлов в DOS - COM и EXE. Файлы типа COM содержат только скомпилированный код без какой-либо дополнительной информации о программе. Весь код, данные и стек такой программы располагаются в одном сегменте и не могут превышать 64 Кб. Файлы типа EXE содержат заголовок, где описывается размер файла, требуемый объем памяти, список команд в программе, использующих абсолютные адреса, которые зависят от расположения программы в памяти, и т. д. EXE-файл может иметь любой размер. Формат EXE также используется для исполняемых файлов в различных версиях DOS-расширителей и Windows, но с большими изменениями.

Несмотря на то что файлам типа COM принято давать расширение .com, а файлам типа EXE - .exe, DOS не использует расширения для определения типа файла. Первые два байта заголовка EXE-файла - символы «MZ»

или «ZM», и если файл начинается с них и длиннее некоторого порогового значения, отличающегося в разных версиях DOS, он загружается как EXE, если нет - как COM.

Кроме обычных исполняемых программ DOS может загружать драйверы устройств - специальные программы, используемые для упрощения доступа к внешним устройствам. Например, драйвер устройства LPT, входящий в IO.SYS, позволяет посылать тексты на печать из DOS простым копированием файла в LPT, а драйвер RAMDISK.SYS разрешает выделить область памяти и обращаться к ней, как к диску. Написание драйверов значительно сложнее, чем написание обычных программ (см. далее).

4.1. Программа типа COM

Традиционно первая программа для освоения нового языка программирования - программа, выводящая на экран текст **Hello world!**. Не будет исключением и эта книга, поскольку такая программа всегда являлась удобной отправной точкой для дальнейшего освоения языка.

Итак, наберите в любом текстовом редакторе, который может записывать файлы как обычный текст (например: EDIT.COM в DOS, встроенный редактор в Norton Commander или аналогичной программе, NOTEPAD в Windows), следующий текст:

```
; hello-1.asm
; Выводит на экран сообщение "Hello World!" и завершается.

        .model  tiny                    ; Модель памяти, используемая для COM.
        .code                           ; Начало сегмента кода.
        org     100h                    ; Начальное значение счетчика - 100h.
start:  mov     ah,9                    ; Номер функции DOS - в AH.
        mov     dx,offset message       ; Адрес строки - в DX.
        int     21h                     ; Вызов системной функции DOS.
        ret                             ; Завершение COM-программы.
message         db      "Hello World!";0Dh,0Ah,'$' ; Строка для вывода.
        end     start                   ; Конец программы.
```

и сохраните его как файл hello-1.asm. Можно также использовать готовый файл с этим именем. (Все программы, рассмотренные в этой книге как примеры, вы можете найти в Internet по адресу: http://www.dmk.ru.)

Для превращения программы в исполняемый файл сначала надо вызвать ассемблер, чтобы скомпилировать ее в объектный файл с именем hello-1.obj, набрав в командной строке следующую команду:

Для TASM:

```
tasm hello-1.asm
```

Для MASM:

```
ml /c hello-1.asm
```

Для WASM:

```
wasm hello-1.asm
```

С ассемблерными программами также можно работать из интегрированных сред разработки, как обычно поступают с языками высокого уровня. Но в них, как правило, удобнее создавать ассемблерные процедуры, вызываемые из программ на языке, для которого предназначена среда, а создание полноценных программ на ассемблере требует некоторой перенастройки.

Формат объектных файлов, применяемых всеми тремя рассматриваемыми ассемблерами по умолчанию (OMF-формат), совпадает, так что можно пользоваться ассемблером из одного пакета и компоновщиком из другого.

Для TASM:

```
tlink /t /x hello-1.obj
```

Для MASM (команда link должна вызывать 16-битную версию LINK.EXE):

```
link hello-1.obj,,NUL,,,
exe2bin hello-1.exe hello-1.com
```

Для WASM:

```
wlink file hello-1.obj form DOS COM
```

Теперь получился файл HELLO-1.COM размером 23 байта. Если его выполнить, на экране появится строка **Hello World!** и программа завершится.

Рассмотрим исходный текст программы, чтобы понять, как она работает.

Первая строка определяет модель памяти TINY, в которой сегменты кода, данных и стека объединены. Эта модель предназначена для создания файлов типа COM.

Директива .CODE начинает сегмент кода, который в нашем случае также должен содержать и данные.

ORG 100h устанавливает значение программного счетчика в 100h, потому что при загрузке COM-файла в память DOS занимает первые 256 байт (lOOh) блоком данных PSP и располагает код программы только после этого блока. Все программы, которые компилируются в файлы типа COM, должны начинаться с этой директивы.

Метка START располагается перед первой командой в программе и будет использоваться в директиве END, чтобы указать, с какой команды начинается программа.

Команда MOV AH,9 помещает число 9 в регистр AH - номер функции DOS «вывод строки».

Команда MOV DX,OFFSET MESSAGE помещает в регистр DX смещение метки MESSAGE относительно начала сегмента данных, который в нашем случае совпадает с сегментом кода.

Команда INT 21h вызывает системную функцию DOS. Эта команда - основное средство взаимодействия программ с операционной системой. В нашем примере вызывается функция DOS номер 9 - вывести строку на экран. Эта функция выводит строку от начала, адрес которого задается в регистрах DS:DX, до первого встреченного символа $. При запуске COM-файла регистр DS автоматически загружается сегментным адресом программы, а регистр DX был подготовлен предыдущей командой.

Команда RET используется обычно для возвращения из процедуры. DOS вызывает COM-программы так, что команда RET корректно завершает программу.

DOS при вызове COM-файла помещает в стек сегментный адрес программы и ноль, так что RET передает управление на нулевой адрес текущего сегмента, то есть на первый байт PSP. Там находится код команды INT 20h, которая и используется для возвращения управления в DOS. Можно сразу заканчивать программу командой INT 20h, хотя это длиннее на 1 байт.

Следующая строка программы HELLO-1.ASM определяет строку данных, содержащую текст **Hello World!**, управляющий символ ASCII **возврат каретки** с кодом 0Dh, управляющий символ ASCII **перевод строки** с кодом 0Ah и символ $, завершающий строку. Эти два управляющих символа переводят курсор на первую позицию следующей строки точно так же, как в строках на языке С действует последовательность \n.

И наконец, директива END завершает программу, одновременно указывая, с какой метки должно начинаться ее выполнение.

4.2. Программа типа EXE

EXE-программы немного сложнее в исполнении, но для них отсутствует ограничение размера в 64 Кб, так что все достаточно большие программы используют именно этот формат. Конечно, ассемблер позволяет уместить и в 64 Кб весьма сложные и большие алгоритмы, а все данные хранить в отдельных файлах, но ограничение размера все равно очень серьезно, и даже чисто ассемблерные программы могут с ним сталкиваться.

```
; hello-2.asm
; Выводит на экран сообщение "Hello World!" и завершается.
        .model  small                   ; Модель памяти, используемая для EXE.
        .stack  100h                    ; Сегмент стека размером в 256 байт.
        .code
start:  mov     ax,DGROUP               ; Сегментный адрес строки message
        mov     ds,ax                   ; помещается в DS.
        flov    dx,offset message
        mov     ah,9
        int     21h                     ; Функция DOS "вывод строки".
        mov     ax,4C00h
        int     21h                     ; Функция DOS "завершить программу".
        .data
message         db      "Hello World!",0Dh,0Ah,'$'
        end     start
```

В этом примере определяются три сегмента - сегмент стека директивой .STACK размером в 256 байт; сегмент кода, начинающийся с директивы .CODE; и сегмент данных, начинающийся с .DATA и включающий строку. При запуске EXE-программы регистр DS уже не содержит адреса сегмента со строкой message (он указывает на сегмент с блоком данных PSP), а для вызова используемой функции DOS этот регистр должен иметь сегментный адрес строки. Команда MOV AX,DGROUP загружает в AX сегментный адрес группы сегментов данных DGROUP, а MOV DS,AX

копирует его в DS. Для ассемблеров MASM и TASM вместо DGROUP можно использовать предопределенную метку **@data**, но единственная модель памяти, в которой группа сегментов данных называется иначе, - FLAT (ей мы пока пользоваться не будем). И наконец, программы типа EXE должны завершаться системным вызовом DOS 4Ch: в регистр AH помещается значение 4Ch, в регистр AL - код возврата (в данном примере код возврата 0 и регистры AH и AL загружаются одной командой MOVAX,4COOh), после чего вызывается прерывание 21h.

Компиляция hello-2.asm.

Для TASM:

```
tasm hello-2.asm
tlink /x hello-2.obj
```

Размер получаемого файла hello-2.exe - 559 байт.

Для MASM:

```
ml /c hello-2.asm
link hello-2.obj
```

Размер получаемого файла hello-2.exe - 545 байт.

Для WASM:

```
wasm hello-2.asm
wlink file hello-2.obj form DOS
```

Размер получаемого файла hello-2.exe - 81 байт.

Расхождения в размерах файлов вызваны различными соглашениями о выравнивании сегментов программы по умолчанию. Почти все примеры программ для DOS в этой книге рассчитаны на компиляцию в COM-файлы, так как идеология работы с памятью в них во многом совпадает с идеологией, используемой при программировании под расширители DOS, DPMI и Windows.

4.3. Вывод на экран в текстовом режиме

4.3.1. Средства DOS

На примере первой ассемблерной программы мы уже познакомились с одним из способов вывода текста на экран - вызовом функции DOS 09h. Это далеко не единственный способ вывода текста - DOS предоставляет для этого несколько функций.

Функция DOS 02h: Записать символ в STDOUT с проверкой на **Ctrl-Break**

Вход: AH - 02h
DL = ASCII-код символа

Выход: Никакого, согласно документации, но на самом деле:
AL = код последнего записанного символа (равен DL, кроме случая, когда DL = 09h (табуляция), тогда в AL возвращается 20h).

Данная функция обрабатывает некоторые управляющие символы: при выводе символа BEL (07h) появляется звуковой сигнал, посредством BS (08h) курсор

перемещается влево на одну позицию, символ HT (09h) используется для замены на несколько пробелов, символ LF (OAh) - для перевода курсора на одну позицию вниз, а CR (0Dh) - для перехода на начало текущей строки.

Если в ходе работы этой функции была нажата комбинация клавиш **Ctrl-Break,** вызывается прерывание 23h, которое по умолчанию осуществляет выход изпрограммы.

Например, напишем программу, выводящую на экран все ASCII-символы, 16 строк по 16 символов в строке.

```
; dosout1.asm
; Выводит на экран все ASCII-символы
;
        .model  tiny
        .code
        org     100h            ; Начало COM-файла.
start:
        mov     cx,256          ; Вывести 256 символов.
        mov     dl,0            ; Первый символ - с кодом 00.
        mov     ah,2            ; Номер функции DOS "вывод символа".
cloop:  int     21h             ; Вызов DOS.
        inc     dl              ; Увеличение DL на 1 - следующий символ,.
        test    dl,0Fh          ; Если DL не кратен 16,
        jnz     continue_loop   ; продолжить цикл,
        push    dx              ; Иначе: сохранить текущий символ,
        mov     dl,0Dh          ; вывести CR,
        int     21h
        mov     dl,0Ah          ; вывести LF,
        int     21h
        pop     dx              ; восстановить текущий символ,
continue_loop:
        loop    cloop           ; продолжить цикл.

        ret                     ; Завершение COM-файла.
        end     start
```

Это программа типа COM, и компилироваться она должна точно так же, как hello-1.asm из раздела 4.1. Здесь с помощью команды ШОР оформляется цикл, выполняющийся 256 раз (значение регистра CX в начале цикла). Регистр DL содержит код символа, который равен нулю в начале цикла и увеличивается каждый раз на 1 командой INC DL. Если значение DL сразу после увеличения на 1 кратно 16, оно временно сохраняется в стеке и на экран выводятся символы CR и LF, выполняющие переход на начало новой строки. Проверка осуществляется командой TEST DL,0Fh - результат операции AND над DL, и 0Fh = 0, только если младшие четыре бита DL равны нулю, что и соответствует кратности шестнадцати.

Все функции DOS вывода на экран используют устройство STDOUT, стандартный вывод. Это позволяет перенаправлять вывод программы в файл или на стандартный ввод другой программы. Например, если написать в командной строке

```
hello-1.com > hello-Lout
```

то на экране ничего не появится, а в текущей директории появится файл hello-1.out, содержащий строку **Hello World!**. Точно так же, если написать

```
dosout1.com > dosout1.out
```

то в файле dosoutl.out окажутся все символы ASCII, причем символы BEL и BS не будут интерпретироваться и запишутся в файл как есть. Символы CR и LF тоже запишутся как есть, но поскольку они отмечают конец строки, редакторы и просмотрщики текстовых файлов будут разрывать первую строку символов.

Функция DOS 06h: Записать символ в STDOUT без проверки на **Ctrl-Break**
Вход: AH = 06h
DL = ASCII-код символа (кроме 0FFh)
Выход: Никакого, согласно документации, но на самом деле:
AL = код записанного символа (копия DL)

Эта функция не обрабатывает управляющие символы (CR, LF, HT и BS выполняют свои функции при выводе на экран, но сохраняются при перенаправлении вывода в файл) и не проверяет нажатие **Ctrl-Break.** Можно заменить MOV AH,2 командой MOV AH,6 в программе dosoutl.asm и перекомпилировать этот пример, чтобы получить более полную таблицу символов.

Функция DOS 09h: Записать строку в STDOUT с проверкой на **Ctrl-Break**
Вход: AH = 09h
DS:DX = адрес строки, заканчивающейся символом $ (24h)
Выход: Никакого, согласно документации, но на самом деле:
AL = 24h (код последнего символа)

Действие этой функции полностью аналогично действию функции 02h, но выводится не один символ, а целая строка, как в программах hello-1.asm и hello-2.asm.

Функция DOS 40h: Записать в файл или устройство
Вход: AH = 40h
BX = 1 для STDOUT или 2 для STDERR
DS:DX = адрес начала строки
CX = длина строки
Выход: CF = O,
AX = число записанных байтов

Эта функция предназначена для записи в файл, но, если в регистр BX поместить число 1, функция 40h будет выводить данные на STDOUT, а если BX = 2 - на устройство STDERR. Оно всегда выводит данные на экран и не перенаправляется в файлы. На DOS 40h основаны используемые в C функции стандартного вывода - фактически функция C fputs() просто вызывает это прерывание, помещая свой первый аргумент в BX, адрес строки (второй аргумент) - в DS:DX и длину - в CX.

```
; dosout2.asm
;  Выводит на экран строку "Эта функция может вывозить знак $",
; используя вывод в STDERR, так что ее нельзя перенаправить в файл.
```

```
        .model  tiny
        .code
        org     100h                    ; Начало COM-файла.
start:
        mov     ah,40h                  ; Номер функции DOS.
        mov     bx,2                    ; Устройство STDERR.
        mov     dx,offset message       ; DS:DX - адрес строки.
        mov     cx,message_length       ; CX - длина строки.
        int     21h
        ret                             ; Завершение COM-файла.
message         db      "Эта функция может выводить знак $"
message_length = $-message              ; Длина строки = текущий адрес минус
                                        ; адрес начала строки.
        end     start
```

Если скомпилировать эту программу и запустить ее командой

```
dosout2.com > dosout2.out
```

то сообщение появится на экране, а файл dosout2.out окажется пустым.

И наконец, последняя функция DOS вывода на экран - недокументированное прерывание 29h.

INT 29h: Быстрый вывод символа на экран
Вход: AL = ASCII-код символа

В большинстве случаев INT 29h немедленно вызывает функцию BIOS «вывод символа на экран в режиме телетайпа», поэтому никаких преимуществ, кроме экономии байтов при написании как можно более коротких программ, она не имеет.

4.3.2. *Средства BIOS*

Функции DOS вывода на экран позволяют перенаправлять вывод в файл, но не дают возможности вывести текст в любую позицию экрана и не разрешают изменить цвет текста. DOS предполагает, что для более тонкой работы с экраном программы должны использоваться видеофункции BIOS. BIOS (базовая система ввода-вывода) - это набор программ, расположенных в постоянной памяти компьютера, которые выполняют его загрузку сразу после включения и обеспечивают доступ к отдельным устройствам, в частности к видеоадаптеру. Все функции видеосервиса BIOS вызываются через прерывание 10h. Рассмотрим функции, которые могут быть полезны для вывода текстов на экран.

Выбор видеорежима

BIOS позволяет переключать экран в различные текстовые и графические режимы. Режимы отличаются друг от друга разрешением (для графических) и количеством строк и столбцов (для текстовых), а также количеством возможных цветов.

INT 10h, AH = 00h: Установить видеорежим
Вход: AH = 00h
AL = номер режима в младших 7 битах

Вызов этой функции приводит к тому, что экран переводится в выбранный режим. Если старший бит AL не установлен в 1, экран очищается. Номера текстовых режимов - 0, 1, 2, 3 и 7. 0 и 1 - 16-цветные режимы 40x25 (с 25 строками по 40 символов в строке), 2 и 3 - 16-цветные режимы 80x25, 7 - монохромный режим 80x25. Мы не будем пока рассматривать графические режимы, хотя функции вывода текста на экран DOS и BIOS могут работать и в них. Существуют и другие текстовые режимы с более высоким разрешением (80×43, 80×60, 132×50 и т. д.), но их номера для вызова через эту функцию отличаются для разных видеоадаптеров (например, режим 61h - 132x50 для Cirrus 5320 и 132x29 для Genoa 6400). Однако, если видеоадаптер поддерживает стандарт VESA BIOS Extention, в режимы с высоким разрешением можно переключаться, используя функцию 4Fh.

INT 10h, AH = 4Fh, AL = 02h: Установить видеорежим SuperVGA
Вход: AX = 4F02h
BX = номер режима в младших 13 битах

Если бит 15 регистра BX установлен в 1, видеопамять не очищается. Текстовые режимы, которые можно вызвать с использованием этой функции: 80x60 (режим 108h), 132x25 (109h), 132x43 (10Ah), 132x50 (10Bh), 132x60 (10Ch).

Видеорежим, используемый в DOS по умолчанию, - текстовый режим 3.

Управление положением курсора

INT 10h, AH = 02h: Установить положение курсора
Вход: AH = 02h
BH = номер страницы
DH = строка
DL = столбец

С помощью этой функции можно установить курсор в любую позицию экрана, и дальнейший вывод текста будет происходить из этой позиции. Отсчет номера строки и столбца ведется от верхнего левого угла экрана (символ в левой верхней позиции имеет координаты 0, 0). Номера страниц 0-3 (для режимов 2 и 3) и 0-7 (для режимов 0 и 1) соответствуют области памяти, содержимое которой в данный момент отображается на экране. Можно вывести текст в неактивную в настоящий момент страницу, а затем переключиться на нее, чтобы изображение изменилось мгновенно.

INT 10h, AH = 03h: Считать положение и размер курсора
Вход: AH = 03h
BH = номер страницы
Выход: DH, DL = строка и столбец текущей позиции курсора
CH, CL = первая и последняя строки курсора

Возвращает текущее состояние курсора на выбранной странице (каждая страница использует собственный независимый курсор).

Вывод символов на экран

Каждый символ на экране описывается двумя байтами - ASCII-кодом символа и байтом атрибута, указывающим цвет символа и фона, а также является ли символ мигающим.

Атрибут символа

- бит 7: символ мигает (по умолчанию) или фон яркого цвета (если его действие было переопределено видеофункцией 10h);
- биты 6-4: цвет фона;
- бит 3: символ яркого цвета (по умолчанию) или фон мигает (если его действие было переопределено видеофункцией 11h);
- биты 2-0: цвет символа.

Цвета кодируются в битах, как показано в табл. 18.

INT 10h, AH = 08k. Считать символ и атрибут символа в текущей позиции курсора

Вход: AH = 08h
BH = номер страницы
Выход: AH = атрибут символа
AL = ASCII-код символа

INT 10h, AH - 09h: Вывести символ с заданным атрибутом на экран

Вход: AH = 09h
BH = номер страницы
AL = ASCII-код символа
BL = атрибут символа
CX = число повторений символа

С помощью этой функции можно вывести на экран любой символ, включая даже символы CR и LF, которые обычно интерпретируются как конец строки. В графических режимах CX не должен превышать число позиций, оставшееся до правого края экрана.

INT 10h, AH = 0Ah: Вывести символ с текущим атрибутом на экран

Вход: AH = OAh
BH = номер страницы
AL = ASCII-код символа
CX = число повторений символа

Эта функция также выводит любой символ на экран, но в качестве атрибута символа используется атрибут, который имел символ, находившийся ранее в данной позиции.

Таблица 18. Атрибуты символов

Атрибут	Обычный цвет	Яркий цвет
000b	Черный	Темно-серый
001b	Синий	Светло-синий
010b	Зеленый	Светло-зеленый
011b	Голубой	Светло-голубой
100b	Красный	Светло-красный
101b	Пурпурный	Светло-пурпурный
110b	Коричневый	Желтый
111B	Светло-серый	Белый

INT 10h, AH = 0Eh: Вывести символ в режиме телетайпа
Вход: AH = OEh
BH = номер страницы
AL = ASCII-код символа

Символы CR (0Dh), LF (0Ah), BEL (7) интерпретируются как управляющие символы. Если текст при записи выходит за пределы нижней строки, экран прокручивается вверх. В качестве атрибута используется атрибут символа, находившегося в данной позиции.

INT 10h, AH = 13h: Вывести строку символов с заданными атрибутами
Вход: AH - 13h
AL = режим вывода:
бит 0: переместить курсор в конец строки после вывода
бит 1: строка содержит не только символы, но и атрибуты, так что каждый символ описывается двумя байтами: ASCII-код и атрибут
биты 2-7: зарезервированы
CX = длина строки (только число символов)
BL = атрибут, если строка содержит только символы
DH, DL = строка и столбец, начиная с которых будет выводиться строка
ES:BP = адрес начала строки в памяти

Функция 13h выводит на экран строку символов, интерпретируя управляющие символы CR (ODh), LF (OAh), BS (08) и BEL (07). Если строка подготовлена в формате символ,атрибут - гораздо быстрее просто скопировать ее в видеопамять, о чем рассказано в следующей главе.

Воспользуемся теперь функциями BIOS, чтобы усовершенствовать программу DOSOUT1 и вывести на экран все 256 символов, включая даже символы перевода строки. Кроме того, для лучшей читаемости таблицы после каждого символа будет выводиться пробел.

```
; biosout.asm
; Выводит на экран все ASCII-символы без исключения.
;
        .model  tiny
        .code
        org     100h            ; Начало COM-файла.
start:
        mov     ax,0003h
        int     10h             ; Видеорежим 3 (очистка экрана
                                ; и установка курсора в 0, 0).
        mov     dx,0            ; DH и DL будут использоваться
                                ; для хранения положения курсора.
                                ; Начальное положение - 0,0.
        mov     si,256          ; SI будет счетчиком цикла.
        mov     al,0            ; Первый символ - с кодом 00h.
        mov     ah,9            ; Номер видеофункции "вывод символа с атрибутом".
        mov     cx,1            ; Выводится один символ за раз.
        mov     bl,00011111b    ; Атрибут символа - белый на синем.

cloop:
        int     10h             ; Вывести символ на экран
        push    ax              ; Сохранить текущий символ и номер функции.
        mov     ah,2            ; Номер видеофункции 2 -
                                ; изменить положение курсора.
        inc     dl              ; Увеличить текущий столбец на 1.
        int     10h             ; Переместить курсор
        mov     ax,0920h        ; AH = 09, AL = 20h (ASCII-код пробела)
        int     10h             ; Вывести пробел.
        mov     ah,2            ; Номер видеофункции 2.
        inc     dl              ; Увеличить столбец на 1.
        int     10h             ; Переместить курсор.
        pop     ax              ; Восстановить номер функции в ah
                                ; и текущий символ в al.
        inc     al              ; Увеличить AL на 1 - следующий символ.
        test    al,0Fh          ; Если AL не кратен 16,
        jnz     continue_loop
                                ; продолжить цикл.
        push    ax              ; Иначе - сохранить номер функции
                                ; и текущий символ.
        mov     ah,2            ; Номер видеофункции 2.
        inc     dh              ; Увеличить номер строки на 1.
        mov     dl,0            ; Столбец = 0.
        int     10h             ; Установить курсор на начало следующей строки.
        pop     ax              ; Восстановить номер видеофункции
                                ; и текущий символ.

continue_loop:
        dec     si              ; Уменьшить SI на 1.
                                ; Если он не стал нулем - продолжить.
        jnz     cloop           ; CX используется внутри цикла,
```

```
                                ; так что нельзя применить команду LOOP
                                ; для его организации.

        ret                     ; Завершение COM-файла.
        end     start
```

Так как функция 09h выводит символ в позиции курсора, но ,не перемещает сам курсор, это приходится делать каждый раз специально.

Функции BIOS удобны для переключения и настройки видеорежимов, но часто оказывается, что вывод текста на экран гораздо быстрее и проще выполнять обычным копированием изображения в видеопамять.

43.3. Прямая работа с видеопамятью

Все, что изображено на мониторе - и графика, и текст, одновременно присутствует в памяти, встроенной в видеоадаптер. Чтобы изображение появилось на мониторе, оно должно быть записано в память видеоадаптера. Для этой цели отводится специальная область памяти, начинающаяся с абсолютного адреса 0B800h:0000h (для текстовых режимов) и заканчивающаяся OB800h:OFFFFh. Все, что программы пишут в эту область памяти, немедленно пересылается в память видеоадаптера. В текстовых режимах для хранения каждого изображенного символа используются два байта: байт с ASCII-кодом символа и байт с его атрибутом, так что по адресу 0B800h:0000h лежит байт с кодом символа, находящимся в верхнем левом углу экрана; по адресу OB800h:0001h расположен атрибут этого символа; по адресу OB800h:0002h - код второго символа в верхней строке экрана и т. д.

Таким образом, любая программа может вывести текст на экран простой командой пересылки данных, не прибегая ни к каким специальным функциям DOS или BIOS.

```
; dirout.asm
; Выводит на экран все ASCII-символы без исключения,
; используя прямой вывод на экран.
;
        .model  tiny
        .code
        .386                    ; Будет использоваться регистр EAX
                                ; и команда STOSD.
        org     100h            ; Начало COM-файла.
start:
        mov     ax,0003h
        int     10h             ; Видеорежим 3 (очистка экрана).
        cld                     ; Обработка строк в прямом направлении.
                                ; Подготовка данных для вывода на экран:
        mov     eax,1F201F00h   ; первый символ 00 с атрибутом 1Fh,
                                ; затем пробел (20h) с атрибутом 1Fh
        mov     bx,OF20h        ; Пробел с атрибутом OFh.
        mov     cx,255          ; Число символов минус 1.
        mov     di,offset ctable ; ES:DI - начало таблицы.
```

```
cloop:
        stosd                   ; Записать символ и пробел в таблицу ctable.
        inc     al              ; AL содержит следующий символ.

        test    cx,0Fh          ; Если CX не кратен 16,
        jnz     continue_loop   ; продолжить цикл.
        push    cx              ; Иначе: сохранить значение счетчика.
        mov     cx,80-32        ; Число оставшихся до конца строки символов.
        x'chg   ax,bx
        rep     stosw           ; Заполнить остаток строки пробелами
                                ; с атрибутом OF.
        xchg    bx,ax           ; Восстановить значение EAX.
        pop     cx              ; Восстановить значение счетчика.
continue_loop:
        loop    cloop

        stosd                   ; Записать последний (256-й) символ и пробел.

; собственно вывод на экран
        mov     ax,0B800h       ; Сегментный адрес видеопамяти.
        mov     es,ax
        xor     di,di           ; DI = 0, адрес начала видеопамяти в ES:DI;
        mov     si,offset ctable ; Адрес таблицы в DS:SI.
        mov     cx,15*80+32     ; 15 строк по 80 символов, последняя строка - 32.
        rep     movsw           ; Скопировать таблицу ctable в видеопамять.
        ret                     ; Завершение COM-файла.

ctable:                         ; Данные для вывода на экран начинаются сразу
                                ; за концом файла. В EXE-файле такие данные
                                ; определяют в сегменте .data?
        end     start
```

В указанной программе при подготовке данных для копирования в видеопамять учитывалось следующее: в архитектуре Intel при записи слова (или двойного слова) в память старший байт располагается по старшему адресу. Так что при записи в память двойного слова 1F201F00h сначала записывается самый младший байт 00h (ASCII-код текущего символа), потом 1Fh, используемый в этом примере атрибут, далее 20h (код пробела) и лишь затем, по самому старшему адресу, - самый старший байт, 1Fh, атрибут для этого пробела. Кроме того, в данном примере использовались некоторые 32-битные команды (MOV и STOSD). Ими можно пользоваться из 16-битной программы (разумеется, если процессор 80386 и выше), но не стоит этим злоупотреблять, потому что каждая из команд оказывается длиннее на 1 байт и выполняется дольше на 1 такт.

4.4. Ввод с клавиатуры

4.4.1. Средства DOS

Как и в случае вывода на экран, DOS предоставляет набор функций для чтения данных с клавиатуры, которые используют стандартное устройство ввода

STDIN, так что в качестве источника данных можно применить файл или стандартный вывод другой программы.

Функция DOS 0Ah: Считать строку символов из STDIN в буфер
Вход: AH = OAh
DS:DX = адрес буфера
Выход: Буфер содержит введенную строку

Для вызова этой функции надо подготовить буфер, первый байт которого заключает в себе максимальное число символов для ввода (1–254), а содержимое, если оно задано, может использоваться как подсказка для ввода. При наборе строки обрабатываются клавиши **Esc, F3, F5, BS, Ctrl-C/Ctrl-Break** и т. д., как при наборе команд DOS (то есть Esc начинает ввод сначала, **F3** восстанавливает подсказку для ввода, **F5** запоминает текущую строку как подсказку, **Backspace** стирает предыдущий символ). После нажатия клавиши Enter строка (включая последний символ CR (ODh)) записывается в буфер, начиная с третьего байта. Во второй байт записывается длина реально введенной строки без учета последнего CR.

Рассмотрим пример программы, выполняющей преобразование десятичного числа в шестнадцатеричное.

```
; dosin1.asm
; Переводит десятичное число в шестнадцатеричное.
;
        .model  tiny
        .code
        .286                    ; Для команды shr al,4.
        org     100h            ; Начало COM-файла.
start:
        mov     dx,offset message1
        mov     ah,9
        int     21h             ; Вывести приглашение ко вводу message1.
        mov     dx,offset buffer
        mov     ah,0Ah
        int     21h             ; Считать строку символов в буфер.
        mov     dx,offset crlf
        mov     ah,9
        int     21h             ; Перевод строки.

; Перевод числа в ASCII-формате из буфера в бинарное число в AX.
        xor     di,di           ; DI = 0 - номер байта в буфере.
        xor     ax,ax           ; AX = 0 - текущее значение результата.
        mov     cl,blength
        xor     ch,ch
        xor     bx,bx
        mov     si,cx           ; SI - длина буфера
        mov     cl,10           ; CL = 10/ множитель для MUL
asc2hex:
        mov     bl,byte ptr bcontents[di]
        sub     bl,'0'          ; Цифра = код цифры - код символа "0".
```

```
        jb      asc_error       ; Если код символа был меньше, чем код "0",
        cmp     bl,9            ; или больше, чем "9",
        ja      asc_error       ; выйти из программы с сообщением об ошибке.
        mul     cx              ; Иначе: умножить текущий результат на 10,
        add     ax,bx           ; добавить к нему новую цифру,
        inc     di              ; увеличить счетчик.
        cmp     di,si           ; Если счетчик+1 меньше числа символов -
        jb      asc2hex         ; продолжить (счетчик ведет отсчет от 0).

; Вывод на экран строки message2.
        push    ax              ; Сохранить результат преобразования.
        mov     ah,9
        mov     dx,offset message2
        int     21h
        pop     ax

; Вывод на экран числа из регистра AX.
        push    ax
        xchg    ah,al           ; Поместить в AL старший байт.
        call    print_al        : Вывести его на экран.
        pop     ax              ; Восстановить в AL младший байт.
        call    print_al        ; Вывести его на экран.

        ret                     ; Завершение COM-файла.

asc_error:
        mov     dx,offset err_msg
        mov     ah,9
        int     21h             ; Вывести сообщение об ошибке
        ret                     ; и завершить программу.

; Процедура print_al.
; Выводит на экран число в регистре AL
; в шестнадцатеричном формате,
; модифицирует значения регистров AX и DX.

print_al:
        mov     dh,al
        and     dh,0Fh          ; DH - младшие 4 бита.
        shr     al,4            ; AL - старшие.
        call    print_nibble    ; Вывести старшую цифру.
        mov     al,dh           ; Теперь AL содержит младшие 4 бита.
print_nibble:                   ; Процедура вывода 4 бит (шестнадцатеричной цифры).
        cmp     al,10           ; Три команды, переводящие цифру в AL
        sbb     al,69h          ; в соответствующий ASCII-код.
        das                     ; (см. описание команды DAS)

        mov     dl,al           ; Код символа в DL.
        mov     ah,2            ; Номер функции DOS в AH.
        int     21h             ; Вывод символа.
        ret                     ; Этот RET работает два раза - один раз
                                ; для возврата из процедуры print_nibble,
```

```
                                ; вызванной для старшей цифры,
                                ; и второй раз - для возврата из print_al.

message1        db      "Десятичное число: $"
message2        db      "Шестнадцатеричное число: $"
err_msg         db      "Ошибка ввода"
crlf            db      0Dh,0Ah,'$'
buffer          db      6       ; Максимальный размер буфера ввода.
blength         db      ?       ; Размер буфера после считывания.
bcontents:                      ; Содержимое буфера располагается за
                                ; концом COM-файла.
        end     start
```

Функция 0Ah предоставляет удобный, но ограниченный способ ввода данных. Чаще всего используют функции посимвольного ввода, позволяющие контролировать отображение символов на экране, реакцию программы на функциональные и управляющие клавиши и т. д.

Функция DOS 01h: Считать символ из STDIN с эхом, ожиданием и проверкой на **Ctrl-Break**

Вход: AH = 01h
Выход: AL = ASCII-код символа или 0. Если AL = 0, второй вызов этой функции возвратит в AL расширенный ASCII-код символа.

При чтении с помощью этой функции введенный символ автоматически отображается на экране (посылается в устройство STDOUT - так что его можно перенаправить в файл). При нажатии **Ctrl-C** или **Ctrl-Break** выполняется команда INT 23h. Если нажата клавиша, не соответствующая какому-нибудь символу (стрелки, функциональные клавиши **Ins,** Del и т. д.), то в AL возвращается 0 и функцию надо вызвать еще один раз, чтобы получить расширенный ASCII-код (см. приложение 1).

В трех следующих вариантах этой функции код символа возвращается в AL по такому же принципу.

Функция DOS 08h: Считать символ из STDIN без эха, с ожиданием и проверкой на **Ctrl-Break**

Вход: AH = 08h
Выход: AL = код символа

Функция DOS 07h: Считать символ из STDIN без эха, с ожиданием и без проверки на **Ctrl-Break**

Вход: AH - 07h
Выход: AL = код символа

Функция DOS 06h: Считать символ из STDIN без эха, без ожидания и без проверки на **Ctrl-Break**

Вход: AH = 06h
DL = 0FFh

Выход: ZF = 1, если не была нажата клавиша, и AL = 00
ZF = 0, если клавиша была нажата. В этом случае AL = код символа

Кроме перечисленных могут потребоваться и некоторые служебные функции DOS для работы с клавиатурой.

Функция DOS 0Bh: Проверить состояние клавиатуры
Вход: AH = OBh
Выход: AL = 0, если не была нажата клавиша
AL = OFFh, если была нажата клавиша

Данную функцию удобно использовать перед функциями 01, 07 и 08, чтобы не ждать нажатия клавиши. Кроме того, вызов указанной функции позволяет проверить, не считывая символ с клавиатуры, была ли нажата комбинация клавиш **Ctrl-Break;** если это произошло, выполнится прерывание 23h.

Функция DOS 0Ch: Очистить буфер и считать символ
Вход: AH = 0Ch
AL = Номер функции DOS (01, 06, 07, 08, OAh)
Выход: Зависит от вызванной функции

Функция 0Ch очищает буфер клавиатуры, так что следующая функция чтения символа будет ждать ввода с клавиатуры, а не использовать нажатый ранее и еще не обработанный символ. Например, именно эта функция используется для считывания ответа на вопрос «Уверен ли пользователь в том, что он хочет отформатировать диск?».

Функции посимвольного ввода без эха можно использовать для интерактивного управления программой, как в следующем примере.

```
; dosin2.asm
; Изображает пентамино F, которое можно перемещать по экрану клавишами
; управления курсором и вращать клавишами X и Z. Выход из программы - Esc.
;
line_length = 3                 ; Число символов в строке изображения.
number_of_lines = 3             ; Число строк.

        .model  tiny
        .code
        org     100h            ; Начало COM-файла.
start:
        cld                     ; Будут использоваться команды
                                ; строковой обработки.
        mov     ax,0B800h       ; Адрес начала текстовой видеопамяти -
        mov     es,ax           ; в ES.
        mov     ax,0003h
        int     10h             ; Текстовый режим 03 (80x25).
        mov     ah,02h          ; Установить курсор
        mov     bh,0
        mov     dh,26           ; на строку 26, то есть за пределы экрана.
```

```
        mov     dl,1
        int     10h             ; Теперь курсора на экране нет.

        call    update_screen   ; Вывести изображение.

; Основной цикл опроса клавиатуры.
main_loop:
        mov     ah,08h          ; Считать символ с клавиатуры
        int     21h             ; без эха, с ожиданием, с проверкой на Ctrl-Break.
        test    al,al           ; Если AL = 0,
        jz      eASCII_entered  ; введен символ расширенного ASCII.
        cmp     al,1Bh          ; Иначе: если введен символ 1Bh (Esc),
        je      key_ESC         ; выйти из программы.
        cmp     al,'Z'          ; Если введен символ Z,
        je      key_Z           ; перейти на его обработчик.
        cmp     al,'z'          ; То же для z.
        je      key_Z
        cmp     al,'X'          ; Если введен символ X,
        je      key_X           ; перейти на его обработчик.
        cmp     al,'x'          ; То же для x.
        je      key_X
        jmp     short main_loop ; Считать следующую клавишу.

eASCII_entered:                 ; Был введен расширенный ASCII-символ.
        int     21h             ; Получить его код (повторный вызов функции).
        cmp     al,48h          ; Стрелка вверх.
        je      key_UP
        cmp     al,50h          ; Стрелка вниз.
        je      key_DOWN
        cmp     al,4Bh          ; Стрелка влево.
        je      key_LEFT
        cmp     al,4Dh          ; Стрелка вправо.
        je      key_RIGHT
        jmp     short main_loop ; Считать следующую клавишу.
;
; Обработчики нажатий клавиш.
;
key_ESC:                                ; Esc
        ret                             ; Завершить COM-программу.
key_UP:                                 ; Стрелка вверх.
        cmp     byte ptr start_row,0      Если изображение на верхнем
                                        ; краю экрана,
        jna     mainJLoop                 считать следующую клавишу.
        dec     byte ptr start_row        Иначе - уменьшить номер строки,
        call    update_screen               вывести новое изображение
        jmp     short main_loop           и считать следующую клавишу.

key_DOWN:                               ; Стрелка вниз.
        cmp     byte ptr start_row,25-number_of_lines ; Если
                                        ; изображение на нижнем краю экрана,
        jnb     main_loop                 считать следующую клавишу.
```

```
        inc     byte ptr start_row      ; Иначе - увеличить номер строки,
        call    update_screen           ; вывести новое изображение
        jmp     short main_loop         ; и считать следующую клавишу.

key_LEFT:                               ; Стрелка влево.
        cmp     byte ptr start_col,0    ; Если изображение на левом краю экрана,
        jna     main_loop               ; считать следующую клавишу.
        dec     byte ptr start_col      ; Иначе - уменьшить номер столбца,
        call    update_screen           ; вывести новое изображение
        jmp     short main_loop         ; и считать следующую клавишу.

key_RIGHT:                              ; Стрелка вправо.
        cmp     byte ptr start_col,80-line_length ; Если
                                        ; изображение на правом краю экрана,
        jnb     main_loop               ; считать следующую клавишу.
        inc     byte ptr start_col      ; Иначе - увеличить номер столбца,
        call    update_screen           ; вывести новое изображение
        jmp     short main_loop         ; и считать следующую клавишу.

key_Z:                                  ; Клавиша Z (вращение влево).
        mov     ax,current_screen       ; Считать номер текущего изображения
                                        ; (значения 0, 1, 2, 3),
        dec     ax                      ; уменьшить его на 1.
        jns     key_Z_ok                ; Если получился -1 (поменялся знак),
        mov     ax,3                    ; AX = 3.
key_Z_ok:
        mov     current_screen,ax       ; Записать номер обратно,
        call    update_screen           ; вывести новое изображение
        jmp     main_loop               ; и считать следующую клавишу.
key_X:                                  ; Клавиша X (вращение вправо).
        mov     ax,current_screen       ; Считать номер текущего изображения
                                        ; (значения 0, 1, 2, 3),
        inc     ax                      ; увеличить его на 1.
        cmp     ax,4                    ; Если номер стал равен 4,
        jne     key_X_ok
        xor     ax,ax                   ; AX = 0.
key_X_ok:
        mov     current_screen,ax       ; Записать номер обратно,
        call    update_screen           ; вывести новое изображение
        jmp     main_loop               ; и считать следующую клавишу.

; Процедура update_screen.
; Очищает экран и выводит текущее изображение.
; Модифицирует значения регистров AX, BX, CX, DX, SI, DI.
update_screen:
        mov     cx,25*80                ; Число символов на экране.
        mov     ax,0F20h                ; Символ 20h (пробел) с атрибутом 0Fh
                                        ; (белый на черном).
        xor     di,di                   ; ES:DI = начало видеопамяти.
        rep     stosw                   ; Очистить экран.
```

```
        mov     bx,current_screen; Номер текущего изображения в BX.
        shl     bx,1            ; Умножить на 2, так как screens - массив слов.
        mov     si,screens[bx]  ; Поместить в BX смещение начала
                                ; текущего изображения из массива screens.
        mov     ax,start_row    ; Вычислить адрес начала
        mul     row_length      ; изображения в видеопамяти:
        add     ax,start_col    ; (строка x 80 + столбец) x 2.
        shl     ax,1
        mov     di,ax           ; ES:DI - начало изображения в видеопамяти.
        mov     ah,0Fh          ; Используемый атрибут - белый на черном.
        mov     dx,number_of_lines ; Число строк в изображении.
copy_lines:
        mov     cx,line_length  ; Число символов в строке.
copy_1: lodsb                   ; Считать ASCII-код в AL,
        stosw                   ; записать его в видеопамять
                                ; (AL - ASCII, AH - атрибут).
        loop    copy_1          ; Вывести так все символы в строке.
        add     di,(80-line_length) * 2 ; Перевести DI на начало
                                ; следующей строки экрана.
        dec     dx              ; Если строки не закончились -
        jnz     copy_lines      ; вывести следующую.

        ret                     ; Конец процедуры update_screen.

; Изображение пентамино F.
screen1         db      " XX"   ; Выводимое изображение.
                db      "XX "
                db      " X "

screen2         db      " X "   ; Поворот на 90 градусов вправо.
                db      "XXX"
                db      "  X"

screen3         db      " X"    ; Поворот на 180 градусов.
                db      " XX"
                db      "XX"

screen4 d       b       "X  "   ; Поворот на 90 градусов влево.
                db      "XXX"
                db      " X "
; Массив, содержащий адреса всех вариантов изображения.
screens         dw      screen1,screen2,screen3,screen4
current_screen  dw      0       ; Текущий вариант изображения.
start_row       dw      10      ; Текущая верхняя строка изображения.
start_col       dw      37      ; Текущий левый столбец.
row_length      db      80      ; Длина строки экрана для команды MUL.

        end     start
```

В этом примере для вывода на экран используется прямое копирование в видеопамять, так как вызов функции BIOS вывода строки (INT 10h, AH = 13h)

прокручивает экран вверх на одну строку при выводе символа в нижнем правом углу экрана.

4.4.2. *Средства BIOS*

Так же как и для вывода на экран, BIOS предоставляет больше возможностей по сравнению с DOS для считывания данных и управления клавиатурой. Например, функциями DOS нельзя определить нажатие комбинаций клавиш типа **Ctrl-Alt-Enter** или нажатие двух клавиш **Shift** одновременно, DOS не может определить момент отпускания нажатой клавиши, и наконец, в DOS нет аналога функции C ungetch(), помещающей символ в буфер клавиатуры, как если бы его ввел пользователь. Все это можно осуществить, используя различные функции прерывания 16h и операции с байтами состояния клавиатуры.

INT 16h, AH = 0, 10h, 20h: Чтение символа с ожиданием
Вход: AH = 00h (83/84-key), 10h (101/102-key), 20h (122-key)
Выход: AL = ASCII-код символа, 0 или префикс скан-кода
AH = скан-код нажатой клавиши или расширенный ASCII-код

Каждой клавише на клавиатуре соответствует так называемый скан-код (см. приложение 1), соответствующий только этой клавише. Этот код посылается клавиатурой при каждом нажатии и отпускании клавиши и обрабатывается BIOS (обработчиком прерывания INT 9). Прерывание 16h дает возможность получить код нажатия, не перехватывая этот обработчик. Если нажатой клавише соответствует ASCII-символ, то в AH возвращается код этого символа, а в AL - скан-код клавиши. Если нажатой клавише соответствует расширенный ASCII-код, в AL возвращается префикс скан-кода (например, OEOh для серых клавиш) или 0, если префикса нет, а в AH - расширенный ASCII-код. Функция OOh обрабатывает только комбинации, использующие клавиши 84-клавишной клавиатуры; 10h обрабатывает все 101–105-клавишные комбинации; 20h - 122-клавишные. Тип клавиатуры можно определить с помощью функции 09h прерывания 16h, если она поддерживается BIOS (поддерживается ли эта функция, можно узнать с помощью функции OCOh прерывания 15h).

INT 16h, AH = 1, 11h, 21h: Проверка символа
Вход: AH = 01h (83/84-key), 11h (101/102-key), 21h (122-key)
Выход: ZF = 1, если буфер пуст
ZF = 0, если в буфере присутствует символ, тогда
AL = ASCII-код символа, 0 или префикс скан-кода
AH = скан-код нажатой клавиши или расширенный ASCII-код

Символ остается в буфере клавиатуры, хотя некоторые BIOS удаляют символ из буфера при обработке функции Olh, если он соответствует расширенному ASCII-коду, отсутствующему на 84-клавишных клавиатурах.

INT 16h, AH = 05h: Поместить символ в буфер клавиатуры
Вход: AH = 05h
CH = скан-код
CL = ASCII-код
Выход: AL = 00, если операция выполнена успешно
AL = 01h, если буфер клавиатуры переполнен
AH модифицируется многими BIOS

Обычно вместо скан-кода в CH можно поместить 0, если функция, которая будет выполнять чтение из буфера, использует именно ASCII-код. Например, следующая программа при запуске из DOS вызывает команду DIR (но при запуске из некоторых оболочек, например FAR, этого не произойдет).

```
; ungetch.asm
; заносит в буфер клавиатуры команду DIR так, чтобы она выполнилась сразу после
; завершения программы
;
        .model  tiny
        .code
        org     100h        ; COM-файл
start:
        mov     cl,'d'      ; CL = ASCII-код буквы "d"
        call    ungetch
        mov     cl,'i'      ; ASCII-код буквы "i"
        call    ungetch
        mov     cl,'r'      ; ASCII-код буквы "r"
        call    ungetch
        mov     cl,0Dh      ; перевод строки
ungetch:
        mov     ah,5        ; AH = номер функции
        mov     ch,0        ; CH = 0 (скан-код неважен)
        int     16h         ; поместить символ в буфер
        ret                 ; завершить программу

        end     start
```

INT 16h, AH = 02h, 12h, 22h: Считать состояние клавиатуры
Вход: AH - 02h (83/84-key), 12h (101/102-key), 22h (122-key)
Выход: AL = байт состояния клавиатуры 1
AH = байт состояния клавиатуры 2 (только для функций 12h и 22h)

Байт состояния клавиатуры 1 (этот байт всегда расположен в памяти по адресу 0000h:0417h или 0040h:0017h):
бит 7: Ins включена
бит 6: CapsLock включена
бит 5: NumLock включена
бит 4: ScrollLock включена

бит 3: Alt нажата (любая **Alt** для функции 02h, часто только левая **Alt** для 12h/22h)
бит **2: Ctrl** нажата (любая **Ctrl)**
бит 1: Левая **Shift** нажата
бит 0: Правая **Shift** нажата

Байт состояния клавиатуры 2 (этот байт всегда расположен в памяти по адресу 0000h:0418h или 0040h:0018h):

бит **7: SysRq** нажата
бит **6: CapsLock** нажата
бит **5: NumLock** нажата
бит **4: ScrollLock** нажата
бит 3: Правая Alt нажата
бит 2: Правая Ctrl нажата
бит 1: Левая **Alt** нажата
бит 0: Левая **Ctrl** нажата

Оба байта постоянно располагаются в памяти, так что вместо вызова прерывания часто удобнее просто считывать значения напрямую. Более того, в эти байты можно записывать новые значения, и BIOS соответствующим образом изменит состояние клавиатуры:

```
; nolock.asm
; Самая короткая программа для выключения NumLock, CapsLock и ScrollLock.
; Запускать без параметров.
        .model  tiny
        .code
        org     100h                    ; COM-файл. AX при запуске COM-файла
                                        ; без параметров в командой строке
                                        ; всегда равен 0.
start:
        mov     ds,ax                   ; Так что теперь DS = 0.
        mov     byte ptr ds:0417h,al    ; Байт состояния клавиатуры 1 = 0.
        ret                             ; Выход из программы.
        end  ,  start
```

Разумеется, в реальных программах, которые будет запускать кто-то, кроме автора, так делать нельзя, и первой командой дожна быть xor ax,ax.

Помимо этих двух байт BIOS хранит в своей области данных и весь клавиатурный буфер, к которому также можно обращаться напрямую. Буфер занимает 16 слов с 0h:041Eh по Oh:043Dh включительно, причем по адресу 0h:041Ah лежит адрес (ближний) начала буфера, то есть адрес, по которому располагается следующий введенный символ, а по адресу 0h:041Ch находится адрес конца буфера, так что если эти два адреса равны, буфер пуст. Буфер действует как кольцо: если начало буфера - 043Ch, а конец - 0420h, то в буфере расположены три символа по адресам 043Ch, 041Eh и 0420h. Каждый символ хранится в виде слова - того же самого, которое возвращает функция 10h прерывания INT 16h. В некоторых случаях (если) буфер размещается по другим адресам, тогда адрес его начала хранится

в области данных BIOS по адресу 0480h, а конца - по адресу 0482h. Прямой доступ к буферу клавиатуры лишь немногим быстрее, чем вызов соответствующих функций BIOS, и для приложений, требующих максимальной скорости, таких как игры или демо-программы, используют управление клавиатурой на уровне портов ввода-вывода.

4.5. Графические видеорежимы

4.5.7. *Работа с VGA-режимами*

Функция 00 прерывания BIOS 10h позволяет переключаться не только в текстовые режимы, использовавшиеся в предыдущих главах, но и в некоторые графические. Эти видеорежимы стандартны и поддерживаются всеми видеоадаптерами (начиная с VGA), см. табл. 19.

Таблица 19. Основные графические режимы VGA

Номер режима	Разрешение	Число цветов
11h	640x480	2
12h	640x480	16
13h	320x200	256

Существуют еще несколько видеорежимов, использовавшихся более старыми видеоадаптерами CGA и EGA (с номерами от 4 до 10h).

BIOS также предоставляет видеофункции чтения и записи точки на экране в графических режимах, но эти функции настолько медленно исполняются, что никогда не применяются в реальных программах.

INT 10h AH = 0Ch: Вывести точку на экран
Вход: AH = 0Ch
BH = номер видеостраницы (игнорируется для режима 13h, поддерживающего только одну страницу)
DX = номер строки
CX = номер столбца
AL = номер цвета (для режимов 10h и 11h, если старший бит 1, номер цвета точки на экране будет результатом операции «исключающее ИЛИ»)
Выход: Никакого

INT 10h AH = 0Dh: Считать точку с экрана
Вход: AH = ODh
BH = номер видеостраницы (игнорируется для режима 13h, поддерживающего только одну страницу)
DX = номер строки
CX = номер столбца
Выход: AL = номер цвета

Попробуем тем не менее воспользоваться средствами BIOS для вывода на экран. Следующая программа переводит экран в графический режим 13h (320×200), заселяет его точками случайным образом, после чего эти точки эволюционируют согласно законам алгоритма «Жизнь»: если у точки меньше двух или больше трех соседей, она погибает, а если у пустой позиции есть три соседа, в ней появляется новая точка. Мы будем использовать очень простой, но неоптимальный способ реализации этого алгоритма: сначала для каждой точки вычисляется число соседей, затем каждая точка преобразуется в соответствии с полученным числом соседей, после чего каждая точка выводится на экран.

```
; lifebios.asm
; Игра "Жизнь" на поле 320X200, использующая вывод на экран средствами BIOS.

        .model  small
        .stack  100h                ; Явное задание стека - для EXE-программ.
        .code
        .186                        ; Для команд shl al,4 и shr al,4.
start:
        push    FAR_BSS             ; Сегментный адрес буфера в DS.
        pop     ds

; Заполнение массива ячеек псевдослучайными значениями.
        xor     ax,ax
        int     1Ah                 ; Функция AH = 0 INT 1Ah: получить текущее
                                    ; время.
                                    ; DX теперь содержит число секунд,
                                    ; прошедших с момента включения компьютера,
                                    ; которое используется как начальное значение
                                    ; генератора случайных чисел.
        mov     di,320*200+1        ; Максимальный номер ячейки.
fill_buffer:
        imul    dx,4E35h            ; Простой генератор случайных чисел
        inc     dx                  ; из двух команд.
        mov     ax,dx               ; Текущее случайное число копируется в AX,
        shr     ax,15               ; от него оставляется только один бит,
        mov     byte ptr [di],al    ; и в массив копируется 00, если ячейка
                                    ; пуста, и 01, если заселена.
        dec     di                  ; Следующая ячейка.
        jnz     fill_buffer         ; Продолжить цикл, если DI не стал равен нулю.

        mov     ax,0013h            ; Графический режим 320x200, 256 цветов.
        int     10h

; Основной цикл.

new_cycle:

; Шаг 1: для каждой ячейки вычисляется число соседей
; и записывается в старшие 4 бита этой ячейки.

        mov     di,320*200+1        ; Максимальный номер ячейки.
```

```
step_1:
        mov     al,byte ptr. [di+1]     ; В AL вычисляется сумма
        add     al,byte ptr [di-1]      ; значений восьми соседних ячеек,
        add     al,byte ptr [di+319]    ; при этом в младших четырех
        add     al,byte ptr [di-319]    ; битах накапливается число
        add     al,byte ptr [di+320]    ; соседей.
        add     al,byte ptr [di-320]
        add     al,byte ptr [di+321]
        add     al,byte ptr [di-321]
        shl     al,4                    ; Теперь старшие четыре бита AL - число
                                        ; соседей текущей ячейки.
        or      byte ptr [di],al        ; Поместить их в старшие четыре бита
                                        ; текущей ячейки.
        dec     di                      ; Следующая ячейка.
        jnz     stepj                   ; Продолжить цикл, если DI не стал равен нулю.

; Шаг 2: изменение состояния ячеек в соответствии с полученными в шаге 1
; значениями числа соседей.

        mov     di,320*200+1            ; Максимальный номер ячейки.
flip_cycle:
        mov     al,byte ptr [di]        ; Считать ячейку из массива,
        shr     al,4                    ; AL = число соседей.
        cmp     al,3                    ; Если число соседей = 3,
        je      birth                   ; ячейка заселяется.
        cmp     al,2                    ; Если число соседей = 2,
        je      f_c_continue            ; ячейка не изменяется.
        mov     byte ptr [di],0         ; Иначе - ячейка погибает.
        jmp     short f_c_continue
birth:
        mov     byte ptr [di],1
f_c_continue:
        and     byte ptr [di],0Fh       ; Обнулить число соседей в старших
                                        ; битах ячейки.
        dec     di                      ; Следующая ячейка.
        jnz     flip_cycle

;
; Вывод массива на экран средствами BIOS.
;
        mov     si,320*200+1            ; Максимальный номер ячейки.
        mov     cx,319                  ; Максимальный номер столбца.
        mov     dx,199                  ; Максимальный номер строки.
zdisplay:
        mov     al,byte ptr [si]        ; Цвет точки (00 - черный, 01 - синий).
        mov     ah,0Ch                  ; Номер видеофункции в AH.
        int     10h                     ; Вывести точку на экран.
        dec     si                      ; Следующая ячейка.
        dec     cx                      ; Следующий номер столбца.
        jns     zdisplay                ; Если столбцы не закончились - продолжить.
```

```
mov     cx,319          ; Иначе: снова максимальный номер столбца в CX
dec     dx              ; и следующий номер строки в DX.
jns     zdisplay        ; Если и строки закончились - выход из цикла.

mov     ah, 1           ; Если не нажата клавиша -
int     16h
jz      new_cycle       ; следующий шаг жизни.

mov     ax,0003h        ; Восстановить текстовый режим
int     10h
mov     ax,4C00h        ; и завершить программу.
int     21h

.fardata?               ; Сегмент дальних неинициализированных данных
        db      320*200+1 dup(?) ; содержит массив ячеек.
end     start
```

Этот фрагмент оформлен как EXE-программа, потому что применяется массив, близкий по размерам к сегменту, и если разместить его в одном сегменте с COM-программой, стек, растущий от самых старших адресов, может затереть область данных. В нашем примере стек не используется, но он нужен обработчику прерывания BIOS 10h.

Скорость работы указанной программы - в среднем 200 тактов процессора Pentium на точку (измерения выполнены *с* помощью команды RDTSC, см. раздел 10.2), то есть всего 16 поколений в секунду для Pentium-200 (200 миллионов тактов в секунду разделить на 200 тактов на точку и на 320x200 точек). Разумеется, используемый алгоритм крайне нерационален и кажется очевидным, что его оптимизация приведет к значительному выигрышу во времени. Но если измерить скорость выполнения каждого из трех циклов, то окажется, что первый цикл выполняется в среднем за *20,5* такта на точку, второй - за 13, а третий - за 170,5!

Исправить эту ситуацию весьма просто - достаточно отказаться от видеофункций BIOS для работы с графикой и перейти к прямому копированию в видеопамять.

В видеорежиме 13h каждый байт в области памяти, начинающейся с адреса 0A000h:0000h, соответствует одной точке на экране, а значение, которое может принимать этот байт (0-255), - номеру цвета этой точки. (Цвета, которые соотносятся с данными номерами, могут быть перепрограммированы с помощью видеофункции 10h BIOS.) В видеорежимах 11h и 12h каждый бит соответствует одной точке на экране, так что простым копированием в видеопамять можно получить только черно-белое изображение (для вывода цветного изображения в режиме 12h необходимо перепрограммировать видеоадаптер; об этом см. в разделе 5.10.4).

В нашем примере для хранения информации о каждой ячейке также используется один байт, следовательно, для вывода данных на экран в режиме 13h достаточно выполнить простое копирование. Переименуем программу LIFEBIOS.ASM в LIFEDIR.ASM, заменив цикл вывода на экран от команды

```
mov     si,320*200+1
```

до второй команды

```
jns     zdisplay
```

следующим фрагментом кода:

```
push    0A000h          ; Сегментный адрес видеопамяти
pop     es              ; в ES.
mov     cx,320*200      ; Максимальный номер точки
mov     di,cx           ; в видеопамяти - 320x200,
mov     si,cx           ; а в массиве -
inc     si              ; 320x200 + 1.
rep     raovsb          ; Выполнить копирование в видеопамять.
```

Теперь программа обрабатывает одну точку приблизительно за 61,5 такта процессора Pentium, что дает 51 поколение в секунду на Pentium-200. Кроме того, сейчас эту программу можно переписать в виде COM-файла, так как и код, и массив, и стек точно умещаются в одном сегменте размером 64 Кб. Такая COM-программа (LIFECOM.ASM) займет 143 байта.

Оптимизация программы «Жизнь» - хорошее упражнение для программирования на ассемблере. В 1997 году проводился конкурс на самую короткую и на самую быструю программу, выполняющую в точности то же, что и наш пример, - заполнение экрана случайными точками, их эволюция и выход по нажатию любой клавиши. Самой короткой тогда оказалась программа размером в 72 байта, которая с тех пор была усовершенствована до 64 байт (ее скорость 52 такта на точку), а самая быстрая из 16-битных программ тратит на каждую точку в среднем всего 6 тактов процессора Pentium и имеет размер 689 байт. В ней состояния ячеек описываются отдельными битами массива, а для их обработки используются команды логических операций над целыми словами, поэтому одна команда обслуживает сразу 16 точек. Применение 32-битных команд с тем же алгоритмом позволяет ускорить программу до 4,5 такта на точку.

4.5.2. *Работа с SVGA-режимами*

В режиме VGA 320x200 с 256 цветами для отображения видеопамяти на основное адресное пространство используется 64 000 байт, располагающихся с адреса 0A000h:0000h. Дальнейшее увеличение разрешения или числа цветов приводит к тому, что объем видеопамяти превышает максимальные границы сегмента в реальном режиме (65 535 байт), а затем и размер участка адресного пространства, отводимого для видеопамяти (160 Кб, от 0A000h:0000h до 0B800h:0FFFFh. С адреса 0C800h:0000h начинается область ROM BIOS). Чтобы вывести изображение, используются два механизма - переключение банков видеопамяти для реального режима и LFB (линейный кадровый буфер) для защищенного.

Во втором случае видеопамять отображается на непрерывный кусок адресного пространства, но начинающегося не с 0A0000h, а с какого-нибудь другого адреса, так чтобы весь массив видеопамяти, который может занимать несколько мегабайтов, отобразился в одну непрерывную область. В защищенном режиме максимальный

размер сегмента составляет 4 Гб, поэтому никаких сложностей с адресацией этого буфера не возникает. Буфер LFB можно использовать, только если видеоадаптер поддерживает спецификацию VBE 2.0 (см. пример в разделе 6.4).

В реальном режиме вывод на экран осуществляется по-прежнему копированием данных в 64-килобайтный сегмент, обычно начинающийся с адреса 0A000h:0000h, но эта область памяти соответствует только части экрана. Чтобы вывести изображение в другую часть экрана, нужно вызвать функцию перемещения окна (или, что то же самое, переключения банка видеопамяти), изменяющую область видеопамяти, которой соответствует сегмент OAOOOh. Например, в режиме 640x480 с 256 цветами требуется 307 200 байт для хранения всего видеоизображения. Заполнение сегмента 0A000h:0000h - 0A000h:0FFFFh приводит к закраске приблизительно 1/5 экрана, перемещение окна А на позицию 1 (или переключение на банк 1) и повторное заполнение этой же области - к закраске следующей 1/5 экрана и т. д. Перемещение окна осуществляется подфункцией 05 видеофункции 4Fh или передачей управления прямо на процедуру, адрес которой можно получить, активизировав подфункцию 01, как будет показано ниже. Некоторые видеорежимы позволяют использовать сразу два таких 64-килобайтных окна, окно А и окно В, так что можно записать 128 Кб данных, не вызывая прерывания.

Стандартные графические режимы SVGA могут быть 4-, 8-, 15-, 16-, 24- и 32-битными.

4-битные режимы (16 цветов)

VGA
- 012h: 640x480 (64 Кб)

VESA VBE 1.0
- 102h: 800x600 (256 Кб)
- 104h: 1024x768 (384 Кб)
- 106h: 1280x1024 (768 Кб)

Каждый пиксел описывается одним битом, для вывода цветного изображения требуется программирование видеоадаптера на уровне портов ввода-вывода (раздел 5.10.4).

8-битные режимы (256 цветов)

VGA
- 013h: 320x200 (64 Кб)

VBE 1.0
- 100h: 640x400 (256 Кб)
- 101h: 640x480 (320 Кб)
- 103h: 800x600 (512 Кб)
- 105h: 1024x768 (768 Кб)
- 107h: 1280x1024 (1,3 Мб)

VBE 2.0
- 120h: 1600x1200 (1,9 Мб)

Каждый пиксел описывается ровно одним байтом. Значение байта - номер цвета из палитры, значения цветов которой можно изменять, например вызывая подфункцию 09h видеофункции 4Fh.

15-битные режимы (32 К цветов)

VBE 1.2
- 10Dh: 320x200 (128 Кб)
- 110h: 640x480 (768 Кб)
- 113h: 800x600 (1 Мб)
- 116h: 1024x768 (1,5 Мб)
- 119h: 1280x1024 (2,5 Мб)

VBE 2.0
- 121h: 1600x1200 (3,8 Мб)

Каждый пиксел описывается ровно одним словом (16 бит), в котором биты 0-4 содержат значение синей компоненты цвета, биты 5-9 - зеленой, а биты 10-14 - красной. Бит 15 не используется.

16-битные режимы (64 К цветов)

VBE 1.2
- 10Eh: 320x200 (128 Кб)
- 111h: 640x480 (768 Кб)
- 114h: 800x600 (1 Мб)
- 117h: 1024x768 (1,5 Мб)
- 11Ah: 1280x1024 (2,5 Мб)

VBE 2.0
- 122h: 1600x1200 (3,8 Мб)

Так же как и в 15-битных режимах, каждый пиксел описывается, ровно одним словом. Обычно биты 0-4 (5 бит) содержат значение синей компоненты, биты 5-10 (6 бит) - зеленой, а биты 11-15 (5 бит) - красной. В нестандартных режимах число битов, отводимое для каждого цвета, может отличаться, так что при их использовании следует вызвать подфункцию 01 видеофункции 4Fh и получить информацию о видеорежиме, включающую битовые маски и битовые смещения для цветов.

24-битные и 32-битные режимы (16 М цветов)

VBE 1.2
- 10Fh: 320x200 (192 Кб)
- 112h: 640x480 (1 Мб)
- 115h: 800x600 (1,4 Мб)
- 118h: 1024x768 (2,3 Мб)
- 11Bh: 1280x1024 (3,7 Мб)

В режимах с 24-битным и 32-битным цветом каждому пикселу на экране соответствуют три байта и одно двойное слово (4 байта). Если видеорежим использует модель памяти 6 (Direct Color), то младший байт (байт 0) содержит значение синего цвета, байт 1 содержит значение зеленого, байт 2 - значение красного,

а байт 3 - в 32-битных режимах резервный и используется либо для выравнивания, либо содержит значение для альфа-канала. Некоторые видеорежимы могут использовать не Direct Color, а YUV (модель памяти 7) - здесь младший байт соответствует насыщенности красного, байт 1 - насыщенности синего, а байт 2 - яркости.

Видеоадаптер может поддерживать и собственные нестандартные видеорежимы. Список их номеров можно получить, вызвав подфункцию 00h, а получить информацию о режиме по его номеру - вызвав подфункцию 01h видеофункции 4Fh. Более того, для стандартных режимов также следует вызывать подфункцию O1h, чтобы проверить реальную доступность режима (например, режим может быть в списке, но не поддерживаться из-за нехватки памяти). VBE 2.0 разрешает видеоадаптерам не поддерживать никаких стандартных режимов вообще,

INT 10h AH = 4Fh, AL = 00: Получить общую SVGA-информацию
Вход: AX = 4FOOh
ES:DI = адрес буфера (512 байт)
Выход: AL = 4Fh, если функция поддерживается
AH = 01, если произошла ошибка
AH = 00, если данные получены и записаны в буфер

Буфер для общей SVGA-информации:
+00h: 4 байта- будет содержать VESA после вызова прерывания, чтобы получить поля, начиная с 14h, здесь надо предварительно записать строку VBE2
+04h: слово - номер версии VBE в двоично-десятичном формате (0102h - для 1.2, 0200h - для 2.0)
+06h: 4 байта- адрес строки-идентификатора производителя
+0Ah:4 байта– флаги:
бит 0: АЦП поддерживает 8-битные цветовые компоненты (см. подфункцию 08h)
бит 1: видеоадаптер несовместим с VGA
бит 2: АЦП можно программировать только при обратном ходе луча
бит 3: поддерживается спецификация аппаратного ускорения графики VBE/AF 1.0
бит 4: требуется вызов EnableDirectAccess перед использованием LFB
бит 5: поддерживается аппаратный указатель мыши
бит 6: поддерживается аппаратный clipping
бит 7: поддерживается аппаратный BitBlt
биты 8-31 зарезервированы
+0Eh: 4 байта - адрес списка номеров поддерживаемых видеорежимов (массив слов, последнее слово = OFFFFh, после которого обычно следует список нестандартных режимов, также заканчивающийся словом OFFFFh)
+12h: слово - объем видеопамяти в 64-килобайтных блоках
+14h: слово - внутренняя версия данной реализации VBE

+16h: 4 байта - адрес строки с названием производителя
+1Ah: 4 байта - адрес строки с названием видеоадаптера
+1Eh: 4 байта - адрес строки с версией видеоадаптера
+22h: слово - версия VBE/AF (BCD, то есть 0100h для 1.0)
+24h: 4 байта - адрес списка номеров режимов, поддерживающих аппаратное ускорение (если бит поддержки VBE/AF установлен в 1)
+28h: 216 байт – зарезервировано VESA
+100h: 256 байт - зарезервировано для внутренних данных VBE. Так, например, в эту область копируются строки с названиями производителя, видеоадаптера, версии и т. д.

INT 10h AH = 4Fh, AL = 01: Получить информацию о режиме
Вход: AX = 4F01h
CX = номер SVGA-режима (бит 14 соответствует использованию LFB, бит 13 - аппаратному ускорению)
ES:DI = адрес буфера для информации о режиме (256 байт)
Выход: AL = 4Fh, если функция поддерживается
AH = 01h, если произошла ошибка
AH = 00h, если данные получены и записаны в буфер

Буфер для информации о SVGA-режиме:
+00h: слово - атрибуты режима:
бит 0: режим присутствует
бит 1: дополнительная информация (смещения 12h - 1Eh) присутствует (для VBE 2.0 эта информация обязательна и бит всегда установлен)
бит 2: поддерживается вывод текста на экран средствами BIOS
бит 3: режим цветной
бит 4: режим графический
бит 5: режим несовместим с VGA
бит 6: переключение банков не поддерживается
бит 7: LFB не поддерживается
бит 8: не определен
бит 9: (для VBE/AF) приложения должны вызвать EnableDirectAccess, прежде чем переключать банки
+02h: байт - атрибуты окна A:
бит 1: окно существует
бит 2: чтение из окна разрешено
бит 3: запись в окно разрешена
+03h: байт - атрибуты окна B
+04h: слово - гранулярность окна - число килобайтов, которому всегда кратен адрес начала окна в видеопамяти (обычно 64)
+06h: слово - размер окна в килобайтах (обычно 64)
+08h: слово - сегментный адрес окна A (обычно OAOOOh)
+OAh: слово - сегментный адрес окна B

+0Ch: 4 байта - адрес процедуры перемещения окна (аналог подфункций 05h, но выполняется быстрее)
+10h: слово - число целых байтов в логической строке
+12h: слово - ширина в пикселах (для графики) или символах (для текста)
+14h: слово - высота в пикселах (для графики) или символах (для текста)
+16h: байт - высота символов в пикселах
+17h: байт - ширина символов в пикселах
+18h: байт - число плоскостей памяти (4 - для 16-цветных режимов, 1 - для обычных, число переключений банков, требуемое для доступа ко всем битам (4 или 8), - для модели памяти 5)
+19h: байт - число битов на пиксел
+1Ah: байт - число банков для режимов, в которых строки группируются в банки (2 - для CGA, 4 - для HGC)
+1Bh: байт - модель памяти:
00h - текст
01h - CGA-графика
02h - HGC-графика
03h - EGA-графика (16 цветов)
04h – VGA-графика (256 цветов в одной плоскости)
05h - Режим X (256 цветов в разных плоскостях)
06h - RGB (15-битные и выше)
07h-YUV
08h - 0Fh - зарезервированы VESA
10h - FFh - нестандартные модели
+1Ch: байт - размер банка в килобайтах (8 - для CGA и HGC, 0 - для остальных)
+1Dh: байт - число видеостраниц
+1Eh: байт - зарезервирован
+1Fh: байт - битовая маска красной компоненты
+20h: байт - первый бит красной компоненты
+21h: байт - битовая маска зеленой компоненты
+22h: байт - первый бит зеленой компоненты
+23h: байт - битовая маска синей компоненты
+24h: байт - первый бит синей компоненты
+25h: байт - битовая маска зарезервированной компоненты
+26h: байт - первый бит зарезервированной компоненты
+27h: байт - бит 0: поддерживается перепрограммирование цветов (подфункция 09h)
бит 1: приложение может использовать биты в зарезервированной компоненте
+28h: 4 байта - физический адрес начала LFB
+2Ch: 4 байта - смещение от начала LFB, указывающее на первый байт после конца участка памяти, отображающейся на экране
+30h: слово - размер памяти в LFB, не отображающейся на экране, в килобайтах
+32h: 206 байт - зарезервировано

INT 10h AH = 4Fh, AL = 02: Установить режим
Вход: AX = 4F02h
BX = номер режима:
биты 0-6: собственно номер режима
бит 7: видеопамять не очищается при установке режима, если все следующие биты - нули
бит 8: стандартный VBE SVGA-режим
бит 9: нестандартный SVGA-режим
биты 10-12: зарезервированы
бит 13: режим использует аппаратное ускорение
бит 14: режим использует LFB
бит 15: видеопамять не очищается при установке режима

Кроме того, специальный номер режима 81FFh соответствует доступу ко всей видеопамяти и может использоваться для сохранения ее содержимого.
Выход: AL = 4Fh, если функция поддерживается
AH = 00, если режим установлен
AH = 01 или 02, если произошла ошибка

INT 10h AH = 4Fh, AL = 03: Узнать номер текущего видеорежима
Вход: AX = 4F03h
Выход: AL = 4Fh, если функция поддерживается
BX = номер режима

INT 10h AH = 4FhAL = 05: Перемещение окна (переключение банка видеопамяти)
Вход: AX = 4F05h
BH = 00 - установить окно
BH = 01 - считать окно
BL = 00 - окно A
BL = 01 – окно B
DX = адрес окна в единицах гранулярности (номер банка), если BH = O
Выход: AL = 4Fh, если функция поддерживается
DX = адрес окна в единицах гранулярности (номер банка), если BH = 1
AH = 03, если функция была вызвана в режиме, использующем LFB

Всегда предпочтительнее переключать банки прямым дальним вызовом процедуры, адрес которой возвращается подфункцией 01h в блоке информации о видеорежиме. Все параметры передаются в процедуру точно так же, как и в подфункцию 05h, но содержимое регистров AX и DX по возвращении не определено.

INT 10h AH = 4FhAL = 07: Установка начала изображения
Вход: AX = 4F07h
BH-00
BL = 00 - считать начало изображения
BL = 80h - установить начало изображения (в VBE 2.0 автоматически выполняется при следующем обратном ходе луча)

CX = первый изображаемый пиксел в строке (для BL = 80h)
DX = первая изображаемая строка (для BL = 80h)
Выход: AL = 4Fh, если функция поддерживается
AH = 01, если произошла ошибка
AH = 00, если функция выполнилась успешно
BH = 00 (для BL = 00)
CX = первый изображаемый пиксел в строке (для BL = 00)
DX = первая изображаемая строка (для BL - 00)

С помощью этой функции можно выполнять как плавный сдвиг экрана, перемещая начало изображения на одну строку за один раз, так и быстрый показ двух разных изображений, изменяя одно, пока на экране находится другое, — своего рода эффект плавной анимации.

```
; scrolls.asm
; Изображает в разрешении 1024x768x64K окрашенный конус, который можно
; плавно перемещать по экрану стрелками вверх и вниз.
;
        .model  tiny
        .code
        .386                            ; Используется команда shrd.
        org     100h                    ; COM-файл.
start:
        mov     ax,4F01h                ; Получить информацию о видеорежиме.
        mov     cx,116h                 ; 1024x768x64K
        mov     di,offset vbe_mode_buffer
        int     10h
; Здесь для простоты опущена проверка наличия режима.
        mov     ax,4F02h                ; Установить режим.
        mov     bx,116h
        int     10h
        push    word ptr [vbe_mode_buffer+8]
        pop     es                      ; Поместить в ES адрес начала видеопамяти
                                        ; (обычно A000h).
        cld

; Вывод конуса на экран.

        mov     cx,-1                   ; Начальное значение цвета (белый).
        mov     si,100                  ; Начальный радиус.
        mov     bx,300                  ; Номер столбца.
        mov     ax,200                  ; Номер строки.
main_loop:
        inc     si                      ; Увеличить радиус круга на 1.
        inc     ax                      ; Увеличить номер строки.
        inc     bx                      ; Увеличить номер столбца.
        call    fast_circle             ; Нарисовать круг.
        sub     cx,0000100000100001b    ; Изменить цвет.
        cmp     si,350                  ; Если еще не нарисовано 350 кругов,
```

```
        jb      main_loop       ; продолжить.
        xor     cx,cx           ; Иначе: выбрать черный цвет,
        call    fast_circle     ; нарисовать последний круг.

; Плавное перемещение изображения по экрану с помощью функции 4F07h.

        xor     bx,bx           ; BX = 0 - установить начало экрана.
        xor     dx,dx           ; Номер строки = 0.
                                ; Номер столбца в CX уже ноль.
main_loop_2:
        mov     ax,4F07h
        int     10h             ; Переместить начало экрана.
        mov     ah,7            ; Считать нажатую клавишу с ожиданием, без эха
        int     21h             ; и без проверки на Ctrl-Break.
        test    al,al           ; Если это обычная клавиша -
        jnz     exit_loop_2     ; завершить программу.
        int     21h             ; Иначе: получить расширенный ASCII-код.
        cmp     al,50h          ; Если это стрелка вниз
        je      key_down
        cmp     al,48h          ; или вверх - вызвать обработчик.
        je      key_up
exit_loop_2:                    ; Иначе - завершить программу.
        mov     ax,3            ; Текстовый режим.
        int     10h
        ret                     ; Завершить COM-программу.

key_down:                       ; Обработчик нажатия стрелки вниз.
        dec     dx              ; Уменьшить номер строки начала экрана.
        jns     main_loop_2     ; Если знак не изменился - продолжить цикл.
                                ; Иначе (если номер был 0, а стал -1) -
                                ; увеличить номер строки.
key_up:                         ; Обработчик нажатия стрелки вверх.
        inc     dx              ; Увеличить номер строки начала экрана.
        jmp     short main_loop_2

; Процедура вывода точки на экран в 16-битном видеорежиме.
; Вход: DX = номер строки, BX = номер столбца, ES = 0A000h, CX = цвет.
; Модифицирует AX.
putpixel16b:
        push    dx
        push    di
        xor     di,di
        shrd    di,dx,6         ; DI = номер строки x 1024 mod 65 536.
        shr     dx,5            ; DX = номер строки / 1024 x 2.
        inc     dx
        cmp     dx,current_bank ; Если номер банка для выводимой точки
        jne     bank_switch     ; отличается то текущего - переключить банки.
switched:
        add     di,bx           ; Добавить к DI номер столбца.
        mov     ax,cx           ; Цвет в AX.
```

```
        shl     di,1            ; DI = DI x 2, так как адресация идет в словах.
        stosw                   ; Вывести точку на экран.
        pop     di              ; Восстановить регистры.
        pop     dx
        ret
bank_switch:                    ; Переключение банка.
        push    bx
        xor     bx,bx           ; BX = 0 -> установить начало экрана.
        mov     current_bank,dx ; Сохранить новый номер текущего банка.
        call    dword ptr [vbe_mode_buffer+0Ch]  ; Переключить банк.
        pop     bx
        jmp     short switched

; Алгоритм рисования круга, используя только сложение, вычитание и сдвиги
; (упрощенный алгоритм промежуточной точки).
; Вход: SI = радиус, CX = цвет, AX = номер столбца центра круга.
; BX = номер строки центра круга модифицирует DI, DX.
fast_circle:
        push    si
        push    ax
        push    bx
        xor     di,di           ; DI - относительная X-координата текущей точки.
        dec     di              ; (SI - относительная Y-координата, начальное
        mov     ax,1            ; значение - радиус).
        sub     ax,si           ; AX - наклон (начальное значение 1-Радиус).
circle_loop:
        inc     di              ; Следующий X (начальное значение - 0).
        cmp     di,si           ; Цикл продолжается, пока X < Y.
        ja      exit_main_loop

        pop     bx              ; BX = номер строки центра круга.
        pop     dx              ; DX = номер столбца центра круга.
        push    dx
        push    bx

        push    ax              ; Сохранить AX (putpixel16b его изменяет).
        add     bx,di           ; Вывод восьми точек на окружности:
        add     dx,si
        call    putpixel16b     ; центр_X + X, центр_Y + Y,
        sub     dx,si
        sub     dx,si
        call    putpixel16b     ; центр_X + X, центр_Y - Y,
        sub     bx,di
        sub     bx,di
        call    putpixel16b     ; центр_X - X, центр_Y - Y,
        add     dx,si
        add     dx,si
        call    putpixel16b     ; центр_X - X, центр_Y + Y,
        sub     dx,si
        add     dx,di
```

```
        add     bx,di
        add     bx,si
        call    putpixel16b     ; центр_X + Y, центр_Y + X,
        sub     dx,di
        sub     dx,di
        call    putpixel16b     ; центр_X + X, центр_Y - X,
        sub     bx,si
        sub     bx,si
        call    putpixel16b     ; центр_X - Y, центр_Y - X,
        add     dx,di
        add     dx,di
        call    putpixel16b     ; центр_X - Y, центр_Y + X.
        pop     ax

        test    ax,ax           ; Если наклон положительный,
        js      slop_negative
        mov     dx,di
        sub     dx,si
        shl     dx,1
        inc     dx
        add     ax,dx           ; наклон  = наклон + 2(X '- Y) + 1,
        dec     si              ; Y = Y - 1,
        jmp     circle_loop
slop_negative:                  ; Если наклон отрицательный,
        mov     dx,di
        shl     dx,1
        inc     dx
        add     ax,dx           ; наклон = наклон + 2X + 1,
        jmp     clrcle_loop     ; и Y не изменяется.
exit_main_loop:
        pop     bx
        pop     ax
        pop     si
        ret

current_bank    dw      0       ; Номер текущего банка.
vbe_mode_buffer:                ; Начало буфера данных о видеорежиме.
        end     start
```

В этом примере для наглядности отсутствуют необходимые проверки на поддержку VBE (все прерывания должны возвращать 4Fh в AL), на поддержку видеорежима (атрибут видеорежима в первом байте буфера, заполняемого подфункцией 02) или на объем видеопамяти (должно быть как минимум 2 Мб) и на другие ошибки (все прерывания должны возвращать 0 в AH).

Для вывода точки на экран используется выражение типа

```
номер_банка = номер_строки x байт_в_строке / байт_в_банке
смещение = номер_строки x байт_в_строке MOD байт_в_банке
```

Но так как и число байтов в строке, и число байтов в банке являются степенями двойки, умножение, деление и вычисление остатка от деления можно заменить более быстрыми операциями сдвига, как это сделано в процедуре putpixel16b.

Переключение банков всегда отнимает значительное время, так что по возможности программированием для SVGA-режимов лучше всего заниматься в 32-битном режиме с линейным кадровым буфером, например используя DOS-расширители, как показано в разделе 6.4.

4.6. Работа с мышью

Все общение с мышью в DOS выполняется через прерывание 33h, обработчик которого устанавливает драйвер мыши, загружаемый обычно при запуске системы. Современные драйверы поддерживают около 60 функций, позволяющих настраивать разрешение мыши, профили ускорений, виртуальные координаты, дополнительные обработчики событий и т. п. Большинство этих функций требуются редко, сейчас рассмотрим основные.

INT 33h, AX = 0: Инициализация мыши
Вход: AX - 0000h
Выход: AX = 0000h, если мышь или драйвер мыши не установлены
AX = OFFFFh, если драйвер и мышь установлены
BX = число кнопок:
0002 или OFFFFh - две
0003 - три
0000 — другое количество
Выполняется аппаратный и программный сброс мыши и драйвера.

INT 33h, AX = 1: Показать курсор
Вход: AX = 0001h

INT 33h, AX = 2: Спрятать курсор
Вход: AX - 0002h

Драйвер мыши поддерживает внутренний счетчик, управляющий видимостью курсора мыши. Функция 2 уменьшает значение счетчика на единицу, а функция 1 увеличивает его, но только до значения 0. Если значение счетчика - отрицательное число, он спрятан, если ноль - показан. Это позволяет процедурам, использующим прямой вывод в видеопамять, вызывать функцию 2 в самом начале и 1 в самом конце, не заботясь о том, в каком состоянии был курсор мыши у вызвавшей эту процедуру программы.

INT 33h, AX = 3: Определить состояние мыши
Вход: AX = 0003h
Выход: BX = состояние кнопок:
бит 0: нажата левая кнопка
бит 1: нажата правая кнопка
бит 2: нажата средняя кнопка

CX = X-координата
DX = Y-координата

Возвращаемые координаты совпадают с координатами пикселов соответствующей точки на экране в большинстве графических режимов, кроме 04, 05, ODh, 13h, где X-координату мыши нужно разделить на 2, чтобы получить номер столбца соответствующей точки на экране. В текстовых режимах обе координаты надо разделить на 8 для получения номера строки и столбца соответственно.

В большинстве случаев эта функция не используется в программах, так как для того, чтобы реагировать на нажатие кнопки или перемещение мыши в заданную область, требуется вызывать это прерывание постоянно, что приводит к трате процессорного времени. Функции 5 (определить положение курсора при последнем нажатии кнопки), 6 (определить положение курсора при последнем отпускании кнопки) и 0Bh (определить расстояние, пройденное мышью) могут помочь оптимизировать работу программы, самостоятельно «следящей» за всеми передвижениями мыши, но гораздо эффективнее указать драйверу контролировать ее передвижения (чем он, собственно, и занимается постоянно) и передавать управление в программу, как только выполнится заранее определенное условие, например пользователь нажмет на левую кнопку мыши. Такой сервис обеспечивает функция 0Ch - установить обработчик событий.

INT 33h,AX = 0Ch: Установить обработчик событий
Вход: AX = 000Ch
ES:DX = адрес обработчика
CX = условие вызова
бит 0: любое перемещение мыши
бит 1: нажатие левой кнопки
бит 2: отпускание левой кнопки
бит 3: нажатие правой кнопки
бит 4: отпускание правой кнопки
бит 5: нажатие средней кнопки
бит 6: отпускание средней кнопки
CX = 0000h - отменить обработчик

Обработчик событий должен быть оформлен, как дальняя процедура (то есть завершаться командой RETF). На входе в процедуру обработчика AX содержит условие вызова, BX - состояние кнопок, CX, DX - X- и Y-координаты курсора, SI, DI - счетчики последнего перемещения по горизонтали и вертикали (единицы измерения для этих счетчиков - мики, 1/200 дюйма), DS - сегмент данных драйвера мыши. Перед завершением программы установленный обработчик событий должен быть обязательно удален (вызов функции 0Ch с CX = 0), так как иначе при первом же выполнении условия управление будет передано по адресу в памяти, с которого начинался обработчик.

Функция 0Ch используется так часто, что у нее появилось несколько модификаций - функция 14h, дающая возможность установить одновременно три обработчика с разными условиями, и функция 18h, также позволяющая установить

три обработчика и включающая в условие вызова состояние клавиш **Shift, Ctrl** и **Alt.** Воспользуемся обычной функцией 0Ch, чтобы написать простую программу для рисования.

```
; mousedr.asm
; Рисует на экране прямые линии, оканчивающиеся в позициях, которые указываются мышью.
;
        .model  tiny
        .code
        org     100h                    ; COM-файл.
        .186                            ; Для команды shr cx,3.
start:
        mov     ax,12h
        int     10h                     ; Видеорежим 640x480.
        mov     ax,0                    ; Инициализироватьмышь.
        int     33h
        mov     ax,1                    ; Показать курсор мыши.
        int     33h

        mov     ax,000Ch                ; Установить обработчик событий мыши.
        mov     cx,0002h                ; Событие - нажатие левой кнопки.
        mov     dx,offset handler       ; ES:DX - адрес обработчика.
        int     33h

        mov     ah,0                    ; Ожидание нажатия любой клавиши.
        int     16h
        mov     ax,000Ch
        mov     cx,0000h                ; Удалить обработчик событий мыши.
        int     33h
        mov     ax,3                    ; Текстовый режим.
        int     10h
        ret                             ; Конец программы.

; Обработчик событий мыши: при первом нажатии выводит точку на экран,
; при каждом дальнейшем вызове проводит прямую линию от предыдущей
; точки к текущей.

handler:
        push    0A000h
        pop     es                      ; ES - начало видеопамяти.
        push    cs
        pop     ds                      ; DS - сегмент кода и данных этой программы.
        push    cx                      ; CX (X-координата) и DX (Y-координата)
        push    dx                      ; потребуются в конце.

        mov     ax,2                    ; Спрятать курсор мыши перед выводом на экран.
        int     33h

        cmp     word ptr previous_X,-1  ; Если это первый вызов,
        je      first_point             ; только вывести точку.

        call    line_bresenham          ; Иначе - провести прямую.
```

```
exit_handler:
        pop     dx                      ; Восстановить CX и DX
        pop     cx
        mov     previous_X,cx           ; и запомнить их как предыдущие
        mov     previous_Y,dx           ; координаты.

        mov     ax,1                    ; Показать курсор мыши.
        int     33h

        retf                            ; Выход из обработчика - команда RETF.

first_point:
        call    putpixel1b              ; Вывод одной точки (при первом вызове).
        jmp     short exit_handler

; Процедура рисования прямой линии с использованием алгоритма Брезенхама.
; Вход: CX, OX - X, Y конечной точки,
; previous_X,previous_Y - X, Y начальной точки.

line_bresenham:
        mov     ax,cx
        sub     ax,previous_X           ; AX = длина проекции прямой на ось X.
        jns     dx_pos                  ; Если AX отрицательный -
        neg     ax                      ; сменить его знак, причем
        mov     word ptr X_increment, 1 ; координата X при выводе
        jmp     short dx_neg            ; прямой будет расти.
dx_pos: mov     word ptr X_increment, -1 ; Иначе - уменьшаться.

dx_neg: mov     bx,dx
        sub     bx,previous_Y           ; BX = длина проекции прямой на ось Y.
        jns     dy_pos                  ; Если BX отрицательный -
        neg     bx                      ; сменить его знак, причем
        mov     word ptr Y_increment, 1 ; координата Y при выводе
        jmp     short dy_neg            ; прямой будет расти.
dy_pos: mov     word ptr Y_increment,-1 ; Иначе - уменьшаться.
dy_neg:
        shl     ax,1                    ; Удвоить значения проекций,
        shl     bx,1                    ; чтобы избежать работы с полуцелыми числами.

        call    putpixel1b              ; Вывести первую точку (прямая рисуется от
                                        ; CX,DX к previous_X,previous_Y).
        cmp     ax,bx                   ; Если проекция на ось X больше, чем на Y,
        jna     dx_le_dy
        mov     di,ax                   ; DI будет указывать, в какую сторону мы
        shr     di,1                    ; отклонились от идеальной прямой.
        neg     di                      ; Оптимальное начальное значение DI:
        add     di,bx                   ; DI = 2 * dy - dx
cycle:
        cmp     cx,word ptr previous_X  ; Основной цикл выполняется,
        je      exit_bres               ; пока X не станет равно previous_X.
        cmp     di,0                    ; Если DI > 0,
        jl      fractlt0
```

```
        add     dx,word ptr Y_increment  ; перейти к следующему Y
        sub     di,ax                    ; и уменьшить DI на 2 * dx.
fractlt0:
        add     cx,word ptr X_increment  ; Следующий X (на каждом шаге).
        add     di,bx                    ; Увеличить DI на 2 * dy.
        call    putpixel1b               ; Вывести точку.
        jmp     short cycle              ; Продолжить цикл.

dx_le_dy:                                ; Если проекция на ось Y больше, чем на X.
        mov     di,bx
        shr     di,1
        neg     di                       ; Оптимальное начальное значение DI:
        add     di,ax                    ; DI = 2 . dx - dy.
cycle2:
        cmp     dx,word ptr previous_Y   ; Основной цикл выполняется,
        je      exit_bres                ; пока Y не станет равным previous_Y.
        cmp     di,0                     ; Если DI > 0,
        jl      fractlt02
        add     cx,word ptr X_increment  ; перейти к следующему X
        sub     di,bx                    ; и уменьшить DI на 2 * dy.
fractlt02:
        add     dx,word ptr Y_increment  ; Следующий Y (на каждом шаге).
        add     di,ax                    ; Увеличить DI на 2 * dy,
        call    putpixel1b               ; вывести точку,
        jmp     short cycle2             ; продолжить цикл.
exit_bres:
        ret                              ; Конец процедуры.

; Процедура вывода точки на экран в режиме, использующем один бит для
; хранения одного пиксела.
; DX = строка, CX = столбец.
; Все регистры сохраняются.

putpixel1b:
        pusha                    ; Сохранить регистры.
        xor     bx,bx
        mov     ax,dx            ; AX = номер строки.
        imul    ax,ax,80         ; AX = номер строки x число байтов в строке.
        push    cx
        shr     cx,3             ; CX = номер байта в строке.
        add     ax,cx            ; AX = номер байта в видеопамяти.
        mov     di,ax            ; Поместить его в SI и DI для команд
        mov     si,di            ; строковой обработки.

        pop     cx               ; CX снова содержит номер столбца.
        mov     bx,0080h
        and     cx,07h           ; Последние три бита CX =
                                 ; остаток от деления на 8 = номер бита в байте
                                 ; считая справа налево.
        shr     bx,cl            ; Теперь в BL установлен в 1 нужный бит.
```

```
        lods    es:byte ptr some_label   ; AL = байт из видеопамяти.
        or      ax,bx                    ; Установить выводимый бит в 1,
; Чтобы стереть пиксел о экрана, эту команду OR можно заменить на
; not bx
; and ax,bx
; или лучше инициализировать BX не числом 0080h, а числом FF7Fh и использовать
; только and
        stosb                            ; И вернуть байт на место.
        popa                             ; Восстановить регистры.
        ret                              ; Конец.

previous_X      dw      -1               ; Предыдущая X-координата.
previous_Y      dw      -1               ; Предыдущая Y-координата.
Y_increment     dw      -1               ; Направление изменения Y.
X_increment     dw      -1               ; Направление изменения X.
some_label:                              ; Метка, используемая для переопределения
                                         ; сегмента-источника для lods с DS на ES.
        end     start
```

Алгоритм Брезенхама, использованный в нашей программе, является самым распространенным алгоритмом рисования прямой. Существуют, конечно, и более эффективные, например алгоритм Цаолинь Ву, работающий по принципу конечного автомата, но алгоритм Брезенхама стал стандартом де-факто.

Реализовать этот алгоритм можно гораздо быстрее при помощи самомодифицирующегося кода, то есть после проверки на направление прямой в начале алгоритма вписать прямо в дальнейший текст программы команды INC CX, DEC CX, INC DX и DEC DX вместо команд сложения этих регистров с переменными X_increment и Y_increment. Самомодифицирующийся код часто применяется при программировании для DOS, но в большинстве многозадачных систем текст программы загружается в область памяти, защищенную от записи, вот почему в последнее время зона применения этого подхода становится ограниченной.

4.7. Другие устройства

4.7.1 *Системный таймер*

Начиная с IBM AT, персональные компьютеры содержат два устройства для управления процессами - часы реального времени (RTC) и собственно системный таймер. Часы реального времени получают питание от аккумулятора на материнской плате и работают даже тогда, когда компьютер выключен. Это устройство можно применять для определения/установки текущих даты и времени, установки будильника с целью выполнения каких-либо действий и для вызова прерывания IRQ8 (INT 4Ah) каждую миллисекунду. Системный таймер используется одновременно для управления контроллером прямого доступа к памяти, для управления динамиком и как генератор импульсов, вызывающий прерывание IRQ0 (INT 8h) 18,2 раза в секунду. Таймер предоставляет богатые возможности

для препрограммирования на уровне портов ввода-вывода, но на уровне DOS и BIOS часы реального времени и системный таймер используются только как средство определения/установки текущего времени и организации задержек.

Функция DOS 2Ah: Определить дату
Вход: AH = 2Ah
Выход: CX = год (1980-2099)
DH = месяц
DL = день
AL = день недели (0 - воскресенье, 1 - понедельник...)

Функция DOS 2Ch: Определить время
Вход: AH = 2Ch
Выход: CH = час
CL = минута
DH = секунда
DL = сотая доля секунды

Эта функция использует системный таймер, поэтому время изменяется только 18,2 раза в секунду и число в DL увеличивается сразу на 5 или 6.

Функция DOS 2Bh: Установить дату
Вход: AH = 2Bh
CX - год (1980 - 2099)
DH = месяц
DL = день
Выход: AH = FFh, если введена несуществующая дата; AH = OOh, если дата установлена

Функция DOS 2Dh: Установить время
Вход: AH = 2Dh
CH = час
CL = минута
DH = секунда
DL = сотая доля секунды
Выход: AL = FFh, если введено несуществующее время, и AL = 00, если время установлено

Функции 2Bh и 2Dh устанавливают одновременно как внутренние часы DOS, которые управляются системным таймером и обновляются 18,2 раза в секунду, так и часы реального времени. BIOS позволяет управлять часами напрямую.

INT 1Ah AH = 04h: Определить дату RTC
Вход: AH = 04h
Выход: CF = 0, если дата прочитана
CX = год (в формате BCD, то есть 2001h для 2001-го года)
DH = месяц (в формате BCD)
DL = день (в формате BCD)

CF = 1, если часы не работают или попытка чтения пришлась на момент обновления

INT 1Ah AH = 02k. Определить время RTC
Вход: AH = 02h
Выход: CF = 0, если время прочитано
CH = час (в формате BCD)
CL = минута (в формате BCD)
DH = секунда (в формате BCD)
DL = 01h, если действует летнее время; 00h, если нет
CF = 1, если часы не работают или попытка чтения пришлась на момент обновления

INT 1Ah AH = 05h: Установить дату RTC
Вход: AH = 05h
CX = год (в формате BCD)
DH = месяц (в формате BCD)
DL = день (в формате BCD)

INT 1Ah AH = 03h: Установить время RTC
Вход: AH = 03h
CH = час (в формате BCD)
CL = минута (в формате BCD)
DH = секунда (в формате BCD)
DL = Olh, если используется летнее время, 0 — если нет

Кроме того, BIOS позволяет использовать RTC для организации будильников и задержек.

INT 1Ah AH = 06h: Установить будильник
Вход: AH = 06h
CH - час (BCD)
CL = минута (BCD)
DH = секунда (BCD)
Выход: CF = 1, если произошла ошибка (будильник уже установлен или прерывание вызвано в момент обновления часов);
CF = 0, если будильник установлен

Теперь каждые 24 часа, когда время совпадет с заданным, часы реального времени вызовут прерывание IRQ8 (INT 4Ah), которое должна обрабатывать установившая будильник программа. Если при вызове CH = 0FFh, CL = 0FFh, а DH = OOh, то будильник начнет срабатывать раз в минуту.

INT 1Ah AH = 07: Отменить будильник
Вход: AH - 07h

Эта функция позволяет отменить будильник, например для того, чтобы установить его на другое время.

BIOS отслеживает каждый отсчет системного таймера с помощью своего обработчика прерывания IRQ0 (INT 8h) и увеличивает на 1 значение 32-битного счетчика, который располагается в памяти по адресу OOOOh:046Ch, причем во время переполнения этого счетчика байт по адресу 0000h:0470h увеличивается на 1.

INT 1Ah AH = 00h: Узнать значение счетчика времени
Вход: AH = 00h
Выход: CX:DX - значение счетчика
AL = байт переполнения счетчика

INT 1Ah AH = 01h: Изменить значение счетчика времени
Вход: AH = 01h
CX:DX - значение счетчика

Программа может считывать значение этого счетчика в цикле (через прерывание или просто командой MOV) и организовывать задержки, например, пока счетчик не увеличится на 1. Но так как этот счетчик использует системный таймер, минимальная задержка будет равна приблизительно 55 мкс. Частоту таймера можно изменить, программируя его на уровне портов, но BIOS предоставляет для этого специальные функции.

INT 15h AH = 86h: Формирование задержки
Вход: AH = 86h
CX:DX - длительность задержки в микросекундах (миллионных долях секунды!)
Выход: AL = маска, записанная обработчиком в регистр управления прерываниями
CF = 0, если задержка выполнена
CF = 1, если таймер был занят

Если нужно запустить счетчик времени и продолжить выполнение программы, можно воспользоваться еще одной функцией.

INT 15h AH = 83h: Запуск счетчика времени
Вход: AH = 83h
AL = 0 - запустить счетчик
CX:DX - длительность задержки в микросекундах
ES:BX - адрес байта, старший бит которого по окончании работы счетчика будет установлен в 1
AL = 1 - прервать счетчик

Минимальный интервал для этих функций на большинстве систем обычно составляет около 1000 микросекунд. Воспользуемся функцией организации задержки для небольшой интерактивной игры:

```
; worm.asm
; Игра "Питон" (или "Змея", или "Червяк"). Управление осуществляется
; курсорными клавишами. Питон погибает, если он выходит за верхнюю
; или нижнюю границу экрана либо самопересекается.
```

```
        .model tiny
        .code
        .186                                    ; Для, команды push 0A00h.
        org     100h                            ; COM-файл.
start:
        mov     ax,cs                           ; Текущий сегментный адрес плюс
        add     ax,1000h                        ; 1000h = следующий сегмент,
        mov     ds,ax                           ; который будет использоваться
                                                ; для адресов головы и хвоста.
        push    0A000h                          ; 0A000h - сегментный адрес
        pop     es                              ; видеопамяти (в ES).
        mov     ax,13h                          ; Графический режим 13h.
        int     10h

        mov     di,320*200
        mov     cx,600h                         ; Заполнить часть видеопамяти,
                                                ; остающуюся за пределами
        rep     stosb                           ; экрана, ненулевыми значениями
                                                ; (чтобы питон не смог выйти за
                                                ; пределы экрана).
        xor     si,si                           ; Начальный адрес хвоста в DS:SI.
        mov     bp,10                           ; Начальная длина питона - 10.
        jmp     init_food                       ; Создать первую еду.
main_cycle:
; Использование регистров в этой программе:
; AX - различное.
; BX - адрес головы, хвоста или еды на экране.
; CX - 0 (старшее слово числа микросекунд для функции задержки).
; DX - не используется (модифицируется процедурой random).
; DS - сегмент данных программы (следующий после сегмента кода).
; ES - видеопамять.
; DS:DI - адрес головы.
; DS:SI - адрес хвоста.
; BP - добавочная длина (питон растет, пока BP > 0; BP уменьшается на каждом
; шаге, пока не станет нулем).

        mov     dx,20000                        ; Пауза - 20 000 мкс.
        mov     ah,86h                          ; (CX = 0 после REP STOSB
                                                ; и больше не меняется)
        int     15h                             ; Задержка.

        mov     ah,1                            ; Проверка состояния клавиатуры.
        int     16h
        jz      short no_keypress               ; Если клавиша нажата -
        xor     ah,ah                           ; AH = 0 - считать скан-код
        int     16h                             ; нажатой клавиши в AH.
        cmp     ah,48h                          ; Если это стрелка вверх,
        jne     short not_up
        mov     word ptr cs:move_direction,-320 ; изменить
                                                ; направление движения на "вверх".
```

```
not_up:
        cmp     ah,50h                  ; Если это стрелка вниз,
        jne     short not_down
        mov     word ptr cs:move_direction,320 ; изменить
                                        ; направление движения на "вниз".
not_down:
        cmp     ah,4Bh                  ; Если это стрелка влево,
        jne     short  not_left
        mov     word ptr cs:move_direction,-1 ; изменить
                                        ; направление движения на "влево".
not_left:
        cmp     ah,4Dh                  ; Если это стрелка вправо,
        jne     short no_keypress
        mov     word ptr cs:move_direction,1 ; изменить
                                        ; направление движения на "вправо".
no_keypress:
        and     bp,bp                   ; Если питон растет (BP > 0),
        jnz     short advance_head      ; пропустить стирание хвоста.
        lodsw                           ; Иначе: считать адрес хвоста из
                                        ; DS:SI в AX и увеличить SI на 2.
        xchg    bx,ax
        mov     byte ptr es:[bx],0      ; Стереть хвост на экране.
        mov     bx,ax
        inc     bp                      ; Увеличить BP, чтобы следующая
                                        ; команда вернула его в 0.
advance_head:
        dec     bp                      ; Уменьшить BP, так как питон
                                        ; вырос на 1, если стирание хвоста
                                        ; было пропущено, или чтобы вернуть его
                                        ; в 0 - в другом случае.
        add     bx,word ptr cs:move_direction
                                        ; bx = следующая координата головы.
        mov     al,es:.[bx]             ; Проверить содержимое экрана ь точке
                                        ; с этой координатой.
        and     al,al                   ; Если там ничего нет,
        jz      short  move_worm        ; передвинуть голову.
        cmp     al,0Dh                  ; Если там еда,
        je      short  grow_worm        ; увеличить длину питона.
        mov     ax,3                    ; Иначе - питон умер,
        int     10h                     ; Перейти в текстовый режим
        retn                            ; и завершить программу.

move_worm:
        mov     [di],bx                 ; Поместить адрес головы в DS:DI
        inc     di
        inc     di                      ; и увеличить DI на 2.
        mov     byte ptr es:[bx],09     ; Вывести точку на экран.
        cmp     byte ptr cs:eaten_food,1 ; Если предыдущим
                                        ; ходом была съедена еда -
```

```
        je      if_eaten_food           ; создать новую еду.
        jmp     short main_cycle        ; Иначе - продолжить основной цикл.

grow_worm:
        push    bx                      ; Сохранить адрес головы.
        mov     bx,word ptr cs:food_at  ; BX - адрес еды,
        xor     ax,ax                   ; AX = 0.
        call    draw_food               ; Стереть еду.
        call    random                  ; AX - случайное число.
        and     ax,3Fh                  ; AX - случайное число от 0 до 63.
        mov     bp,ax                   ; Это число будет добавкой
                                        ; к длине питона.
        mov     byte ptr cs:eaten_food,1 ; Установить флаг
                                        ; для генерации еды на следующем ходе.
        pop     bx                      ; Восстановить адрес головы BX.
        jmp     short move_worm         ; Перейти к движению питона.

if_eaten_food:                          ; Переход сюда, если еда была съедена.
        mov     byte ptr cs:eaten_food,0 ; Восстановить флаг.
init_food:                              ; Переход сюда в самом начале.
        push    bx                      ; Сохранить адрес головы.
raake_food:
        call    random                  ; AX - случайное число.
        and     ax,0FFFEh               ; AX - случайное четное число.
        mov     bx,ax                   ; BX - новый адрес для еды.
        xor     ax,ax
        cmp     word ptr es:[bx],ax     ; Если по этому адресу
                                        ; находится тело питона,
        jne     make_food               ; еще раз сгенерировать случайный адрес.
        cmp     word ptr es:[bx+320],ax ; Если на строку ниже
                                        ; находится тело питона -
        jne     make_food               ; то же самое.
        mov     word ptr cs:food_at,bx  ; Поместить новый адрес
                                        ; еды в food_at,
        mov     ax,0D0Dh                ; цвет еды в AX и
        call    draw_food               ; нарисовать еду на экране.
        pop     bx
        jmp     main_cycle

; Процедура draw_food.
; Изображает четыре точки на экране - две по адресу BX и две на следующей
; строке. Цвет первой точки из пары - AL, второй - AH.

draw_food:
        mov     es:[bx],ax
        mov     word ptr es:[bx+320],ax
        retn

; Генерация случайного числа.
; Возвращает число в AX, модифицирует DX.

random: mov     ax,word ptr cs:seed
```

```
        mov     dx,8E45h
        mul     dx
        inc     ax
        mov     cs:word ptr seed,ax
        retn

; Переменные.

eaten_food      db      0
move_direction  dw      1       ; Направление движения: 1 - вправо,
                                ; -1 - влево, 320 - вниз, -320 - вверх.
seed:                           ; Это число хранится за концом программы,
food_at equ     seed+2          ; а это - за предыдущим.
        end     start
```

4.7.2. *Последовательный порт*

Каждый компьютер обычно оборудован, по крайней мере, двумя последовательными портами, которые чаще всего используются для подключения мыши и модема, а также других дополнительных устройств или соединения компьютеров между собой. Для работы с устройствами, подключенными к портам, такими как мышь, применяются драйверы, которые общаются с последовательным портом непосредственно на уровне дортов ввода-вывода и предоставляют программам некоторый набор функций более высокого уровня, так что прямая работа с последовательными портами оказывается необходимой только при написании таких драйверов, работе с нестандартными устройствами или с модемами.

DOS всегда инициализирует первый порт COM1 как 2400 бод, 8N1 (8 бит в слове, 1 стоп-бит, четность не проверяется) и связывает с ним устройство STDAUX, куда функциями 3 и 4 можно записывать и откуда считывать один байт.

Функция DOS 03h: Считать байт из STDAUX
Вход: AH = 03h
Выход: AL = считанный байт

Функция DOS 04h: Записать байт в STDAUX
Вход: AH = 04h
DL = байт

Можно также воспользоваться функциями записи в файл (40h) и чтения из файла (3Fh), поместив в BX число 3, как это показано ранее для вывода на экран.

Несмотря на то что есть возможность изменить установленную DOS скорость работы порта (2400 бод) командой MODE, все равно отсутствие обработки ошибок, буферизации и гибкого управления состоянием порта делает указанные функции DOS практически неприменимыми. BIOS позволяет управлять любым из портов, писать и читать один байт и считывать состояние порта с помощью функций прерывания 14h, однако, так же как и DOS, не допускает инициализацию порта на скорость выше, чем 9600 бод. Таким образом выясняется, что многие программы вынуждены программировать порты напрямую, но, если в системе присутствует драйвер, предоставляющий набор сервисов FOSSIL (такие как XOO

или BNU), то для полноценного буферированного обмена данными с последовательными портами Можно пользоваться лишь функциями прерывания 14h.

INT 14h AH = 04: Инициализация FOSSIL-драйвера
Вход: AH = 04h
DX .= номер порта (0 - для COM1, 1 - для COM2 и т. д.)
Выход: AX = 1954h
BL = максимальный поддерживаемый номер функции
BH = версия спецификации FOSSIL

INT 14h AH = 05: Деинициализация FOSSIL-драйвера
Вход: AH = 05
DX = номер порта (00h - 03h)

INT 14h AH = 00: Инициализация последовательного порта
Вход: AH = 00h
AL = параметры инициализации:
биты 7-5:
000 - 19 200 бод (ПО бод без FOSSIL)
001 - 38 400 бод (150 бод без FOSSIL)
010 - 300 бод
011 - 600 бод
100 - 1 200 бод
101 - 2 400 бод
ПО- 4 800 бод
111 – 9 600 бод
биты 4-3: четность (01 - нечетная, И - четная, 00 или 10 - нет)
бит 2: число стоп-битов (0 - один, 1 - два)
биты 1-0: длина слова (00 - 5, 01 - 6, 10 - 7, 11 - 8)
DX = номер порта (00h - 03h)
Выход: AH = состояние порта
бит 7: тайм-аут
бит 6: буфер вывода пуст (без FOSSIL: регистр сдвига передатчика пуст)
бит 5: в буфере вывода есть место (без FOSSIL: регистр хранения передатчика пуст)
бит 4: обнаружено состояние BREAK
бит 3: ошибка синхронизации
бит 2: ошибка четности
бит 1: ошибка переполнения - данные потеряны
бит 0: в буфере ввода есть данные
AL = состояние модема
бит 7: обнаружена несущая (состояние линии DCD)
бит 6: обнаружен звонок (состояние линии RI)
бит 5: запрос для передачи (состояние линии DSR)
бит 4: сброс для передачи (состояние линии CTS)

бит 3: линия DCD изменила состояние
бит 2: линия RI изменила состояние
бит 1: линия DSR изменила состояние
бит 0: линия CTS изменила состояние

INT 14h AH = 01: Запись символа в последовательный порт
Вход: AH = 01h
AL = символ
DX = номер порта (00h - 03h)
Выход: AH = состояние порта

INT 14h AH - 02: Чтение символа из последовательного порта с ожиданием
Вход: AH - 02h
DX = номер порта
Выход: AH = состояние порта
AL = считанный символ, если бит 7 AH равен нулю (не было тайм-аута)

INT 14h AH = 03: Получить текущее состояние порта
Вход: AH = 03h
DX = номер порта (OOh - 03h)
Выход: AH = состояние линии
AL = состояние модема

Воспользуемся этими функциями, чтобы написать короткую терминальную программу:

```
; term.asm
; Простая терминальная программа для модема на COM2. Выход по Alt-X.
;
        .model  tiny
        .code
        org 100h                ; Начало COM-файла.
start:
        mov     ah,0            ; Инициализировать порт.
        mov     al,11100011b    ; 9600/8n1
        mov     dx,1            ; Порт COM2.
        int     14h

main_loop:
        mov     ah,2
        int     14h             ; Получить байт от модема.
        test    ah,ah           ; Если что-нибудь получено,
        jnz     no_input
        int     29h             ; вывести его на экран.
no_input:                       ; Иначе:
        mov     ah,1
        int     16h             ; проверить, была ли нажата клавиша.
        jz      main_loop       ; Если да:
        mov     ah,8
        int     21h             ; считать ее код (без отображения на экране).
        test    al,al           ; Если это нерасширенный ASCII-код -
```

```
        jnz     send_char       ; отправить его в модем.
        int     21h             ; Иначе получить расширенный ASCII-код.
        cmp     al,2Dh          ; Если это Alt-X
        jne     send_char
        ret                     ; завершить программу.
send_char:
        mov     ah,1
        int     14h             ; Послать введенный символ в модем.
        jmp     short main_loop ; Продолжить основной цикл.

        end     start
```

Этот терминал тратит чрезмерно много процессорного времени на постоянные вызовы прерываний 14h и 16h. Более эффективным оказывается подход, заключающийся в перехвате прерываний от внешних устройств, о котором рассказано далее.

4.7.3. *Параллельный порт*

Параллельные порты используются в первую очередь для подключения принтеров, хотя встречаются и другие устройства, например переносные жесткие диски, которые могут присоединяться к этим портам. Базовые средства DOS и BIOS для работы с параллельными портами аналогичны соответствующим средствам для работы с последовательными портами: DOS инициализирует стандартное устройство PRN, соответствующее первому порту LPT1, которое может быть переопределено командой MODE, и предоставляет прерывание для вывода в это устройство.

Функция DOS 05h: Вывод символа в стандартное устройство PRN
Вход: AH = 05h
DL = символ

Кроме того, можно пользоваться функцией записи в файл или устройство, поместив в BX число 4, соответсвующее устройству PRN. BIOS, в свою очередь, предоставляет базовый набор из трех функций для работы с принтером.

INT 17h, AH = 00: Вывести символ в принтер
Вход: AH = 00h
AL = символ
DX = номер параллельного порта (00 - LPT1, 01 - LPT2, 02 - LPT3)
Выход: AH = состояние принтера:
бит 7: принтер не занят
бит 6: подтверждение
бит 5: нет бумаги
бит 4: принтер в состоянии online
бит 3: ошибка ввода-вывода
бит 0: тайм-аут

INT 17h, AH = 01: Выполнить аппаратный сброс принтера
Вход: AH = 01h
DX = номер порта (OOh - 02h)
Выход: AH = состояние принтера

INT 17h, AH - 02: Получить состояние принтера
Вход: AH - 02h
DX = номер порта (00h - 02h)
Выход: AH = состояние принтера

Например, чтобы распечатать содержимое экрана на принтере, можно написать такую программу:

```
; prtscr.asm
;  Распечатать  содержимое  экрана  на  принтере.
;
        .model  tiny
        .code
        .186                    ; Для команды push 0B800h.
        org     100h            ; Начало COM-файла.
start:
        mov     ah,1
        mov     dx,0            ; Порт  LPT1.
        int     17h             ; Инициализировать принтер.
        cmp     ah,90h          ; Если принтер не готов,
        jne     printer_error   ; выдать сообщение об ошибке.
        push    0B800h          ; Иначе:
        pop     ds              ; DS = сегмент видеопамяти в текстовом режиме,
        xor     si,si           ; SI = 0,
        mov     cx,80*40        ; CX = число символов на экране.
        cld                     ; Строковые операции вперед.
main_loop:
        lodsw                   ; AL - символ, AH - атрибут,  SI = SI + 2.
        mov     ah,0            ; AH - номер функции.
        int     17h             ; Вывод символа из AL на принтер.
        loop    main_loop
        ret                     ; Закончить  программу.

printer_error:
        mov     dx,offset msg   ; Адрес сообщения об ошибке в DS:DX.
        mov     ah,9
        int     21h             ; Вывод строки на экран.
        ret

msg             db      "Принтер на LPT1 находится в состоянии offline или занят$"
        end     start
```

Чтобы распечатать экран в текстовом режиме на LPT1, достаточно всего лишь одной команды INT 05h, что в точности эквивалентно нажатию клавиши **PrtScr**.

4.8. Работа с файлами

Возможно, основная функция DOS в качестве операционной системы - организация доступа к дискам как к набору файлов и директорий. DOS поддерживает только один тип файловой системы - FAT и, начиная с версии 7.0 (Windows 95), его модификацию VFAT с длинными именами файлов. Первоначальный набор

функций для работы *с* файлами, предложенный в MS DOS 1.0, оказался очень неудобным: каждый открытый файл описывался 37-байтной структурой FCB (блок управления файлом), адрес которой требовался для всех файловых операций, а передача данных осуществлялась через структуру данных DTA (область передачи данных). Уже в MS DOS 2.0, вместе с усовершенствованием FAT (например, появлением вложенных директорий), появился набор UNIX-подобных функций работы с файлами, использующих для описания файла всего одно 16-битное число, идентификатор файла или устройства. Все остальные функции работы с файлами используют затем только это число. Первые пять идентификаторов инициализируются системой следующим образом:

0: STDIN - стандартное устройство ввода (обычно клавиатура);
1: STDOUT – стандартное устройство вывода (обычно экран);
2: STDERR — устройство вывода сообщений об ошибках (всегда экран);
3: AUX - последовательный порт (обычно COM1);
4: PRN - параллельный порт (обычно LPT1);

так что функции чтения/записи (а также сброс буферов на диск) файлов можно применять и к устройствам.

4.8.1. Создание и открытие файлов

Функция DOS 3Ch: Создать файл

Вход: AH = 3Ch
CX = атрибут файла
бит 7: файл можно открывать разным процессам в Novell Netware
бит 6: не используется
бит 5: архивный бит (1, если файл не сохранялся)
бит 4: директория (должен быть 0 для функции 3Ch)
бит 3: метка тома (игнорируется функцией 3Ch)
бит 2: системный файл
бит 1: скрытый файл
бит 0: файл только для чтения
DS:DX = адрес ASCIZ-строки с полным именем файла (ASCIZ-строка ASCII-символов, оканчивающаяся нулем)

Выход: CF = 0 и AX = идентификатор файла, если не произошла ошибка
CF = 1 и AX = 03h, если путь не найден
CF = 1 и AX = 04h, если слишком много открытых файлов
CF = 1 и AX = 05h, если доступ запрещен

Если файл уже существует, функция 3Ch все равно открывает его, присваивая ему нулевую длину. Чтобы этого не произошло, следует пользоваться функцией 5Bh.

Функция DOS 3Dh: Открыть существующий файл

Вход: • AH = 3Dh
AL = режим доступа
бит 0: открыть для чтения
бит 1: открыть для записи
биты 2-3: зарезервированы (0)

биты 6-4: режим доступа для других процессов:
000: режим совместимости (остальные процессы также должны открывать этот файл в режиме совместимости)
001: все операции запрещены
010: запись запрещена
011: чтение запрещено
100: запрещений нет
бит 7: файл не наследуется порождаемыми процессами
DS:DX = адрес ASCIZ-строки с полным именем файла
CL = маска атрибутов файлов
Выход: CF = 0 и AX = идентификатор файла, если не произошла ошибка
CF = 1 и AX = код ошибки (02h - файл не найден, 03h - путь не найден, 04h - слишком много открытых файлов, 05h - доступ запрещен, 0Ch — неправильный режим доступа)

Функция DOS 5Bh: Создать и открыть новый файл
Вход: AH = 5Bh
CX = атрибут файла
DS:DX = адрес ASCIZ-строки с полным именем файла
Выход: CF = 0 и AX = идентификатор файла, открытого для чтения/записи в режиме совместимости, если не произошла ошибка
CF = 1 и AX = код ошибки (03h - путь не найден, 04h - слишком много открытых файлов, 05h - доступ запрещен, 50h - файл уже существует)

Функция DOS 5Ah: Создать и открыть временный файл
Вход: AH = 5Ah
CX = атрибут файла
DS:DX = адрес ASCIZ-строки с путем, оканчивающимся символом \, и тринадцатью нулевыми байтами в конце
Выход: CF = 0 и AX = идентификатор файла, открытого для чтения/записи в режиме совместимости, если не произошла ошибка (в строку по адресу DS:DX дописывается имя файла)
CF = 1 и AX = код ошибки (03h - путь не найден, 04h - слишком много открытых файлов, 05h - доступ запрещен)

Функция 5Ah создает файл с уникальным именем, который не является на самом деле временным. Такой файл следует специально удалять, для чего его имя и записывается в строку в DS:DX.

Во всех случаях строка с полным именем файла имеет вид типа

```
filespec        db      'c:\data\filename.ext',0
```

причем, если диск или путь опущены, используются их текущие значения.

Для работы с длинными именами файлов в DOS 7.0 (Windows 95) и Старше используются дополнительные функции, которые вызываются так же, как функция DOS 71h.

Функция LFN 6Ch: Создать или открыть файл с длинным именем
Вход: AX = 716Ch
BX = режим доступа Windows 95
биты 2-0: доступ
000 - только для чтения
001 - только для записи
010 - для чтения и записи
100 - только для чтения, не изменять время последнего обращения к файлу
биты 6-4: доступ для других процессов (см. функцию 3Dh)
бит 7: файл не наследуется порождаемыми процессами
бит 8: данные не буферируются
бит 9: не архивировать файл, если используется архивирование файловой системы (DoubleSpace)
бит 10: использовать число в DI для записи в конце короткого имени файла
бит 13: не вызывать прерывание 24h при критических ошибках
бит 14: сбрасывать буфера на диск после каждой записи в файл
CX = атрибут файла
DX = действие
бит 0: открыть файл (ошибка, если файл существует)
бит 1: заменить файл (ошибка, если файл не существует)
бит 4: создать файл (ошибка, если файл существует)
DS:SI = адрес ASCIZ-строки с именем файла
DI = число, которое будет записано в конце короткого варианта имени файла
Выход: CF = O
AX = идентификатор файла
CX = 1, если файл открыт
CX = 2, если файл создан
CX = 3, если файл заменен
CF = 1, если произошла ошибка
AX = код ошибки (7100h, если функция не поддерживается)

Если функции открытия файлов возвращают ошибку «слишком много открытых файлов» (AX = 4), следует увеличить число допустимых идентификаторов с помощью функции 67h.

Функция DOS 67h: Изменить максимальное число идентификаторов файлов
Вход: AH = 67h
BX = новое максимальное число идентификаторов (20-65 535)
Выход: CF = 0, если не произошла ошибка
CF = 1 и AX = код ошибки, если произошла ошибка (например: 04h, если заданное число меньше, чем количество уже открытых файлов, или 08h, если DOS не хватает памяти для новой таблицы идентификаторов)

Следует помнить, что все дочерние процессы будут наследовать только первые 20 идентификаторов и должны вызывать функцию 67h сами, если им требуется больше.

4.8.2. *Чтение и запись в файл*

Функция DOS 3Fh: Чтение из файла или устройства

Вход: AH = 3Fh
BX = идентификатор
CX = число байтов
DS:DX = адрес буфера для приема данных

Выход: CF = 0 и AX = число считанных байтов, если не было ошибки
CF = 1 и AX = 05h, если доступ запрещен, 06h, если неправильный идентификатор

Если при чтении из файла число фактически считанных байтов в AX меньше, чем заказанное число в CX, то был достигнут конец файла. Каждая следующая операция чтения, так же как и записи, начинается не с начала файла, а с того байта, на котором остановилась предыдущая операция чтения/записи. Если требуется считать (или записать) произвольный участок файла, используют функцию 42h (функция lseek в C).

Функция DOS 42h: Переместить указатель чтения/записи

Вход: AH - 42h
BX = идентификатор
CX:DX = расстояние, на которое надо переместить указатель (со знаком)
AL = перемещение:
0 — от начала файла
1 - от текущей позиции
2 — от конца файла

Выход: CF = 0 и CX:DX = новое значение указателя (в байтах от начала файла), если не произошла ошибка
CF = 1 и AX = 06h, если неправильный идентификатор

Указатель можно установить за реальными пределами файла: в отрицательное число, тогда следующая операция чтения/записи вызовет ошибку; в положительное число, большее длины файла, тогда очередная операция записи увеличит размер файла. Эта функция также часто используется для определения длины файла - достаточно вызвать ее с CX = 0, DX = 0, AL = 2, и в CX:DX будет возвращена длина файла в байтах.

Функция DOS 40k. Запись в файл или устройство

Вход: AH = 40h
BX = идентификатор
CX = число байтов
DS:DX = адрес буфера с данными

Выход: CF = 0 и AX = число записанных байтов, если не произошла ошибка
CF = 1 и AX = 05h, если доступ запрещен; 06h, если неправильный идентификатор

Если при записи в файл указать CX = 0, он будет обрезан по текущему значению указателя. На самом деле происходит запись в буфер DOS, данные из которого сбрасываются на диск во время закрытия файла или если их количество превышает размер сектора диска. Для немедленной очистки буфера можно использовать функцию 68h (функция fflush в C).

Функция DOS 68h: Сброс файловых буферов DOS на диск
Вход: AH = 68h
BX = идентификатор
Выход: CF = 0, если операция выполнена
CF = 1, если произошла ошибка (AX = код ошибки)

Для критических участков программ предпочтительнее использовать более эффективную функцию 0Dh.

Функция DOS ODh: Сброс всех файловых буферов на диск
Вход: AH = ODh
Выход: Никакого

4.8.3. Закрытие и удаление файла

Функция DOS 3Eh: Закрыть файл
Вход: AH = 3Eh
BX = идентификатор
Выход: CF = 0, если не произошла ошибка
CF = 1 и AX = 6, если неправильный идентификатор

Если файл был открыт для записи, все файловые буфера сбрасываются на диск, устанавливается время модификации файла и записывается его новая длина.

Функция DOS 41h: Удаление файла
Вход: AH = 41h
DS:DX = адрес ASCIZ-строки с полным именем файла
Выход: CF = 0, если файл удален
CF = 1 и AH = 02h, если файл не найден; 03h, если путь не найден; 05h, если доступ запрещен

Удалить файл можно только после того, как он будет закрыт, иначе DOS продолжит выполнение записи в несуществующий файл, что может привести к разрушению файловой системы. Функция 41h не позволяет использовать маски (символы * и ? в имени файла) для удаления сразу нескольких файлов, хотя этого можно добиться, вызывая ее через недокументированную функцию 5DOOh. Но, начиная с DOS 7.0 (Windows 95), официальная функция удаления файла способна работать сразу с несколькими файлами.

Функция LFN 41h: Удаление файлов с длинным именем
Вход: AX = 7141h
DS:DX = адрес ASCIZ-строки с длинным именем файла
SI = OOOOh: маски не разрешены и атрибуты в CX игнорируются

SI = 0001h: маски в имени файла и атрибуты в CX разрешены:
CL = атрибуты, которые файлы могут иметь
CH = атрибуты, которые файлы должны иметь

Выход: CF = 0, если файл или файлы удалены
CF = 1 и AX = код ошибки, если произошла ошибка. Код 7100h означает, что функция не поддерживается

4.8.4. Поиск файлов

Найти нужный файл на диске намного сложнее, чем просто открыть его, - для этого требуются две функции при работе с короткими именами (найти первый файл и найти следующий файл) и три - при работе с длинными именами в DOS 7.0 (найти первый файл, найти следующий файл, прекратить поиск).

Функция DOS 4Eh: Найти первый файл

Вход: AH = 4Eh
AL используется при обращении к функции APPEND
CX = атрибуты, которые должен иметь файл (биты 0 (только для чтения) и 5 (архивный бит) игнорируются. Если бит 3 (метка тома) установлен, все остальные биты игнорируются)
DS:DX = адрес ASCIZ-строки с именем файла, которое может включать путь и маски для поиска (символы * и ?)

Выход: CF = 0 и область DTA заполняется данными, если файл найден
CF = 1 и AX = 02h, если файл не найден; 03h, если путь не найден; 12h, если неправильный режим доступа

Вызов этой функции заполняет данными область памяти DTA (область передачи данных), которая начинается по умолчанию со смещения 0080h от начала блока данных PSP (при запуске COM- и EXE-программ сегменты DS и ES содержат сегментный адрес начала PSP), но ее можно переопределить с помощью функции 1Ah.

Функция DOS 1Ah: Установить область DTA

Вход: AH = 1Ah
DS:DX = адрес начала DTA (128-байтный буфер)

Функции поиска файлов заполняют DTA следующим образом:

+00h: байт - биты 0-6: ASCII-код буквы диска
бит 7: диск сетевой
+01h: 11 байт - маска поиска (без пути)
+0Ch: байт - атрибуты для поиска
+0Dh: слово - порядковый номер файла в директории
+OFh: слово - номер кластера начала внешней директории
+11h: 4 байта — зарезервировано
+15h: байт - атрибут найденного файла
+16h: слово - время создания файла в формате DOS:
биты 15-11: час (0-23)
биты 10-5: минута
биты 4-0: номер секунды, деленный на 2 (0-30)

+18h: слово - дата создания файла в формате DOS:
биты 15-9: год, начиная с 1980
биты 8-5: месяц
биты 4-0: день
+1Ah: 4 байта - размер файла
+1Eh: 13 байт - ASCIZ-имя найденного файла с расширением

После того как DTA заполнена данными, для продолжения поиска следует вызывать функцию 4Fh, пока не будет возвращена ошибка.

Функция DOS 4Fh: Найти следующий файл
Вход: AH = 4Fh
DTA - содержит данные от предыдущего вызова функции 4Eh или 4Fh
Выход: CF = 0 и DTA содержит данные о следующем найденном файле, если не произошла ошибка
CF = 1 и AX = код ошибки, если произошла ошибка

В случае с длинными именами файлов (LFN) применяется набор из трех подфункций функции DOS 71h, которые можно использовать, только если запущен IFSmgr (всегда запускается при обычной установке Windows 95, но не запускается, например, с загрузочной дискеты MS DOS 7.0).

Функция LFN 4Eh: Найти первый файл с длинным именем
Вход: AX - 714Eh
CL = атрибуты, которые файл может иметь (биты 0 и 5 игнорируются)
CH = атрибуты, которые файл должен иметь
SI = 0: использовать Windows-формат даты/времени
SI = 1: использовать DOS-формат даты/времени
DS:DX = адрес ASCIZ-строки с маской для поиска (может включать * и ?. Для совместимости маска *.* ищет все файлы в директории, а не только файлы, содержащие точку в имени)
ES:DI = адрес 318-байтного буфера для информации о файле
Выход: CF = 0
AX = поисковый идентификатор
CX = Unicode-флаг:
бит 0: длинное имя содержит подчеркивания вместо непреобразуемых Unicode-символов
бит 1: короткое имя содержит подчеркивания вместо непреобразуемых Unicode-символов
CF = 1, AX = код ошибки, если произошла ошибка (7100h - функция не поддерживается)

Если файл, подходящий под маску и атрибуты поиска, найден, область данных по адресу ES:DI заполняется следующим образом:

+00h: 4 байта - атрибуты файла
биты 0-6: атрибуты файла DOS
бит 8: временный файл

+04h: 8 байт - время создания файла
+0Ch: 8 байт - время последнего доступа к файлу
+14h: 8 байт — время последней модификации файла
+1Ch: 4 байта - старшее двойное слово длины файла
+20h: 4 байта - младшее двойное слово длины файла
+24h: 8 байт - зарезервировано
+2Ch: 260 байт - ASCIZ-имя файла длинное
+130h: 14 байт - ASCIZ-имя файла короткое

Причем даты создания/доступа/модификации записываются в одном из двух форматов, в соответствии со значением SI при вызове функции. Windows-формат - 64-битное число 100-наносекундных интервалов с 1 января 1601 года; если используется DOS-формат - в старшее двойное слово записывается DOS-дата, а в младшее - DOS-время.

Функция LFN 4Fh: Найти следующий файл
Вход: AX = 714Fh
BX = поисковый идентификатор (от функции 4Eh)
SI = формат даты/времени
ES:DI = адрес буфера для информации о файле
Выход: CF = 0 и CX = Unicode-флаг, если следующий файл найден
CF = 1, AX = код ошибки, если произошла ошибка (7100h - функция не поддерживается)

Функция LFN A1h: Закончить поиск файла
Вход: AX = 71A1h
BX = поисковый идентификатор
Выход: CF = 0, если операция выполнена
CF = 1 и AX = код ошибки, если произошла ошибка (7100h — функция не поддерживается)

В качестве примера, использующего многие функции работы с файлами, рассмотрим программу, заменяющую русские буквы Н латинскими Н во всех файлах с расширением .TXT в текущей директории (такая замена требуется для каждого текста, пересылающегося через сеть Fidonet, программное обеспечение которой воспринимает русскую букву Н как управляющий символ).

```
; fidoh.asm
; Заменяет русские "Н" латинскими "H" во всех файлах с расширением .TXT
; в текущей директории.
;
        .model  tiny
        .code
        org     100h                    ; COM-файл
start:
        mov     ah,4Eh                  ; Поиск первого файла.
        xor     cx,cx                   ; Не системный, не директория и т. д.
        mov     dx,offset filespec      ; Маска для поиска в DS:DX.
```

```
file_open:
        int     21h
        jc      no_more_files           ; Если CF = 1 - файлы кончились.

        mov     ax,3D02h                ; Открыть файл для чтения и записи.
        mov     dx,80h+1Eh              ; Смещение DTA + смещение имени файла
        int     21h                     ; от начала DTA.
        jc      not_open                ; Если файл не открылся - перейти
                                        ; к следующему.

        mov     bx,ax                   ; Идентификатор файла в BX.
        mov     cx,1                    ; Считывать один байт.
        mov     dx,offset buffer        ; Начало буфера - в DX.
read_next:
        mov     ah,3Fh                  ; Чтение файла.
        int     21h
        jc      find_next               ; Если ошибка - перейти к следующему.
        dec     ax                      ; Если AX = 0 - файл кончился -
        js      find_next               ; перейти к следующему.
        cmp     byte ptr buffer,8Dh     ; Если не считана русская "Н",
        jne     read_next               ; считать следующий байт.
        mov     byte ptr buffer,48h     ; Иначе - записать в буфер
                                        ; латинскую букву "H".

        mov     ax,4201h                ; Переместить указатель файла от текущей
        dec     cx                      ; позиции назад на 1.
        dec     cx                      ; CX = OFFFFh,
        mov     dx,cx                   ; DX = OFFFFh.
        int     21h

        mov     ah,40h                  ; Записать в файл.
        inc     cx
        inc     cx                      ; Один байт (CX = 1)
        mov     dx,offset buffer        ; из буфера в DS:DX.
        int     21h
        jmp     short read_next         ; Считать следующий байт.

find_next:
        mov     ah,3Eh                  ; Закрыть предыдущий файл.
        int     21h
not_open:
        mov     ah,4Fh                  ; Найти следующий файл.
        mov     dx,80h"                 ; Смещение DTA от начала PSP.
        jmp     short file_open

no_more_files:                          ; Если файлы кончились,
        ret                             ; выйти из программы.

filespec        db      "*.txt",0       ; Маска для поиска.
buffer          label   byte            ; Буфер для чтения/записи -
        end     start                   ; за концом программы.
```

4.8.5. Управление файловой системой

Начиная с MS DOS 2.0 файловая система организована в виде директорий. Поиск файлов выполняется только в пределах текущей директории, а создание и удаление файлов неприемлемы к директориям, хотя на самом низком уровне директория - тот же файл, в атрибуте которого бит 4 установлен в 1 и который содержит список имен вложенных файлов, их атрибутов и физических адресов на диске.

Функция DOS 39h: Создать директорию
Вход: AH = 39h
DS:DX = адрес ASCIZ-строки с путем, в котором все директории, кроме последней, существуют. Для DOS 3.3 и более ранних версий длина всей строки не должна превышать 64 байта
Выход: CF = 0, если директория создана
CF = 1 и AX = 3, если путь не найден; *5,* если доступ запрещен

Функция LFN39h: Создать директорию с длинным именем
Вход: AX - 7139h
DS:DX = адрес ASCIZ-строки с путем
Выход: CF = 0, если директория создана
CF = 1 и AX = код ошибки (7100h, если функция не поддерживается)

Функция DOS 3Ah: Удалить директорию
Вход: AH = 3Ah
DS:DX = адрес ASCIZ-строки с путем, где последняя директория будет удалена (только если она пустая, не является текущей, не занята командой SUBST)
Выход: CF = 0, если директория удалена
CF = 1 и AX = 3, если путь не найден; 5, если доступ запрещен; 10h, если удаляемая директория - текущая

Функция LFN 3Ah: Удалить директорию с длинным именем
Вход: AX = 713Ah
DS:DX = адрес строки с путем
Выход: CF = 0, если директория удалена, иначе CF = 1 и AX = код ошибки

Функция DOS 47k. Определить текущую директорию
Вход: AH = 47h
DL = номер диска (00h - текущий, 01h – A и т. д.)
DS:SI = 64-байтный буфер для текущего пути (ASCIZ-строка без имени диска, первого и последнего символа \)
Выход: CF = 0 и AX = 0100h, если операция выполнена
CF = 1 и AX = 0Fh, если указан несуществующий диск

Функция LFN 47h: Определить текущую директорию с длинным именем
Вход: AX = 7147h
DL = номер диска
DS:SI = буфер для пути (ASCIZ-строка без имени диска, первого и последнего символа \. Необязательно содержит лишь длинные

имена - возвращается тот путь, который использовался при последней смене текущей директории.)
Выход: CF = 0, если директория определена, иначе CF = 1 и AX = код ошибки

Функция DOS 3Bh: Сменить директорию
Вход: AH = 3Bh
DS:DX = адрес 64-байтного ASCIZ-буфера с путем, который станет текущей директорией
Выход: CF = 0, если директория изменена, иначе CF = 1 и AX = 3 (путь не найден)

Функция LFN 3B: Сменить директорию с длинным именем
Вход: AX = 713Bh
DS:DX = адрес ASCIZ-буфера с путем
Выход: CF = 0, если директория изменена, иначе CF = 1 и AX = код ошибки

Перед работой с любыми функциями LFN следует один раз вызвать подфункцию 0A0h, чтобы определить размеры буферов для имен файлов и путей.

Функция LFN OAOh: Получить информацию о разделе файловой системы VFAT
Вход: AX - 71A0h
DS:DX = адрес ASCIZ-строки с именем раздела (например: db "C:\",0)
ES:DI = адрес буфера для имени файловой системы (FAT, NTFS, CDFS)
CX = размер буфера в ES:DI (обычно 32 байта)
Выход: CX = O, AX = 0000h или 0200h
BX = флаги файловой системы:
бит 0: функции поиска учитывают регистр символов
бит 1: регистр символов сохраняется для имен директорий
бит 2: используются символы Unicode
бит 14: поддерживаются функции LFN
бит 15: включено сжатие раздела (DoubleSpace)
CX = максимальная длина имени файла (обычно 255)
DX = максимальная длина пути (обычно 260) в Windows 95 SP1 возвращает 0000h для CD-ROM
CF = 1 и AX = код ошибки, если произошла ошибка (7100h, если функция не поддерживается)

Кроме того, при вызове любой функции LFN следует устанавливать CF в 1 для совместимости с ранними версиями DOS. Старые версии DOS не изменяли CF, так что в результате, если функция не поддерживается, CF останется равным 1.

4.9. Управление памятью

4.9.1. Обычная память

До сих пор, если требовалось создать массив данных в памяти, мы просто обращались к памяти за концом программы, считая, что там имеется еще хотя бы 64 Кб свободной памяти. Разумеется, как и во всех операционных системах, в DOS есть средства управления распределением памяти - выделение блока

(аналог стандартной функции языка C malloc), изменение его размеров (аналог realloc) и освобождение (free).

Функция DOS 48h: Выделить память
Вход: AH = 48h
BX = размер блока в 16-байтных параграфах
Выход: CF = 0, если блок выделен
AX = сегментный адрес выделенного блока
CF = 1, если произошла ошибка:
AX — 7 — блоки управления памятью разрушены
AX = 8 – недостаточно памяти:
BX = размер максимального доступного блока

Эта функция с большим значением в BX (обычно OFFFFh) используется для определения размера самого большого доступного блока памяти.

Функция DOS 49h: Освободить память
Вход: AH = 49h
ES = сегментный адрес освобождаемого блока
Выход: CF = 0, если блок освобожден
CF = 1, AX = 7, если блоки управления памятью разрушены;
AX = 9, если в ES содержится неверный адрес

Эта функция не позволит освободить блок памяти, которым текущая программа не владеет, но с помощью функции DOS 50h (AX = 50h, BX = сегментный адрес PSP процесса) программа может «притвориться» любым другим процессом.

Функция DOS 4Ah: Изменить размер блока памяти
Вход: AH = 4Ah
BX = новый размер в 16-байтных параграфах
ES = сегментный адрес модифицируемого блока
Выход: CF = 1, если при выполнении операции произошла ошибка
AX = 7, если блоки управления памятью разрушены
AX = 8, если не хватает памяти (при увеличении)
AX = 9, если ES содержит неверный адрес
BX = максимальный размер, доступный для этого блока

Если для увеличения блока не хватило памяти, DOS расширяет его до возможного предела.

При запуске COM-программы загрузчик DOS выделяет самый большой доступный блок памяти для этой программы, так что при работе с основной памятью эти функции требуются редко (в основном для того, чтобы сократить выделенный программе блок памяти до минимума перед загрузкой другой программы), но уже в MS DOS 5.0 и далее с помощью этих же функций можно выделять память в областях UMB - неиспользуемых участках памяти выше 640 Кб и ниже 1 Мб, для чего требуется сначала подключить UMB к менеджеру памяти и изменить стратегию выделения памяти с помощью функции DOS 58h.

4.9.2. *Область памяти UMB*

Функция DOS 58k. Считать/изменить стратегию выделения памяти

Вход: AH = 58h
AL = 00h - считать стратегию
AL = 01h - изменить стратегию
BX = новая стратегия
биты 2-0:
00 - первый подходящий блок
01 - наиболее подходящий блок
11 - последний подходящий блок
биты 4-3:
00 - обычная память
01 - UMB (DOS 5.0+)
10 - UMB, затем обычная память (DOS 5.0+)
AL = 02h - считать состояние UMB
AL = 03h - установить состояние UMB
BX = новое состояние: 00 - не используются, 01 - используются

Выход: CF = O, AX = текущая стратегия для AL = 0, состояние UMB для AL = 2
CF = 1, AX = 01h, если функция не поддерживается (если не запущен менеджер памяти (например, EMM386) или нет строки DOS = UMB в CONFIG.SYS)

Если программа изменяла стратегию выделения памяти или состояние UMB, она обязательно должна их восстановить перед окончанием работы.

4.9.3. *Область памяти HMA*

Область памяти от OFFFFh:0010h (конец первого мегабайта) до OFFFFh:OFFFFh (конец адресного пространства в реальном режиме), 65 520 байт, может использоваться на компьютерах, начиная с 80286. Доступ к этой области осуществляется с помощью спецификации XMS, причем вся она выделяется целиком одной программе. Обычно, если загружен драйвер HIMEM.SYS и если в файле CONFIG.SYS присутствует строка DOS = HIGH, DOS занимает эту область, освобождая почти 64 Кб в основной памяти. При этом ОС может оставить небольшой участок HMA (16 Кб или меньше) для пользовательских программ, которые обращаются к нему с помощью недокументированной функции мультиплексора 4Ah.

INT 2Fh, AX = 4A01h: Определить размер доступной части HMA (DOS 5.0+)

Вход: AX = 4A01h

Выход: BX = размер доступной части HMA в байтах, 0000h, если DOS не в HMA
ES:DI = адрес начала доступной части HMA (OFFFFh:OFFFFh, если DOS не в HMA)

INT 2Fh, AX = 4A02h: Выделить часть HMA (DOS 5.0+)

Вход: AX = 4A02h
BX = размер в байтах

Выход: ES:DI = адрес начала выделенного блока
BX = размер выделенного блока в байтах

В версиях DOS 5.0 и 6.0 нет функций освобождения выделенных таким образом блоков HMA. В DOS 7.0 (Windows 95) выделение памяти в HMA было организовано аналогично выделению памяти в обычной памяти и UMB, с функциями изменения размера и освобождения блока.

INT 2Fh, AX = 4A03h: Управление распределением памяти в HMA (DOS 7.0+)
Вход: AX = 4A03h
DL = 0 - выделить блок (BX = размер в байтах)
DL = 1 - изменить размер блока (ES:DI = адрес, BX = размер)
DL = 2 - освободить блок (ES:DI = адрес)
CX = сегментный адрес владельца блока
Выход: DI = OFFFFh, если не хватило памяти, ES:DI = адрес блока (при выделении)

Следует помнить, что область HMA доступна для программ только в том случае, когда адресная линия процессора A20 разблокирована. Если DOS не занимает HMA, она практически постоянно заблокирована на совместимость с программами, написанными для процессора 8086/8088, которые считают, что адреса 0FFFFh:0010h — 0FFFFh:0FFFFh всегда совпадают с 0000h:0000h - 0000h:0FFEFh. Функции XMS 01–07 позволяют управлять состоянием этой адресной линии.

4.9.4. Интерфейс EMS

Расширенная память (EMS) - дополнительная возможность для программ, запускающихся в реальном режиме (или в режиме V86), обращаться к памяти, которая находится за пределами первого мегабайта. EMS позволяет отобразить сегмент памяти, начинающийся обычно с ODOOOh, на любые участки памяти, аналогично тому, как осуществляется доступ к видеопамяти в SVGA-режимах. Вызывать функции EMS (прерывание 67h) разрешается, только если в системе присутствует драйвер с именем EMMXXXXO. Для проверки его существования можно, например, вызвать функцию 3Dh (открыть файл или устройство). Причем на тот случай, если драйвер EMS отсутствует, а в текущей директории есть файл с именем EMMXXXXO, следует дополнительно вызвать функцию IOCTL - INT 21h с AX = 4400h и BX = идентификатор файла или устройства, полученный от функции 3Dh. Если значение бита 7 в DX после вызова этой функции равно 1, то драйвер EMS наверняка присутствует в системе.

Основные функции EMS:

INT 67h, AH = 46h: Получить номер версии
Вход: AH = 46h
Выход: AH = 0 и AL = номер версии в упакованном BCD (40h для 4.0).
Во всех случаях, если AH не ноль, произошла ошибка

INT 67h, AH = 41h: Получить сегментный адрес окна
Вход: AH = 41h
Выход: AH = 0 и BX = сегментный адрес окна

INT 67h, AH = 42h: Получить объем памяти
Вход: AH = 42h
Выход: AH = O
DX = объем EMS-памяти в 16-килобайтных страницах
BX = объем свободной EMS-памяти в 16-килобайтных страницах

INT 67h, AH = 43h: Выделить идентификатор и EMS-память
Вход: AH = 43h
BX = требуемое число 16-килобайтных страниц
Выход: AH = O, DX = идентификатор

Теперь указанный в этой функции набор страниц в EMS-памяти описывается как занятый и другие программы не смогут его выделить для себя.

INT 67h, AH = 44h: Отобразить память
Вход: AH = 44h
AL = номер 16-килобайтной страницы в 64-килобайтном окне EMS (0-3)
BX = номер 16-килобайтной страницы в EMS-памяти
DX = идентификатор
Выход: AH = O

Далее запись/чтение в указанную страницу в реальном адресном пространстве приведет к записи/чтению в указанную страницу в EMS-памяти.

INT 67h, AH = 45h: Освободить идентификатор и EMS-память
Вход: AH = 45h
DX = идентификатор
Выход: AH = 0

Спецификация EMS была разработана для компьютеров IBM XT, снабжавшихся особой платой, на которой и находилась расширенная память. С появлением процессора 80286 стало возможным устанавливать больше одного мегабайта памяти на материнской плате, и для работы с ней была введена новая спецификация - XMS. Тогда же были созданы менеджеры памяти, эмулировавшие EMS поверх XMS, для совместимости со старыми программами, причем работа через EMS выполнялась медленнее. Позже, когда в процессорах Intel появился механизм страничной адресации, выяснилось, что теперь уже EMS можно реализовать гораздо быстрее XMS. Большинство программ для DOS, которым требуется дополнительная память, поддерживают обе спецификации.

4.9.5. *Интерфейс XMS*

Спецификация доступа к дополнительной памяти (XMS) - еще один метод, позволяющий программам, запускающимся под управлением DOS в реальном режиме (или в режиме V86), использовать память, расположенную выше границы первого мегабайта.

INT 2Fh, AH = 43: XMS- и DPMS-сервисы
Вход: AX = 4300h: проверить наличие XMS
Выход: AH = 80h, если HIMEM.SYS или совместимый драйвер загружен
Вход: AX = 4310h: получить точку входа XMS
Выход: ES:BX = дальний адрес точки входа XMS

После определения точки входа все функции XMS вызываются с помощью команды CALL на указанный дальний адрес.

Функция XMS 00h: Определить номер версии
Вход: AH = 00h
Выход: AX = номер версии, не упакованный BCD (0300h для 3.0)
BX = внутренний номер модификации
DX = 1, если HMA существует; 0, если нет

Функция XMS 08h: Определить объем памяти
Вход: AH = 08h
BL = 00h
Выход: AX = размер максимального доступного блока в килобайтах
DX = размер XMS-памяти всего в килобайтах
BL = код ошибки (OAOh, если вся XMS-память занята; 00, если нет ошибок)

Так как возвращаемый размер памяти оказывается ограниченным размером слова (65 535 Кб), начиная с версии XMS 3.0, введена более точная функция 88h.

Функция XMS 88h: Определить объем памяти
Вход: AH = 88h
Выход: EAX = размер максимального доступного блока в килобайтах
BL = код ошибки (OAOh, если вся XMS-память занята; 00, если нет ошибок)
ECX = физический адрес последнего байта памяти (верный для ошибки OAOh)
EDX = размер XMS-памяти в килобайтах (0 для ошибки OAOh)

Функция XMS 09h: Выделить память
Вход: AH = 09h
DX = размер запрашиваемого блока (в килобайтах)
Выход: AX = 1, если функция выполнена
DX = идентификатор блока
AX = 0:
BL = код ошибки (OAOh, если не хватило памяти)

Функция XMS 0Ah: Освободить память
Вход: AH = OAh
DX = идентификатор блока
Выход: AX = 1, если функция выполнена.
Иначе - AX = 0 и BL = код ошибки (OA2h - неправильный идентификатор, OABh - участок заблокирован)

Функция XMS 0Bh: Пересылка данных
Вход: AH = OBh
DS:SI = адрес структуры для пересылки памяти
Выход: AX = 1, если функция выполнена
Иначе - AX = 0 и BL = код ошибки

Структура данных, адрес которой передается в DS:SI:

+00h: 4 байта - число байтов для пересылки
+04h: слово — идентификатор источника (0 для обычной памяти)
+06h: 4 байта - смещение в блоке-источнике или адрес в памяти
+OAh: слово - идентификатор приемника (0 для обычной памяти)
+OCh: 4 байта - смещение в блоке-приемнике или адрес в памяти

Адреса записываются в соответствующие двойные слова в обычном виде - сегмент:смещение. Копирование происходит быстрее, если данные выровнены на границы слова или двойного слова; если области данных перекрываются, адрес начала источника должен быть меньше адреса начала приемника.

Функция XMS 0Fh: Изменить размер XMS-блока
Вход: AH - OFh
BX = новый размер
DX = идентификатор блока
Выход: AX = 1, если функция выполнена.
Иначе - AX = 0 и BL = код ошибки

Кроме того, XMS позволяет программам использовать область HMA и блоки UMB, если они не заняты DOS при запуске (так как в CONFIG.SYS не было строк DOS = HIGH или DOS = UMB).

```
; mem.asm
; Сообщает размер памяти, доступной через EMS и XMS.
;
        .model  tiny
        .code
        .186            ; Для команд сдвига на 4.
        org     100h    ; COM-программа.
start:
        cld             ; Команды строковой обработки будут выполняться вперед.

; Проверка наличия EMS.

        mov     dx,offset ems_driver    ; Адрес ASCIZ-строки "EMMXXXX0".
        mov     ax,3D00h
        int     21h                     ; Открыть файл или устройство.
        jc      no_emmx                 ; Если не удалось открыть - EMS нет.
        mov     bx,ax                   ; Идентификатор в BX.
        mov     ax,4400h
        int     21h                     ; IOCTL: проверить состояние файла/устройства.
        jc      no_ems                  ; Если не произошла ошибка,
```

```
        test    dx,-80h             ; проверить старший бит DX.
        jz      no_ems              ; Если он - 0, EMMXXXX0 - файл в текущей директории.

; Определение версии EMS.

        mov     ah,46h
        int     67h                 ; Получить версию EMS.
        test    ah,ah
        jnz     no_ems              ; Если EMS выдал ошибку - не стоит продолжать
                                    ; с ним работать.
        mov     ah,al
        and     al,0Fh              ; AL = старшая цифра.
        shr     ah,4                ; AH = младшая цифра.
        call    output_version      ; Выдать строку о номере версии EMS.

; Определение доступной EMS-памяти.
        mov     ah,42h
        int     67h                 ; Получить размер памяти в 16-килобайтных страницах.
        shl     dx,4                ; DX = размер памяти в килобайтах.
        shl     bx,4                ; BX = размер свободной памяти в килобайтах,
        mov     ax,bx
        mov     si,offset ems_freemem   ; Адрес строки для output_info.
        call    output_info         ; Выдать строки о размерах памяти.

no_ems:
        mov     ah,3Eh
        int     21h                 ; Закрыть файл/устройство EMMXXXX0.
no_emmx:

; Проверка наличия XMS.

        mov     ax,4300h
        int     2Fh                 ; Проверка XMS.
        cmp     al,80h              ; Если AL не равен 80h,
        jne     no_xms              ; XMS отсутствует.
        mov     ax,4310h            ; Иначе:
        int     2Fh                 ; получить точку входа XMS
        mov     word ptr entry_pt,bx    ; и сохранить ее в entry_pt.
        mov     word ptr entry_pt+2,es  ; (старшее слово - по старшему адресу!)
        push    ds
        pop     es                  ; Восстановить ES.

; Определение версии XMS.
        mov     ah,00
        call    dword ptr entry_pt      ; Функция XMS 00h - номер версии.
        mov     byte ptr mem_version,'X' ; Изменить первую букву строки
                                        ; "EMS версии" на "X".
        call    output_version      ; Выдать строку о номере версии XMS.

; Определение доступной XMS-памяти.
        mov     ah,08h
        xor     bx,bx
        call    dword ptr entry_pt      ; Функция XMS 08h.
```

```
        mov     byte ptr totalmem,'X'   ; Изменить первую букву строки
                                        ; "EMS-памяти" на "X".
        mov     si,offset xms_freemem   ; Строка для output_info.

; Вывод сообщений на экран:
; DX - объем всей памяти,
; AX - объем свободной памяти,
; SI - адрес строки с сообщением о свободной памяти (разный для EMS и XMS).

output_info:
        push    ax
        mov     ax,dx                   ; Объем всей памяти в AX.
        mov     bp,offset totalmem      ; Адрес строки - в BP.
        call    output_info1            ; Вывод.
        pop     ax                      ; Объем Свободной памяти - в AX.
        mov     bp,si                   ; Адрес строки - в BP.

output_info1:                           ; Вывод.
        mov     di,offset hex2dec_word

; hex2dec
; Преобразует целое двоичное число в AX.
; В строку десятичных ASCII-цифр в ES:DI, заканчивающуюся символом "$".
        mov     bx,10                   ; Делитель в BX.
        xor     cx,cx                   ; Счетчик цифр в 0.
divlp:  xor     dx,dx
        div     bx                      ; Разделить преобразуемое число на 10,
        add     dl,'0'                  ; добавить к остатку ASCII-код нуля
        push    dx                      ; записать полученную цифру в стек.
        inc     cx                      ; Увеличить счетчик цифр
        test    ax,ax                   ; и, если еще есть, что делить,
        jnz     divlp                   ; продолжить деление на 10.
store:
        pop     ax                      ; Считать цифру из стека.
        stosb                           ; Дописать ее в конец строки в ES:DI.
        loop    store                   ; Продолжить для всех CX-цифр.
        mov     byte ptr es:[di], '$'   ; Дописать '$' в конец строки.

        mov     dx,bp                   ; DX - адрес первой части строки.
        mov     ah,9
        int     21h                     ; Функция DOS 09h - вывод строки.
        mov     dx,offset hex2dec_word  ; DX - адрес строки с десятичным числом.
        int     21h                     ; Вывод строки.
        mov     dx,offset eol           ; DX - адрес последней части строки.
        int     21h                     ; Вывод строки.

no_xms: ret                             ; Конец программы и процедур output_info
                                        ; и output_info1.

; Вывод версии EMS/XMS.
; AX - номер в неупакованном BCD-формате.
```

```
output_version:
        or      ax,3030h ; Преобразование неупакованного BCD в ASCII.
        mov     byte ptr major,ah               ; Старшая цифра в major.
        mov     byte ptr minor,al               ; Младшая цифра в minor.
        mov     dx,offset mem_version           ; Адрес начала строки - в DX.
        mov     ah,9
        int     21h                             ; Вывод строки.
        ret

ems_driver      db      "EMMXXXX0",0    ; Имя драйвера для проверки EMS.
mem_version     db      "EMS версии "   ; Сообщение о номере версии.
major           db      "0."            ; Первые байты этой
minor           db      "0 обнаружен ",0Dh,0Ah,'$'; и этой строк будут
                                        ; заменены реальными номерами версий.
totalmem        db      "EMS-памяти: $"
ems_freemem     db      "EMS-памяти: $"
eol             db      'K',0Dh,0Ah,'$'         ; Конец строки.
xms_freemem     db      "Наибольший свободный блок XMS: $"

entry_pt:                               ; Сюда записывается точка входа XMS.
hex2dec_word    equ     entry_pt+4      ; Буфер для десятичной строки.
        end     start
```

4.10. Загрузка и выполнение программ

Как и любая операционная система, DOS загружает и выполняет программы. При загрузке программы в начале отводимого для нее блока памяти (для COM-программ это вся свободная на данный момент память) создается структура данных PSP (префикс программного сегмента) размером 256 байт (100h). Затем DOS создает копию текущего окружения для загружаемой программы, помещает полный путь и имя программы в конец окружения, заполняет поля PSP следующим образом:

+00h: слово - OCDh 20h - команда INT 20h. Если COM-программа завершается командой RETN, управление передается на эту команду. Введено для совместимости с командой CP/M CALL O

+02h: слово - сегментный адрес первого байта после области памяти, выделенной для программы

+04h: байт – не используется DOS

+05h: 5 байт - 9Ah OFOh OFEh 01Dh OFOh - команда CALL FAR на абсолютный адрес OOOCOh, записанная так, чтобы второй и третий байты составляли слово, равное размеру первого сегмента для COM-файлов (в этом примере OFEFOh). Введено для совместимости с командой CP/M CALL 5

+0Ah:4 байта – адрес обработчика INT 22h (выход из программы)

+OEh:4 байта – адрес обработчика INT 23h (обработчик нажатия Ctrl-Break)

+12h: 4 байта – адрес обработчика INT 24h (обработчик критических ошибок)

+16h: слово – сегментный адрес PSP процесса, из которого был запущен текущий

+18h: 20 байт – JFT - список открытых идентификаторов, один байт на идентификатор, OFFh - конец списка

+2Ch: слово - сегментный адрес копии окружения для процесса
+2Eh: 2 слова - SS:SP процесса при последнем вызове INT 21h
+32h: слово - число элементов JFT (по умолчанию 20)
+34h: 4 байта - дальний адрес JFT (по умолчанию PSP:0018)
+38h: 4 байта - дальний адрес предыдущего PSP
+3Ch: байт — флаг, указывающий, что консоль находится в состоянии ввода 2-байтного символа
+3Dh: байт - флаг, устанавливаемый функцией 0B71lh прерывания 2Fh (при следующем вызове INT 21h для работы с файлом имя файла будет заменено полным)
+3Eh: слово - не используется в DOS
+40h: слово - версия DOS, которую вернет функция DOS 30h (DOS 5.0+)
+42h: 12 байт - не используется в DOS
+50h: 2 байта - OCDh 21h - команда INT 21h
+52h: байт - 0CBh - команда RETF
+53h: 2 байта - не используется в DOS
+55h: 7 байт - область для расширения первого FOB
+5Ch: 16 байт - первый FCB, заполняемый из первого аргумента командной строки
+6Ch: 16 байт - второй FCB, заполняемый из второго аргумента командной строки
+7Ch: 4 байта - не используется в DOS
+80h: 128 байт - командная строка и область DTA по умолчанию

и записывает программу в память, начиная с адреса PSP:0100h. Если загружается EXE-программа, использующая дальние процедуры или сегменты данных, DOS модифицирует эти команды так, чтобы используемые в них сегментные адреса соответствовали сегментным адресам, которые получили указанные процедуры и сегменты данных при загрузке программы в память. Во время запуска COM-программы регистры устанавливаются следующим образом:

AL = OFFh, если первый параметр командной строки содержит неправильное имя диска (например, z:\something), иначе - 00h
AH = OFFh, если второй параметр содержит неправильное имя диска, иначе - OOh
CS = DS = ES = SS = сегментный адрес PSP
SP = адрес последнего слова в сегменте (обычно 0FFFEh; меньше, если не хватает памяти)

При запуске EXE-программы регистры SS:SP устанавливаются в соответствии с сегментом стека, определенным в программе, затем в стек помещается слово 0000h и выполняется переход на начало программы (PSP:0100h для COM, собственная точка входа для EXE).

Все эти действия выполняет одна функция DOS - загрузить и выполнить программу.

Функция DOS 4Bh: Загрузить и выполнить программу
Вход: AH = 4Bh
AL = OOh - загрузить и выполнить

AL = 01h - загрузить и не выполнять
DS:DX - адрес ASCIZ-строки с полным именем программы
ES:BX - адрес блока параметров EPB:
+00h: слово - сегментный адрес окружения, которое будет скопировано для нового процесса (или 0, если используется текущее окружение)
+02h: 4 байта - адрес командной строки для нового процесса
+06h: 4 байта - адрес первого FCB для нового процесса
+0Ah: 4 байта - адрес второго FCB для нового процесса
+0Eh: 4 байта - здесь будет записан SS:SP нового процесса после его завершения (только для AL = 01)
+12h: 4 байта - здесь будет записан CS:IP (точка входа) нового процесса после его завершения (только для AL = 01)

AL = 03h - загрузить как оверлей
DS:DX - адрес ASCIZ-строки с полным именем программы
ES:BX - адрес блока параметров:
+00h: слово — сегментный адрес для загрузки оверлея
+02h: слово - число, которое будет использовано в командах, применяющих непосредственные сегментные адреса, - обычно то же самое, что и в предыдущем поле. О для COM-файлов

AL = 05h - подготовиться к выполнению (DOS 5.0+)
DS:DX - адрес следующей структуры:
+00h: слово - OOh
+02h: слово - бит 0 - программа - EXE
бит 1 - программа - оверлей
+04h: 4 байта - адрес ASCIZ-строки с именем новой программы
+08h: слово - сегментный адрес PSP новой программы
+OAЪ: 4 байта - точка входа новой программы
+OEh: 4 байта - размер программы, включая PSP

Выход:
CF = 0, если операция выполнена, BX и DX модифицируются,
CF = 1, если произошла ошибка, AX = код ошибки (2 - файл не найден, 5 - доступ к файлу запрещен, 8 - не хватает памяти. OAh - неправильное окружение, OBh - неправильный формат)

Подфункциям 00 и 01 требуется, чтобы свободная память для загрузки программы была в нужном количестве, так что COM-программы должны воспользоваться функцией DOS 4Ah с целью уменьшения отведенного им блока памяти до минимально необходимого. При вызове подфункции 03 DOS загружает оверлей в память, выделенную текущим процессом, поэтому EXE-программы должны убедиться, что ее достаточно.

Эта функция игнорирует расширение файла и различает EXE- и COM-файлы по первым двум байтам заголовка (MZ для EXE-файлов).

Подфункция 05 должна вызываться после загрузки и перед передачей управления на программу, причем никакие прерывания DOS и BIOS нельзя вызывать после возвращения из этой подфункции и до перехода на точку входа новой программы.

Загруженной и вызванной таким образом программе предоставляется несколько способов завершения работы. Способ, который чаще всего применяется для COM-файлов, - команда RETN. При этом управление передается на адрес PSP:0000, где располагается код команды INT 20h. Соответственно программу можно завершить сразу, вызвав INT 20h, но оба эти способа требуют, чтобы CS содержал сегментный адрес PSP текущего процесса. Кроме того, они не позволяют вернуть код возврата, который может передать предыдущему процессу информацию о том, как завершилась запущенная программа. Рекомендованный способ завершения программы - функция DOS 4Ch.

Функция DOS 4Ch: Завершить программу
Вход: AH = 4Ch
AL = код возврата

Значение кода возврата можно использовать в пакетных файлах DOS как переменную ERRORLEVEL и определять из программы с помощью функции DOS 4Dh.

Функция DOS 4Dh: Определить код возврата последнего завершившегося процесса
Вход: AH - 4Dh
Выход: AH = способ завершения:
00h - нормальный
01h - **Ctrl-Break**
02h - критическая ошибка
03h - программа осталась в памяти как резидентная
AL = код возврата
CF = 0

Воспользуемся функциями 4Ah и 4Bh в следующем примере программы, которая ведет себя как командный интерпретатор, хотя на самом деле единственная команда, которую она обрабатывает, - команда exit. Все остальные команды передаются настоящему COMMAND.COM с ключом /C (выполнить команду и вернуться).

```
; shell.asm
; Программа, выполняющая функции командного интерпретатора
; (вызывающая command.com для всех команд, кроме exit).
;
        .model  tiny
        .code
        .186
        org     100h                            ; COM-программа

prompt_end      equ     "$"             ; Последний символ в приглашении ко вводу.

start:
        mov     sp,length_of_program+100h+200h  ; Перемещение стека на 200h
                                                ; после конца программы
                                                ; (дополнительные 100h - для PSP).
```

```
        mov     ah,4Ah
stack_shift=length_of_program+100h+200h
        mov     bx,stack_shift shr 4+1
        int     21h                     ; Освободить всю память после конца
                                        ; программы и стека.

; Заполнить поля EPB, содержащие сегментные адреса.
        mov     ax,cs
        mov     word ptr EPB+4,ax       ; Сегментный адрес командной строки.
        mov     word ptr EPB+8,ax       ; Сегментный адрес первого FCB.
        mov     word ptr EPB+0Ch,ax     ; Сегментный адрес второго FCB.

main_loop:

; Построение и вывод приглашения для ввода.

        mov     ah,19h                  ; Функция DOS 19h:
        int     21h                     ; определить текущий диск.
        add     al,'A'                  ; Теперь AL = ASCII-код диска (A, B, C,).
        mov     byte ptr drive_letter,al ; Поместить его в строку.
        mov     ah,47h                  ; Функция DOS 47h:
        mov     dl,00
        mov     si,offset pwd_buffer
        int     21h                     ; определить текущую директорию
        mov     al,0                    ; Найти ноль (конец текущей директории)
        mov     di,offset prompt_start  ; в строке с приглашением.
        mov     cx,prompt_l
        repne   scasb
        dec     di                      ; DI - адрес байта с нулем.

        mov     dx,offset prompt_start  ; DS:DX - строка приглашения.
        sub     di,dx                   ; DI - длина строки приглашения.
        mov     cx,di
        mov     bx,1                    ; stdout
        mov     ah,40h
        int     21h                     ; Вывод строки в файл или устройство.
        mov     al,prompt_end
        int     29h                     ; Вывод последнего символа в приглашении.

; получить команду от пользователя
        mov     ah,0Ah                  ; Функция DOS 0Ah:
        mov     dx,offset command_buffer
        int     21h                     ; буферированный ввод.

        mov     al,0Dh                  ;  Вывод символа CR
        int     29h
        mov     al,0Ah                  ;  Вывод символа LF
        int     29h                     ;  (CR и LF вместе - перевод строки).

        cmp     byte ptr command_buffer+1,0     ; Если введена пустая строка,
        je      main_loop                       ; продолжить основной цикл.
```

```
; Проверить, является ли введенная команда командой "exit".

        mov     di,offset command_buffer+2      ; Адрес введенной строки.
        mov     si,offset cmd_exit              ; Адрес эталонной строки
                                                ; "exit",0Dh.
        mov     cx,cmd_exit_l                   ; Длина эталонной строки.
        repe    cmpsb                           ; Сравнить строки.
        jcxz    got_exit                        ; Если строки идентичны -
                                                ; выполнить exit.

; Передать остальные команды интерпретатору DOS (COMMAND.COM).
        xor     cx,cx
        mov     si,offset command_buffer+2      ; Адрес введенной строки.
        mov     di,offset command_text          ; Параметры для command.com.
        mov     cl,byte ptr command_buffer+1    ; Размер введенной строки.
        inc     cl                              ; Учесть 0Dh в конце.
        rep     movsb                           ; Скопировать строку.
        mov     ax,4B00h                        ; Функция DOS 4Bh.
        mov     dx,offset command_com           ; Адрес ASCIZ-строки с адресом.
        mov     bx,offset EPB
        int     21h                             ; Исполнить программу.

        jmp     short main_loop                 ; Продолжить основной цикл.
got_exit:
        int     20h                             ; Выход из программы (ret нельзя,
                                                ; потому что мы перемещали стек).

cmd_exit        db      "exit",0Dh              ; Команда "exit".
cmd_exit_l      equ     $-cmd_exit              ; Ее длина.

prompt_start    db      "tinyshell:"            ; Подсказка для ввода.
drive_letter    db      "C:"
pwd_buffer      db      64 dup (?)              ; Буфер для текущей директории.
prompt_l        equ     $-prompt_start          ; Максимальная длина подсказки.

command_com     db      "C:\COMMAND.COM",0      ; Имя файла.

EPB             dw      0000                    ; Использовать текущее окружение.
                dw      offset commandline,0    ; Адрес командной строки.
                dw      005Ch,0,006Ch,0         ; Адреса FCB, переданных DOS
                                                ; нашей программе при запуске
                                                ; (на самом деле они не используются).

commandline     db      125                     ; Максимальная длина
                                                ; командной строки.
                db      " /C "                  ; Ключ /C для COMMAND.COM.
command_text    db      122 dup (?)             ; Буфер для командной строки.

command_buffer  db 122                          ; Здесь начинается буфер для ввода.
length_of_program equ   124+$-start             ; Длина программы + длина
                                                ; буфера для ввода.

        end     start
```

Чтобы сократить пример, в нем используются функции для работы с обычными короткими именами файлов. Достаточно заменить строку

```
        mov     ah,47h
```

на

```
        mov     ax,7147h
```

и увеличить размер буфера для текущей директории (pwd_buffer) с 64 до 260 байт, и директории с длинными именами будут отображаться корректно в подсказке для ввода. Но с целью совместимости следует также добавить проверку на поддержку функции 71h (LFN) и определить размер буфера для директории с помощью подфункции LFN 0A0h.

4.11. Командные параметры и переменные среды

В случае если команда не передавалась бы интерпретатору DOS, а выполнялась нами самостоятельно, то оказалось бы: чтобы запустить любую программу из-под shell.com, нужно предварительно перейти в директорию с этой программой или ввести ее, указав полный путь. Дело в том, что COMMAND.COM при запуске файла ищет его по очереди в каждой из директорий, указанных в переменной среды PATH. DOS создает копию всех переменных среды (так называемое окружение DOS) для каждого запускаемого процесса. Сегментный адрес копии окружения для текущего процесса располагается в PSP по смещению 2Ch. В этом сегменте записаны все переменные подряд в форме ASCIZ-строк вида "COMSPEC=C:\WINDOWS\COMMAND.COM",0. По окончании последней строки стоит дополнительный нулевой байт, затем слово (обычно 1) - количество дополнительных строк окружения, а потом - дополнительные строки. Первая дополнительная строка - всегда полный путь и имя текущей программы - также в форме ASCIZ-строки. При запуске новой программы с помощью функции 4Bh можно создать полностью новое окружение и передать его сегментный адрес запускаемой программе в блоке EPB или просто указать 0, позволив DOS скопировать окружение текущей программы.

Кроме того, в предыдущем примере мы передавали запускаемой программе (command.com) параметры (/c команда), но пока не объяснили, как программа может определить, что за параметры были переданы ей при старте. Во время запуска программы DOS помещает всю командную строку (включая последний символ 0Dh) в блок PSP запущенной программы по смещению 81h и ее длину в байт 80h (таким образом, длина командной строки не может быть больше 7Eh (126) символов). Под Windows 95 и DOS 4.0, если командная строка превышает эти размеры, байт PSP:0080h (длина) устанавливается в 7Fh, в последний байт PSP (PSP:OOFFh) записывается ODh, первые 126 байт командной строки размещаются в PSP, а вся строка целиком - в переменной среды CMDLINE.

```
; cat.asm
; Копирует объединенное содержимое всех файлов, указанных в командной строке,
; в стандартный вывод. Можно как указывать список файлов, так и использовать
; маски (символы "*" и "?") в одном или нескольких параметрах,
```

```
; например:
; cat header *.txt footer > all-texts помещает содержимое файла
; header, всех файлов с расширением .txt в текущей директории и файла
; footer - в файл all-texts.
; Длинные имена файлов не используются, ошибки игнорируются.
;
        .model  tiny
        .code
        org     80h             ; По смещению 80h от начала PSP находятся:
cmd_length      db ?            ; длина командной строки
cmd_line        db ?            ; и сама командная строка.
        org     100h            ; Начало COM-программы - 100h от начала PSP.
start:
        cld                     ; Для команд строковой обработки.
        mov     bp,sp           ; Сохранить текущую вершину стека в BP.
        mov     cl,cmd_length
        cmp     cl,1            ; Если командная строка пуста -
        jle     show_usage      ; вывести информацию о программе и выйти.

        mov     ah,1Ah          ; функция DOS 1Ah:
        mov     dx,offset DTA
        int     21h             ; переместить DTA (по умолчанию она совпадает
                                ; с командной строкой PSP)

; Преобразовать список параметров в PSP:81h следующим образом:
; Каждый параметр заканчивается нулем (ASCIZ-строка),
; адреса всех параметров помещаются в стек в порядке обнаружения.
; В переменную argc записывается число параметров.

        mov     cx,-1           ; Для команд работы со строками.
        mov     di,offset cmd_line ; Начало командной строки в ES:DI.

find_param:
        mov     al,' '          ; Искать первый символ,
        repz    scasb           ; не являющийся пробелом.
        dec     di              ; DI - адрес начала очередного параметра.
        push    di              ; Поместить его в стек
        inc     word ptr argc   ; и увеличить argc на один.
        mov     si,di           ; SI = DI для следующей команды lodsb.
scan_params:
        lodsb                   ; Прочитать следующий символ из параметра.
        cmp     al,0Dh          ; Если это 0Dh - это был последний параметр
        je      params_ended    ; и он кончился.
        cmp     al,20h          ; Если это 20h - этот параметр кончился,
        jne     scan_params     ; но могут быть еще.

        dec     si              ; SI - первый байт после конца параметра.
        mov     byte ptr [si],0 ; Записать в него 0.
        mov     di,si           ; DI = SI для команды scasb.
        inc     di              ; DI - следующий после нуля символ.
        jmp     short find_param ; Продолжить разбор командной строки.
```

```
params_ended:
        dec     si                      ; SI - первый байт после конца последнего
        mov     byte ptr [si],0         ; параметра - записать в него 0.

; Каждый параметр воспринимается как файл или маска для поиска файлов,
; все найденные файлы выводятся на stdout. Если параметр - не имя файла,
; то ошибка игнорируется.

        mov     si,word ptr argc        ; SI - число оставшихся параметров.
next_file_from_param:
        dec     bp
        dec     bp                      ; BP - адрес следующего адреса параметра.

        dec     si                      ; Уменьшить число оставшихся параметров,
        js      no_more_params          ; если оно стало отрицательным - все.

        mov     dx,word ptr [bp]        ; DS:DX - адрес очередного параметра.

        mov     ah,4Eh                  ; Функция DOS 4Eh.
        mov     cx,0100111b             ; Искать все файлы, кроме директорий
                                        ; и меток тома.
        int     21h                     ; Найти первый файл.
        jc      next_file_from_param    ; Если произошла ошибка - файла нет.

        call    output_found            ; Вывести найденный файл на stdout.

find_next:
        mov     ah,4Fh                  ; Функция DOS 4Fh.
        mov     dx,offset DTA           ; Адрес нашей области DTA.
        int     21h                     ; Найти следующий файл.
        jc      next_file_from_param    ; Если ошибка - файлы кончились.

        call    output_found            ; Вывести найденный файл на stdout.

        jmp     short find_next         ; Продолжить поиск файлов.

no_more_params:
        mov     ax,word ptr argc
        shl     ax,1
        add     sp,ax                   ; Удалить из стека 2 x argc байтов (то есть весь
                                        ; список адресов параметров командной строки).
        ret                             ; Конец программы.

; Процедура show_usage
; Выводит информацию о программе.
;
show_usage:
        mov     ah,9                    ; Функция DOS 09h.
        mov     dx,offset usage
        int     21h                     ; Вывести строку на экран.
        ret                             ; Выход из процедуры.

; Процедура output_found.
; Выводит в stdout файл, имя которого находится в области DTA.
```

```
output_found:
        mov     dx,offset DTA+1Eh; Адрес ASCIZ-строки с именем файла.
        mov     ax,3D00h        ; Функция DOS 3Dh:
        int     21h             ; открыть файл (al = 0 - только на чтение).
        jc      skip_file       ; Если ошибка - не трогать этот файл.
        mov     bx,ax           ; Идентификатор файла - в BX.
        mov     di,1            ; DI будет хранить идентификатор STDOUT.
do_output:
        mov     cx,1024         ; Размер блока для чтения файла.
        mov     dx,offset DTA+45 ; Буфер для чтения/записи располагается за концом DTA.
        mov     ah,3Fh          ; Функция DOS 3Fh.
        int     21h             ; Прочитать 1024 байта из файла.
        jc      file_done       ; Если ошибка - закрыть файл.
        mov     cx,ax           ; Число реально прочитанных байтов в CX.
        jcxz    file_done       ; Если это не ноль - закрыть файл.

        mov     ah,40h          ; Функция DOS 40h.
        xchg    bx,di           ; BX = 1 - устройство STDOUT.
        int     21h             ; Вывод прочитанного числа байтов в STDOUT.
        xchg    di,bx           ; Вернуть идентификатор файла в BX.
        jc      file_done       ; Если ошибка - закрыть файл,
        jmp     short do_output ; продолжить вывод файла.
file_done:
        mov     ah,3Eh          ; Функция DOS 3Eh:
        int     21h             ; закрыть файл.
skip_file:
        ret                     ; Конец процедуры output_found.

usage           db      "cat.com v1.0",0Dh,0Ah
                db      "объединяет и выводит файлы на stdout",0Dh,0Ah
                db      "использование: cat имя_файла, ...",0Dh,0Ah
                db      "(имя файла может содержать маски * и ?)",0Dh,0Ah,'$'

argc            dw      0       ; Число параметров
                                ; (должен быть 0 при старте программы!).

DTA:                            ; Область DTA начинается сразу за концом файла,
                                ; а за областью DTA -
                                ; 1024-байтный буфер для чтения файла.
        end     start
```

Размер блока для чтения файла можно значительно увеличить, но в таком случае почти наверняка потребуется проследить за объемом памяти, доступным для программы.

Глава 5. Более сложные приемы программирования

Все примеры программ из предыдущей главы в первую очередь предназначались для демонстрации работы с теми или иными основными устройствами компьютера при помощи средств, предоставляемых DOS и BIOS. В этой главе рассказано о том, что и в области собственно программирования ассемблер позволяет больше, чем любой другой язык, и рассмотрены те задачи, решая которые, принято использовать язык ассемблера при программировании для DOS.

5.1. Управляющие структуры

5.1.1. Структуры IF... THEN... ELSE

Эти часто встречающиеся управляющие структуры передают управление на один участок программы, если некоторое условие выполняется, и на другой, если оно не выполняется, записываются на ассемблере в следующем общем виде:

```
(набор команд, проверяющих условие)
        Jcc     Else
(набор команд, соответствующих блоку THEN)
        jmp     Endif
Else:   (набор команд, соответствующих блоку ELSE)
Endif:
```

Для сложных условий часто оказывается, что одной командой условного перехода обойтись нельзя, поэтому реализация проверки может сильно увеличиться. Например, следующую строку на языке С

```
if (((x>y) && (z<t)) || (a!=b)) c=d;
```

можно представить на ассемблере так:

```
; Проверка условия.
        mov     ax,A
        cmp     ax,B
        jne     then    ; Если a! = b - условие выполнено.
        mov     ax,X
        cmp     ax,Y
        jng     endif   ; Если x < или = y - условие не выполнено.
        mov     ax,Z
        cmp     ax,T
        jnl     endif   ; Если z > или = t - условие не выполнено.
```

```
then:                                   ; Условие выполняется.
        mov     ax,D
        mov     C,ax
endif:
```

5.1.2. Структуры CASE

Управляющая структура типа CASE проверяет значение некоторой переменной (или выражения) и передает управление на различные участки программы. Кажется очевидным, что эта структура должна реализовываться в виде серии структур IF... THEN... ELSE, как показано в примерах, где требовались различные действия в зависимости от значения нажатой клавиши.

Пусть переменная I принимает значения от 0 до 2, и в зависимости от значения надо выполнить процедуры case0, easel и case2:

```
        mov     ax,I
        cmp     ax,0            ; Проверка на 0.
        jne     not0
        call    case0
        jmp     endcase
not0:   cmp     ax,1            ; Проверка на 1.
        jne     not1
        call    easel
        jmp     endcase
not1:   cmp     ax,2            ; Проверка на 2.
        jne     not2
        call    case2
not2:
endcase:
```

Но ассемблер предоставляет более удобный способ реализации таких структур - таблицу переходов:

```
        mov     bx,I
        shl     bx,1            ; Умножить BX на 2 (размер адреса
                                ; в таблице переходов - 4 для
                                ; 32-битных адресов).
        jmp     cs:jump_table[bx]; Разумеется, в этом примере
                                ; достаточно использовать call.

jump_table      dw      foo0,foo1,foo2  ; Таблица переходов.

foo0:   call    case0
        jmp     endcase
foo1:   call    easel
        jmp     endcase
foo2:   call    case2
        jmp     endcase
```

Очевидно, что для переменной с большим числом значений способ с таблицей переходов является наиболее быстрым (не требуется многочисленных проверок), а если значения переменной - числа, следующие в точности друг за другом (так

что в таблице переходов нет пустых участков), то эта реализация структуры CASE окажется еще и значительно меньше.

5.1.3. Конечные автоматы

Конечный автомат - процедура, которая помнит свое состояние и при обращении к ней выполняет различные действия для разных состояний. Например, рассмотрим процедуру, которая складывает регистры AX и BX при первом вызове, вычитает при втором, умножает при третьем, делит при четвертом, снова складывает при пятом и т. д. Очевидная реализация, опять же, состоит в последовательности условных переходов:

```
state           db      0
state_machine:
        cmp     state,0
        jne     not_0
; Состояние 0: сложение.
        add     ax,bx
        inc     state
        ret
not_0:  cmp     state,1
        jne     not_1
; Состояние 1: вычитание.
        sub     ax,bx
        inc     state
        ret
not_1:  cmp     state,2
        jne     not_2
; Состояние 2: умножение.
        push    dx
        mul     bx
        pop     dx
        inc     state
        ret
; Состояние 3: деление.
not_2:  push    dx
        xor     dx,dx
        div     bx
        pop     dx
        mov     state,0
        ret
```

Оказывается, что (как и для CASE) в ассемблере есть средства для более эффективной реализации данной структуры - все тот же косвенный переход:

```
state           dw      offset state_0
state_machine:
        jmp     state
state_0: add    ax,bx           ; Состояние 0: сложение.
        mov     state,offset state_1
        ret
```

```
state_1: sub     ax,bx           ; Состояние 1: вычитание.
         mov     state,offset state_2
         ret
state_2: push    dx              ; Состояние 2: умножение.
         mul     bx
         pop     dx
         mov     state,offset state_3
         ret
state_3: push    dx              ; Состояние 3: деление.
         xor     dx,dx
         div     bx
         pop     dx
         mov     state,offset state_0
         ret
```

Как и в случае с CASE, использование косвенного перехода приводит к тому, что не требуется никаких проверок и время выполнения управляющей структуры остается одним и тем же для четырех или четырех тысяч состояний.

5.1.4. Циклы

Несмотря на то что набор команд Intel включает команды организации циклов, они годятся только для одного типа циклов - FOR-циклов, которые выполняются фиксированное число раз. В общем виде любой цикл записывается в ассемблере как условный переход.

```
WHILE-цикл:
         (команды инициализации цикла)
метка:   IF (не выполняется условие окончания цикла)
         THEN
         (команды тела цикла)
         jmp метка

REPEAT/UNTIL-цикл:
         (команды инициализации цикла)
метка:   (команды тела цикла)
         IF (не выполняется условие окончания цикла)
         THEN (переход на метку)

(такие циклы выполняются быстрее на ассемблере, и всегда следует стремиться
переносить проверку условия окончания цикла в конец)

LOOP/ENDLOOP-цикл:
         (команды инициализации цикла)
метка:
         (команды тела цикла)
         IF (выполняется условие окончания цикла)
         THEN jmp метка2
         (команды тела цикла)
         jmp метка
метка2:
```

5.2. Процедуры и функции

Можно разделять языки программирования на процедурные (C, Pascal, Fortran, BASIC) и непроцедурные (LISP, FORTH, PROLOG), где процедуры - блоки кода программ, имеющие одну точку входа и одну точку выхода и возвращающие управление на следующую команду после команды передачи управления процедуре. Ассемблер одинаково легко можно использовать как процедурный язык и как непроцедурный, и в большинстве примеров программ до сих пор мы успешно нарушали рамки и того, и другого подхода. В настоящей главе реализация процедурного подхода рассмотрена в качестве наиболее популярной.

5.2.1. Передача параметров

Процедуры могут получать или не получать параметры из вызывающей процедуры и могут возвращать или не возвращать результаты (процедуры, которые что-либо возвращают, называются функциями в языке Pascal, но ассемблер не делает каких-либо различий между ними).

Параметры можно передавать с помощью одного из шести механизмов:

- по значению;
- по ссылке;
- по возвращаемому значению;
- по результату;
- по имени;
- отложенным вычислением.

Параметры можно передавать в одном из пяти мест:

- в регистрах;
- в глобальных переменных;
- в стеке;
- в потоке кода;
- в блоке параметров.

Следовательно, всего в ассемблере возможно 30 различных способов передачи параметров для процедур. Рассмотрим их по порядку.

Передача параметров по значению

Процедуре передается собственно значение параметра. При этом фактически значение параметра копируется, и процедура использует его копию, так что модификация исходного параметра оказывается невозможной. Этот механизм применяется для передачи небольших параметров, таких как байты или слова, к примеру, если параметры передаются в регистрах:

```
mov     ax,word ptr value       ; Сделать копию значения.
call    procedure               ; Вызвать процедуру.
```

Передача параметров по ссылке

Процедуре передается не значение переменной, а ее адрес, по которому процедура сама прочитает значение параметра. Этот механизм удобен для передачи

больших массивов данных и в тех случаях, когда процедура должна модифицировать параметры (хотя он и медленнее из-за того, что процедура будет выполнять дополнительные действия для получения значений параметров).

```
        mov     ax,offset value
        call    procedure
```

Передача параметров по возвращаемому значению

Этот механизм объединяет передачу по значению и по ссылке. Процедуре передают адрес переменной, а процедура делает локальную копию параметра, затем работает с ней, а в конце записывает локальную копию обратно по переданному адресу. Этот метод эффективнее обычной передачи параметров по ссылке в тех случаях, когда процедура должна обращаться к параметру очень большое количество раз, например, если используется передача параметров в глобальной переменной:

```
        mov     . global_variable,offset value
        call    procedure
        [...]
procedure       proc    near
        mov     dx,global_variable
        mov     ax, word ptr [dx]
        (команды, работающие с AX в цикле десятки тысяч раз)
        mov     word ptr [dx],ax
procedure       endp
```

Передача параметров по результату

Этот механизм отличается от предыдущего только тем, что при вызове процедуры предыдущее значение параметра никак не определяется, а переданный адрес используется только для записи в него результата.

Передача параметров по имени

Данный механизм используют макроопределения, директива EQU, а также, например, препроцессор C во время обработки команды #define. При реализации этого механизма в компилирующем языке программирования (к которому относится и ассемблер) приходится заменять передачу параметра по имени другими механизмами с помощью, в частности, макроопределений.

Если установлено макроопределение

```
pass_by_name    macro   parameter1
        mov     ax,parameter1
endm
```

то теперь параметр в программе можно передавать следующим образом:

```
        pass_by_name value
        call    procedure
```

Примерно так же поступают языки программирования высокого уровня, поддерживающие этот механизм: процедура получает адрес специальной функции-заглушки, которая вычисляет адрес передаваемого по имени параметра.

Передача параметров отложенным вычислением

Как и в предыдущем случае, здесь процедура получает адрес функции, вычисляющей значение параметра. Такой механизм удобен, если вычисление значения параметра требует много ресурсов или времени, например, если функция должна выбрать один из нескольких ходов при игре в шахматы, вычисление каждого параметра может занимать несколько минут. Во время передачи параметров отложенным вычислением функция получает адрес заглушки, которая при первом обращении к ней вычисляет значение параметра и сохраняет его во внутренней локальной переменной, а при дальнейших вызовах возвращает ранее вычисленное значение. Если процедуре вообще не потребуются значения части параметров (например, если первый же ход приводит к мату), то использование отложенных вычислений способствует выигрышу с большей скоростью. Этот механизм чаще всего применяется в системах искусственного интеллекта и операционных системах.

Рассказав об основных механизмах передачи параметров процедуре, рассмотрим теперь варианты, *где* их передавать.

Передача параметров в регистрах

Если процедура получает небольшое число параметров, идеальным местом для их передачи оказываются регистры. Примерами служат практически все вызовы прерываний DOS и BIOS. Языки высокого уровня обычно используют регистр AX (EAX) для того, чтобы возвращать результат работы функции.

Передача параметров в глобальных переменных

Когда не хватает регистров, один из способов обойти это ограничение - записать параметр в переменную, к которой затем следует обращаться из процедуры. Этот метод считается неэффективным, и его использование может привести к тому, что рекурсия и повторная входимость станут невозможными.

Передача параметров в стеке

Параметры помещаются в стек сразу перед вызовом процедуры. Именно этот метод используют языки высокого уровня, такие как C и Pascal. Для чтения параметров из стека обычно применяют не команду POP, а регистр BP, в который помещают адрес вершины стека после входа в процедуру:

```
        push    parameter1      ; Поместить параметр в стек.
        push    parameter2
        call    procedure
        add     sp,4            ; Освободить стек от параметров.
        [...]
procedure       proc    near
        push    bp
        mov     bp,sp
        (команды, которые могут использовать стек)
        mov     ax,[bp+4]       ; Считать параметр 2.
; Его адрес в сегменте стека BP + 4, потому что при выполнении команды CALL
; в стек поместили адрес возврата - 2 байта для процедуры
; типа NEAR (или 4 - для FAR), а потом еще и BP - 2 байта.
```

```
        mov     bx,[bp+6]       ; Считать параметр 1.
        (остальные команды)
        pop     bp
        ret
procedure       endp
```

Параметры в стеке, адрес возврата и старое значение BP вместе называются *активизационной записью* функции.

Для удобства ссылок на параметры, переданные в стеке, внутри функции иногда используют директивы EQU, чтобы не писать каждый раз точное смещение параметра от начала активизационной записи (то есть от BP), например так:

```
        push    X
        push    Y
        push    Z
        call    xyzzy
        [...]
xyzzy   proc    near
xyzzy_z equ     [bp+8]
xyzzy_y equ     tbp+6]
xyzzy_x equ     [bp+4]
        push    bp
        mov     bp,sp
        (команды, которые могут использовать стек)
        mov     ax,xyzzy_x      ;Считать параметр X.
        (остальные команды)
        pop     bp
        ret     6
xyzzy   endp
```

При внимательном анализе этого метода передачи параметров возникает сразу два вопроса: кто должен удалять параметры из стека, процедура или вызывающая ее программа, и в каком порядке помещать параметры в стек. В обоих случаях оказывается, что оба варианта имеют свои «за» и «против». Так, например, если стек освобождает процедура (командой RET число_байтов), то код программы получается меньшим, а если за освобождение стека от параметров отвечает вызывающая функция, как в нашем примере, то становится возможным последовательными командами CALL вызвать несколько функций с одними и теми же параметрами. Первый способ, более строгий, используется при реализации процедур в языке Pascal, а второй, дающий больше возможностей для оптимизации, - в языке C. Разумеется, если передача параметров через стек применяется и для возврата результатов работы процедуры, из стека не надо удалять все параметры, но популярные языки высокого уровня не пользуются этим методом. Кроме того, в языке C параметры помещают в стек в обратном порядке (справа налево), так что становятся возможными функции с изменяемым числом параметров (как, например, printf – первый параметр, считываемый из [BP+4], определяет число остальных параметров). Но подробнее о тонкостях передачи параметров в стеке рассказано далее, а здесь приведен обзор методов.

Передача параметров в потоке кода

В этом необычном методе передаваемые процедуре данные размещаются прямо в коде программы, сразу после команды CALL (как реализована процедура print в одной из стандартных библиотек процедур для ассемблера UCRLIB):

```
        call    print
                db   '   "This ASCIZ-line will be printed",0
        (следующая   команда)
```

Чтобы прочитать параметр, процедура должна использовать его адрес, который автоматически передается в стеке как адрес возврата из процедуры. Разумеется, функция должна будет изменить адрес возврата на первый байт после конца переданных параметров перед выполнением команды RET. Например, процедуру print можно реализовать следующим образом:

```
print           proc    near
        push    bp
        mov     bp,sp
        push    ax
        push    si
        mov     si,[bp+2]          Прочитать адрес
                                ; возврата/начала данных.
        cld                     ; Установить флаг направления
                                ; для команды lodsb.
print_readchar:
        lodsb                   ; Прочитать байт из строки.
        test    al,al             Если это 0 (конец строки),
        jz      print_done        вывод строки закончен.
        int     29h               Вывести символ в AL на экран.
        jmp     short print_readchar
print_done:
        mov     [bp+2],si       ; Поместить новый адрес возврата в стек.
        pop     si
        pop     ax
        pop     bp
        ret
print           endp
```

Передача параметров в потоке кода, так. же как и передача параметров в стеке в обратном порядке (справа налево), позволяет передавать различное число параметров, но этот'метод - единственный, дающий возможность передать по значению параметр различной длины, что и продемонстрировал приведенный пример. Доступ к параметрам, переданным в потоке кода, осуществляется несколько медленнее, чем к параметрам, переданным в регистрах, глобальных переменных или стеке, и примерно совпадает со следующим методом.

Передача параметров в блоке параметров

Блок параметров - это участок памяти, содержащий параметры, так же как и в предыдущем примере, но располагающийся обычно в сегменте данных.

Процедура получает адрес начала этого блока при помощи любого метода передачи параметров (в регистре, в переменной, в стеке, в коде или даже в другом блоке параметров). В качестве примеров реализации этого метода можно назвать многие функции DOS и BIOS - поиск файла, использующий блок параметров DTA, или загрузка (и исполнение) программы, использующая блок параметров EPB.

5.2.2. Локальные переменные

Часто процедурам требуются локальные переменные, которые не будут нужны после того, как процедура закончится. По аналогии с методами передачи параметров можно говорить о локальных переменных в регистрах - каждый регистр, который сохраняют при входе в процедуру и восстанавливают при выходе, фактически играет роль локальной переменной. Единственный недостаток регистров в роли локальных переменных - их слишком мало. Следующий вариант - хранение локальных данных в переменной в сегменте данных - удобен и быстр для большинства несложных ассемблерных программ, но процедуру, использующую этот метод, нельзя вызывать рекурсивно: такая переменная на самом деле является глобальной и находится в одном и том же месте в памяти для каждого вызова процедуры. Третий и наиболее распространенный способ хранения локальных переменных в процедуре - стек. Принято располагать локальные переменные в стеке сразу после сохраненного значения регистра BP, так что на них можно ссылаться изнутри процедуры, как [BP-2], [BP-4], [BP-6] и т. д.:

```
foobar          proc    near
foobar_x        equ     [bp+8]   ; Параметры.
foobar_y        equ     [bp+6]
foobar_z        equ     [bp+4]
foobar_l        equ     [bp-2]   ; Локальные переменные.
foobar_m        equ     [bp-4]
foobar_n        equ     [bp-6]

        push    bp      ; Сохранить предыдущий BP.
        mov     bp,sp     Установить BP для этой процедуры.
        sub     sp,6      Зарезервировать 6 байт для
                        ; локальных переменных.
        (тело процедуры)
        mov     sp,bp   ; Восстановить SP, выбросив
                        ; из стека все локальные переменные.
        pop     bp      ; Восстановить BP вызвавшей процедуры.
        ret     6       ; Вернуться, удалив параметры из стека.
foobar          endp
```

Внутри процедуры foobar стек будет заполнен так, как показано на рис. 16.

Последовательности команд, используемые в начале и в конце названных процедур, оказались настолько часто применяемыми, что в процессоре 80186 были введены специальные команды ENTER и LEAVE, выполняющие эти же самые действия:

```
foobar          proc    near
foobar_x        equ     [bp+8]   ; Параметры.
foobar_y        equ     [bp+6]
```

```
foobar_z        equ     [bp+4]
foobar_l        equ     [bp-2]   ; Локальные переменные.
foobar_m        equ     [bp-4]
foobar_n        equ     [bp-6]

        enter 6,0                ; push bp
                                 ; mov bp,sp
                                 ; sub sp,6
(тело процедуры)
        leave                    ; mov sp,bp
                                 ; pop bp
        ret     6                ; Вернуться, удалив
                                 ; параметры из стека.
foobar          endp
```

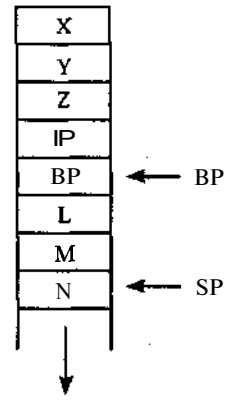

Рис. 16. Стек при вызове процедуры foobar

Область в стеке, отводимая для локальных переменных вместе с активизационной записью, называется *стековым кадром*.

5.3. Вложенные процедуры

Во многих языках программирования можно описывать процедуры внутри друг друга, так что локальные переменные, объявленные в пределах одной процедуры, доступны только из этой процедуры и всех вложенных в нее. Разные языки программирования используют различные способы реализации доступа к переменным, объявленным в функциях с меньшим уровнем вложенности (уровень вложенности главной процедуры определяют как 0 и увеличивают на 1 с каждым новым вложением).

5.3.1. Вложенные процедуры со статическими ссылками

Самый простой способ предоставить вложенной процедуре доступ к локальным переменным, объявленным во внешней процедуре, - просто передать ей вместе с параметрами адрес активизационной записи, содержащей эти переменные (см. рис. 17).

При этом, если процедура вызывает вложенную в себя процедуру, она просто передает ей свой BP, например так:

```
push    bp
call    nested_proc
```

То есть статическая и динамическая ссылки в активизационной записи процедуры nested_proc в этом случае не различаются. Если процедура вызывает другую процедуру на том же уровне вложенности, она должна передать ей адрес активизационной записи из общего предка:

```
push    [bp+4]
call    peer_proc
```

Если же процедура вызывает процедуру значительно меньшего уровня вложенности, так же как если процедура хочет получить доступ к переменным,

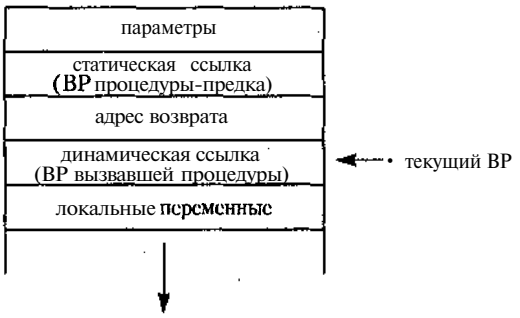

Рис. 17. Стек процедуры со статическими ссылками

объявленным в процедуре меньшего уровня вложенности, она должна проследовать по цепочке статических ссылок наверх, вплоть до требуемого уровня. То есть, если процедура на уровне вложенности 5 должна вызвать процедуру на уровне вложенности 2, она должна поместить в стек адрес активизационной записи внешней по отношению к ним обоим процедуры с уровня вложенности 1:

```
mov     bx,[bp+4]       ; Адрес записи уровня 4 в BX.
mov     bx,ss:[bx+4]    ; Адрес записи уровня 3 в BX.
mov     bx,ss:[bp+4]    ; Адрес записи уровня 2 в BX.
push    ss:[bx+4]       ; Адрес записи уровня 1 в стек.
call    proc_at_level2
```

Метод реализации вложенных процедур имеет как преимущества, так и недостатки. С одной стороны, вся реализация вложенности сводится к тому, что в стек помещается всего одно дополнительное число, а с другой стороны - обращение к переменным, объявленным на низких уровнях вложенности (а большинство программистов определяет все глобальные переменные на уровне вложенности 0), так же как и вызов процедур, объявленных на низких уровнях вложенности, оказывается достаточно медленным. Многие реализации языков программирования, использующих статические ссылки, помещают переменные, определяемые на уровне 0, не в стек, а в сегмент данных, но тем не менее существует способ, открывающий быстрый доступ к локальным переменным с любых уровней.

5.3.2. *Вложенные процедуры с дисплеями*

Процедурам можно передавать не только адрес одной вышерасположенной активизационной записи, но и набор адресов сразу для всех уровней вложенности - от нулевого до более высокого. При этом доступ к любой нелокальной процедуре сводится всего к двум командам, а перед вызовом процедуры вообще не требуется каких-либо дополнительных действий (так как вызываемая процедура поддерживает дисплей самостоятельно).

```
proc_at_3       proc    near
        push    bp              ; Сохранить динамическую ссылку.
        mov     bp,sp           ; Установить адрес текущей записи.
```

```
        push    display[6]          ; Сохранить предыдущее значение адреса
                                    ; третьего уровня в дисплее.
        mov     display[6],bp       ; Инициализировать третий уровень в дисплее.
        sub  sp,N                   ; Выделить место для локальных переменных.
        [...]
        mov     bx,display[4]       ; Получить адрес записи для уровня 2.
        mov     ax,ss:[bx-6]        ; Считать значение второй
                                    ; переменной из уровня 2.
        [...]
        add    sp,n                 ; Освободить стек от локальных переменных.
        pop     display[6]          ; Восстановить старое
                                    ; значение третьего уровня в дисплее.
        pop     bp
        ret
proc_at_3       endp
```

Здесь считается, что в сегменте данных определен массив слов display, имеющий адреса последних использованных активизационных записей для каждого уровня вложенности: display[0] содержит адрес активизационной записи нулевого уровня, display[2] - первого уровня и так далее (для близких адресов).

Команды ENTER и LEAVE можно использовать для организации вложенности с дисплеями, но в такой реализации дисплей располагается не в сегменте данных, а в стеке, и при вызове каждой процедуры создается его локальная копия.

```
; enter N,4 (уровень вложенности 4, N байтов на стековый кадр) эквивалентно набору команд
        push    bp                  ; Адрес записи третьего уровня.
        push    [bp-2]
        push    [bp-4]
        push    [bp-6]
        push    [bp-8]              ; Скопировать дисплей.
        mov     bp, sp              ;
        add     bp,8                ; BP = адрес начала дисплея текущей записи.
        sub     sp,N                ; Выделить кадр для локальных переменных.
```

Очевидно, что такой метод оказывается крайне неэффективным с точки зрения как скорости выполнения программы, так и расходования памяти. Более того, команда ENTER выполняется дольше, чем соответствующий набор простых команд. Тем не менее существуют ситуации, когда может потребоваться создание локальной копии дисплея для каждой процедуры. Например, если процедура, адрес которой передан как параметр другой процедуре, вызывающейся рекурсивно, должна обращаться к нелокальным переменным. Но и в этом случае передачи всего дисплея через стек можно избежать - более эффективным методом оказываются простые статические ссылки, рассмотренные ранее.

5.4. Целочисленная арифметика повышенной точности

Языки высокого уровня обычно ограничены в наборе типов данных, с которыми они могут работать, - для хранения целых чисел применяются отдельные байты,

слова или двойные слова. Используя ассемблер, можно придумать тип данных совершенно любого размера (64 бита, 128 бит, 1024 бита) и легко определить все арифметические операции с такими числами.

5.4.1. Сложение и вычитание

Команды ADC (сложение с учетом переноса) и SBB (вычитание с учетом займа) специально были введены для подобных операций. При сложении сначала складывают самые младшие байты, слова или двойные слова командой ADD, а затем складывают все остальное командами ADC, двигаясь от младшего конца числа к старшему. Команды SUB/SBB действуют полностью аналогично.

```
bigval_1        dd      0,0,0                   ; 96-битное число
bigval_2        dd      0,0,0
bigval_3        dd      0,0,0

; сложение 96-битных чисел bigval_1 и bigval_2
        mov     eax,dword ptr bigval_1
        add     eax,dword ptr bigval_2          ; Сложить младшие двойные слова.

        mov     dword ptr bigval_3,eax
        mov     eax,dword ptr bigval_1[4]
        adc     eax,dword ptr bigval_2[4]       ; Сложить средние двойные слова.

        mov     dword ptr bigval_3[4],eax
        mov     eax,dword ptr bigval_1[8]
        adc     eax,dword ptr bigval_2[8]       ; Сложить старшие двойные слова.

        mov     dword ptr bigval_3[8],eax

; вычитание 96-битных чисел bigval_1 и bigval_2

        mov     eax,dword ptr bigval_1
        sub     eax,dword ptr bigval_2          ; Вычесть младшие двойные слова.

        mov     dword ptr bigval_3,eax
        mov     eax,dword ptr bigval_1[4]
        sbb     eax,dword ptr bigval_2[4]       ; Вычесть средние двойные слова.

        mov     dword ptr bigval_3[4],ea'x
        mov     eax,dword ptr bigval_1[8]
        sbb     eax,dword ptr bigval_2[8]       ; Вычесть старшие двойные слова.

        mov     dword ptr bigval_3[8],eax
```

5.4.2. Сравнение

Поскольку команда сравнения эквивалентна команде вычитания (кроме того, что она не изменяет значение приемника), можно было бы просто выполнять вычитание чисел повышенной точности и отбрасывать результат, но сравнение выполняется и более эффективным образом. В большинстве случаев для определения результата сравнения достаточно сопоставить самые старшие слова (байты или двойные слова), и только если они в точности равны, потребуется сравнение следующих слов.

```
; Сравнение 96-битных чисел bigval_1 и bigval_2.
        mov     eax,dword ptr bigval_1[8]
        cmp     eax,dword ptr bigval_2[8] ; Сравнить старшие слова.
        jg      greater
        jl      less
        mov     eax,dword ptr bigval_1[4]
        cmp     eax,dword ptr bigval_2[4] ; Сравнить средние слова.
        jg      greater
        jl      less
        mov     eax,dword ptr bigval_1
        cmp     eax,dword ptr bigval_2    ; Сравнить младшие слова.
        jg      greater
        jl      less
equal:
greater:
less:
```

5.4.3. Умножение

Чтобы умножить числа повышенной точности, придется вспомнить правила умножения десятичных чисел в столбик: множимое умножают на каждую цифру множителя, сдвигают влево на соответствующее число разрядов и затем складывают полученные результаты. В нашем случае роль цифр будут играть байты, слова или двойные слова, а сложение должно выполняться по правилам сложения чисел повышенной точности. Алгоритм умножения оказывается заметно сложнее, поэтому умножим для примера только 64-битные числа:

```
; Беззнаковое умножение двух 64-битных чисел (X и Y) и сохранение
; результата в 128-битное число Z.
        mov     eax,dword ptr X
        mov     ebx,eax
        mul     dword ptr Y             ; Перемножить младшие двойные слова.
        mov     dword ptr Z, eax        ; Сохранить младшее слово произведения.
        mov     ecx,edx                 ; Сохранить старшее двойное слово.
        mov     eax,ebx                 ; Младшее слово "X" в eax.
        mul     dword ptr Y[4]          ; Умножить младшее слово на старшее.
        add     eax,ecx
        adc     edx,0                   ; Добавить перенос.
        mov     ebx,eax                 ; Сохранить частичное произведение.
        mov     ecx,edx
        mov     eax,dword ptr X[4]
        mul     dword ptr Y             ; Умножить старшее слово на младшее.
        add     eax,ebx                 ; Сложить с частичным произведением.
        mov     dword ptr Z[4],eax
        adc     ecx,edx
        mov     eax,dword ptr X[4]
        mul     dword ptr Y[4]          ; Умножить старшие слова.
        add     eax,ecx                 ; Сложить с частичным произведением
        adc     edx,0                   ; и добавить перенос.
        mov     word ptr Z[8],eax
        mov     word ptr Z[12],edx
```

Для выполнения умножения со знаком потребуется сначала определить знаки множителей, изменить знаки отрицательных множителей, выполнить обычное умножение и изменить знак результата, если знаки множителей были разными.

5.4.4. *Деление*

Общий алгоритм деления числа любого размера на число любого размера нельзя построить с использованием команды DIV - такие операции выполняются при помощи сдвигов и вычитаний и оказываются весьма сложными. Рассмотрим сначала менее общую операцию (деление любого числа на слово или двойное слово), которую можно легко осуществить с помощью команд DIV:

```
; Деление 64-битного числа divident на 16-битное число divisor.
; Частное помещается в 64-битную переменную quotent,
; а остаток - в 16-битную переменную modulo.
        mov     ax,word ptr divident[6]
        xor     dx,dx
        div     divisor
        mov     word ptr quotent[6],ax
        mov     ax,word ptr divident[4]
        div     divisor
        mov     word ptr quotent[4],ax
        mov     ax,word ptr divident[2]
        div     divisor
        mov     word ptr quotent[2],ax
        mov     ax,word ptr divident
        div     divisor
        mov     word ptr quotent,ax
        mov     modulo,dx
```

Деление любого другого числа полностью аналогично - достаточно только добавить нужное число троек команд mov/div/mov в начало алгоритма.

Наиболее очевидный алгоритм для деления чисел любого размера на числа любого размера — деление в столбик с помощью последовательных вычитаний делителя (сдвинутого влево на нужное количество разрядов) из делимого, увеличивая соответствующий разряд частного на 1 при каждом вычитании, пока не останется число, меньшее делителя (остаток):

```
; Деление 64-битного числа в EDX:EAX на 64-битное число в ECX:EBX.
; Частное помещается в EDX:EAX, и остаток - в ESI:EDI.
        mov     ebp,64  ; Счетчик битов.
        xor     esi,esi
        xor     edi,edi ; Остаток = 0.
bitloop: shl    eax,1
        rcl edx,1
        rcl     edi,1   ; Сдвиг на 1 бит влево 128-битного числа.
        rcl     esi,1   ; ESI:EDI:EDX:EAX.
        cmp     esi,ecx ; Сравнить старшие двойные слова.
        ja      divide
        jb      next
```

```
        cmp     edi,ebx  ; Сравнить младшие двойные слова.
        jb      next
divide:
        sub     edi,ebx
        sbb     esi,ecx  ; ESI:EDI = EBX:ECX.
        inc     eax      ; Установить младший бит в EAX.
next:   dec     ebp      ; Повторить цикл 64 раза.
        jne     bitloop
```

Несмотря на то что этот алгоритм не использует сложных команд, он выполняется на порядок дольше, чем одна команда DIV.

5.5. Вычисления с фиксированной запятой

Существует широкий класс задач, где требуются вычисления с вещественными числами, но не нужна высокая точность результатов. Например, в этот класс задач попадают практически все процедуры, оперирующие с координатами и цветами точек в дву- и трехмерном пространстве. Так как в результате все выведется на экран с ограниченным разрешением и каждый компонент цвета будет записываться как 6- или 8-битное целое число, все те десятки знаков после запятой, которые вычисляет FPU, не нужны. А раз не нужна высокая точность, вычисление можно выполнить значительно быстрее. Чаще всего для представления вещественных чисел с ограниченной точностью используется формат чисел с фиксированной запятой: целая часть числа представляется в виде обычного целого числа, и дробная часть - точно так же в виде целого числа (как мы записываем небольшие вещественные числа на бумаге).

Наиболее распространенные форматы для чисел с фиксированной запятой - 8:8 и 16:16. В первом случае на целую и на дробную части числа отводится по одному байту, а во втором - по одному слову. Операции с этими двумя форматами можно выполнять, помещая число в регистр (16-битный - для формата 8:8 и 32-битный - для формата 16:16). Разумеется, можно придумать и использовать совершенно любой формат, например 5:11, но некоторые операции над такими числами могут усложниться.

5.5.1. Сложение и вычитание

Сложение и вычитание для чисел с фиксированной запятой ничем не отличается от сложения и вычитания целых чисел:

```
mov     ax,1080h        ; AX = 1080h = 16,5
mov     bx,1240h        ; BX = 1240h = 18,25
add     ax,bx           ; AX = 2200h = 34,75
sub     ax,bx           ; AX = 1080h = 16,5
```

5.5.2. Умножение

При выполнении этого действия следует просто помнить, что умножение 16-битных чисел дает 32-битный результат, а умножение 32-битных чисел - 64-битный результат. Например, пусть EAX и EBX содержат числа с фиксированной запятой в формате 16:16:

```
        xor     edx,edx
        mul     ebx             ; Теперь EDX:EAX содержит 64-битный результат
;  (EDX содержит всю целую часть, а EAX - всю дробную).
        shrd    eax,edx,16      ; Теперь EAX содержит ответ, если не
;  произошло переполнение (то есть если результат не превысил 65 535).
```

Аналогом IMUL в таком случае будет последовательность команд

```
        imul    ebx
        shrd    eax,edx,16
```

5.5.3. Деление

Число, записанное с фиксированной запятой в формате 16:16, можно представить как число, умноженное на 2^{16}. Если разделить такие числа друг на друга сразу — мы получим результат деления целых чисел: $(A \times 2^{16})/(B \times 2^{16}) = A/B$. Чтобы результат имел нужный нам вид $(A/B) \times 2^{16}$, надо заранее умножить делимое на 2^{16}:

```
; Деление числа с фиксированной запятой в формате 16:16
; в регистре EAX на такое же число в EBX, без знака:
        xor     edx,edx
        ror     eax,16
        xchg    ax,dx           ; EDX:EAX = EAX x 2^16
        div     ebx             ; EAX = результат деления.

; Деление числа с фиксированной запятой в формате 16:16
; в регистре EAX на такое же число в EBX, со знаком:
        cdq
        ror     eax,16
        xchg    ax,dx           ; EDX:EAX = EAX x 2^16
        idiv    ebx             ; EAX = результат деления.
```

5.5.4. Трансцендентные функции

Многие операции при работе с графикой используют умножение числа на синус (или косинус) некоторого угла, например при повороте: s = sin(n) x v. При вычислении с фиксированной запятой 16:16 это уравнение преобразуется в s = int(sin(n) x x 65 536) x v/65 536 (где int - целая часть). Для требовательных ко времени работы участков программ, например для работы с графикой, значения синусов принято считывать из таблицы, содержащей результаты выражения int(sin(n) x 65 535), где n меняется от 0 до 90 градусов с требуемым шагом (редко требуется шаг меньше 0,1 градуса). Затем синус любого угла от 0 до 90 градусов можно вычислить с помощью всего одного умножения и сдвига на 16 бит. Синусы и косинусы других углов вычисляются в соответствии с обычными формулами приведения:

```
sin(x) = sin(180-x) для 90 < x < 180
sin(x) = -sin(x-180) для 180 < x < 270
sin(x) = -sin(360-x) для 270 < x < 360
cos(x) = sin(90-x)
```

хотя часто используют таблицу синусов на все 360 градусов, устраняя дополнительные проверки и изменения знаков в критических участках программы.

Таблицы синусов (или косинусов), используемые в программе, можно создать заранее с помощью простой программы на языке высокого уровня в виде текстового файла с псевдокомандами DW и включить в текст программы директивой include. Другой способ, занимающий меньше места в тексте, но чуть больше времени при запуске программы, - однократное вычисление всей таблицы. Таблицу можно вычислять как с помощью команды FPU fsin и потом преобразовывать к желаемому формату, так и сразу в формате с фиксированной запятой. Существует довольно популярный алгоритм, позволяющий вычислить таблицу косинусов (или синусов, с небольшой модификацией), используя рекуррентное выражение

$$\cos(x_k) = 2\cos(step)\cos(x_{k-1}) - \cos(x_{k-2}),$$

где step - шаг, с которым вычисляются косинусы, например 0,1 градуса.

```
; liss.asm
; Строит фигуры Лиссажу, используя арифметику с фиксированной запятой
; и генерацию таблицы косинусов.
; Фигуры Лиссажу - семейство кривых, задаваемых параметрическими выражениями
; x(t) = cos(SCALE_V x t)
; y(t) = sin(SCALE_H x t)
;
; Чтобы выбрать новую фигуру, измените параметры SCALE_H и SCALE_V,
; для построения незамкнутых фигур удалите строку add di,512 в процедуре move_point.

        .model  tiny
        .code
        .386                    ; Будут использоваться 32-битные регистры.
        org     100h            ; COM-программа.

SCALE_H equ     3               ; Число периодов в фигуре по горизонтали.
SCALE_V equ     5               ; Число периодов по вертикали.

start   proc    near
        cld                     ; Для команд строковой обработки.

        mov     di,offset cos_table ; Адрес начала таблицы косинусов.
        mov     ebx,16777137    ; 224 x cos(360/2048) - заранее вычисленное
        mov     cx,2048         ; число элементов для таблицы.
        call    build_table     ; Построить таблицу косинусов.

        mov     ax,0013h        ; Графический режим
        int     10h             ; 320x200x256.

        mov     ax,1012h        ; Установить набор регистров палитры VGA,
        mov     bx,70h          ; начиная с регистра 70h.
        mov     cx,4                    ; Четыре регистра.
        mov     dx,offset palette       ; Адрес таблицы цветов.
        int     10h

        push    0A000h                  ; Сегментный адрес видеопамяти
        pop     es                      ; в ES.

main_loop:
        call    display_picture         ; Изобразить точку со следом.
```

```
        mov     dx,5000
        xor     cx,cx
        mov     ah,86h
        int     15h                     ; Пауза на CX:DX микросекунд.

        mov     ah,11h                  ;, Проверить, была ли нажата клавиша.
        int     16h
        jz      main_loop               ; Если нет - продолжить основной цикл.

        mov     ax,0003h                ; Текстовый режим
        int     10h                     ; 80x24.

        ret                             ; Конец программы.
start   endp

; Процедура build_table.
; Строит таблицу косинусов в формате с фиксированной запятой 8:24
; по рекуррентной формуле cos(x_k) = 2 x c'os(span/steps) x cos(x_k-1) - cos(x_k-2),
; где span - размер области, на которой вычисляются косинусы (например, 360),
; а steps - число шагов, на которые разбивается область.
; Вход: DS:DI = адрес таблицы
;       DS:[DI] = 224
;       EBX = 224 x cos(span/steps)
;       CX = число элементов таблицы, которые надо вычислить
; Выход: таблица размером CX x 4 байта заполнена
; Модифицируются: DI,CX,EAX,EDX

build_table     proc    near
        mov     dword ptr [di+4],ebx    ; Заполнить второй элемент таблицы.
        sub     cx,2                    ; Два элемента уже заполнены.
        add     di,8
        mov     eax,ebx
build_table_loop:
        imul    ebx                     ; Умножить cos(span/steps) на cos(x_k-1).
        shrd    eax,edx,23              ; Поправка из-за действий с фиксированной
                                        ; запятой 8:24 и умножение на 2.
        sub     eax,dword ptr [di-8]    ; Вычитание cos(x_k-2).
        stosd                           ; Запись результата в таблицу.
        loop    build_table_loop
        ret
build_table     endp

; Процедура display_picture.
; Изображает точку со следом.

display_picture proc    near

        call    move_point              ; Переместить точку.

        mov     bp,73h                  ; Темно-серый цвет в нашей палитре.
        mov     bx,3                    ; Точка, выведенная три шага назад.
        call    draw_point              ; Изобразить ее.
        dec     bp                      ; 72h - серый цвет в нашей палитре.
        dec     bx                      ; Точка, выведенная два шага назад.
```

```
        call    draw_point      ; Изобразить ее.
        dec     bp              ; 71h - светло-серый цвет в нашей палитре.
        dec     bx              ; Точка, выведенная один шаг назад.
        call    draw_point      ; Изобразить ее.
        dec     bp              ; 70h - белый цвет в нашей лалитре.
        dec     bx              ; Текущая точка.
        call    draw_point      ; Изобразить ее.
        ret
display_picture  endp

; Процедура  draw_point.
; Вход: BP - цвет
;       BX - сколько шагов назад выводилась точка
;
draw_point      proc near
        movzx   cx,byte ptr point_x[bx]  ; X-координата.
        movzx   dx,byte ptr point_y[bx]  ; Y-координата.
        call    putpixel_13h             ; Вывод точки на экран.
        ret
draw_point      endp

; Процедура move_point.
; Вычисляет координаты для следующей точки.
; Изменяет координаты точек, выведенных раньше.

move_point      proc near
        inc     word ptr time
        and     word ptr time, 2047      ; Эти две команды организуют счетчик
                ; в переменной time, который изменяется от 0 до 2047 (7FFh).

        mov     eax,dword ptr point_x    ; Считать координаты точек
        mov     ebx,dword ptr point_y    ; (по байту на точку)
        mov     dword ptr point_x[1],eax ; и записать их со сдвигом
        mov     dword ptr point_y[1],ebx ; 1 байт.

        mov     di,word ptr time         ; Угол (или время) в DI.
        imul    di,di,SCALE_H            ; Умножить его на SCALE_H.
        and     di,2047                  ; Остаток от деления на 2048,
        shl     di,2                     ; так как в таблице 4 байта на косинус.
        mov     ax,50                    ; Масштаб по горизонтали.
        mul     word ptr cos_table[di+2] ; Умножение на косинус: берется старшее
                ; слово (смещение + 2) от косинуса, записанного в формате 8:24.
                ; Фактически происходит умножение на косинус в формате 8:8.
        mov     dx,0A000h                ; 320/2 (X центра экрана) в формате 8:8.
        sub     dx,ax                    ; Расположить центр фигуры в центре экрана
        mov     byte ptr point_x,dh      ; и записать новую текущую точку.

        mov     di,word ptr time         ; Угол (или время) в DI.
        imul    di,di,SCALE_V            ; Умножить его на SCALE_V.
        add     di,512                   ; Добавить 90 градусов, чтобы заменить
                ; косинус на синус. Так как у нас 2048 шагов на 360 градусов,
                ; 90 градусов - это 512 шагов.
        and     di,2047                  ; Остаток от деления на 2048,
```

```
        shl     di,2                    ; так как в таблице 4 байта на косинус.
        mov     ax,50                   ; Масштаб по вертикали.
        mul     word 'ptr  cos_table[di+2] ; Умножение на косинус.
        mov     dx,06400h               ; 200/2 (Y центра экрана) в формате 8:8.
        sub     dx,ax                   ; Расположить центр фигуры в центре экрана
        mov     byte ptr point_y,dh     ; и записать новую текущую точку.
        ret
move_point      endp

; putpixel_13h
; Процедура вывода точки на экран в режиме 13h.
; DX = строка, CX = столбец, BP = цвет, ES = A000h
putpixel_13h    proc    near
        push    di
        mov     ax,dx           ; Номер строки.
        shl     ax,8            ; Умножить на 256.
        mov     di,dx
        shl     di,6            ; Умножить на 64
        add     di,ax           ; и сложить - то же, что и умножение на 320.
        add     di,cx           ; Добавить номер столбца.
        mov     ax,bp
        stosb                   ; Записать в видеопамять.
        pop     di
        ret
putpixel_13h    endp

point_x         db      0FFh,0FFh,0FFh,0FFh     ; X-координаты точки и хвоста.
point_y         db      0FFh,0FFh,0FFh,0FFh     ; Y-координаты точки и хвоста.
                db      ?               ; Пустой байт - нужен для команд
                                        ; сдвига координат на один байт.
time            dw      0               ; Параметр в уравнениях Лиссажу - время
                                        ; или угол.

palette         db      3Fh,3Fh,3Fh     ; Белый.
                db      30h,30h,30h     ; Светло-серый.
                db      20h,20h,20h     ; Серый.
                db      10h,10h,10h     ; Темно-серый.

cos_table       dd      1000000h ; Здесь начинается таблица косинусов.
        end     start
```

При генерации таблицы использовались 32-битные регистры, что приводит к увеличению на 1 байт и замедлению на 1 такт каждой команды, применяющей их в 16-битном сегменте, но на практике большинство программ, интенсивно работающих с графикой, - 32-битные.

5.6. Вычисления с плавающей запятой

Набор команд для работы с плавающей запятой в процессорах Intel достаточно разнообразен, чтобы реализовывать весьма сложные алгоритмы, и прост в использовании. Единственное, что может представлять определенную сложность, - почти

все команды FPU по умолчанию работают с его регистрами данных как со стеком, выполняя операции над числами в ST(0) и ST(1) и помещая результат в ST(0), так что естественной формой записи математических выражений для FPU оказывается *обратная польская нотация* (RPN). Эта форма ..аписи встречается в программируемых калькуляторах, языке Форт и почти всегда неявно присутствует во всех алгоритмах анализа математических выражений: они сначала преобразовывают обычные выражения в обратные и только потом начинают их анализ. В обратной польской нотации все операторы указываются после своих аргументов, так что sin(x) превращается в x sin, a a + b превращается в a b +. При этом полностью пропадает необходимость использовать скобки, например: выражение (a + b) x 7 - d записывается как a b + 7 x d -.

Посмотрим, как выражение, записанное в RPN, на примере процедуры вычисления арксинуса легко преобразовывается с помощью команд FPU.

```
; asin
; Вычисляет арксинус числа, находящегося в st(0) (-1 < x < +1),
; по формуле asin(x) = atan(sqrt(x²/(1-x²)))
; (в RPN: x x * x x * 1 - / sqrt atan).
; Результат возвращается в st(0), в стеке FPU должно быть два свободных регистра.
asm     proc    near      Комментарий показывает содержимое стека FPU:.
                        ; Первое выражение - ST(0), второе - ST(1) и т. д.
                        ; x (начальное состояние стека)
        fld     st(0)   ; x, x
        fmul            ; x²
        fld     st(0)   ; x², x²
        fld1            ; 1, x², x²
        fsubr           ; 1-x², x²
        fdiv            ; x²/(1-x²)
        fsqrt           ; sqrt(x²/(1-x²))
        fld1            ; 1, sqrt(x²/(1-x²))
        fpatan          ; atan(sqrt(x²/(1-x²)))
        ret
asin    endp
```

Теперь попробуем решить небольшое дифференциальное уравнение - уравнение Ван-дер-Поля для релаксационных колебаний:

$$x'' = -x + m(1-x^2)x', \quad \tau > 0$$

будем двигаться по времени с малым шагом h, так что

$$x(t + h) = x(t) + hx(t)'$$
$$x(t + h)' = x(t)' + hx(t)''$$

или, сделав замену $y = x'$,

$$y = y + h(m(1-x^2)y - x)$$
$$x = x + hy$$

Решение этого уравнения для всех $m > 0$ оказывается периодическим аттрактором, поэтому, если из-за ошибок округления решение отклоняется от истинного

в любую сторону, оно тут же возвращается обратно. При m = 0, наоборот, решение оказывается неустойчивым и ошибки округления приводят к очень быстрому росту x и y до максимально допустимых значений для вещественных чисел.

Эту программу нельзя реализовать в целых числах или числах с фиксированной запятой, потому что значения x и x' различаются на много порядков - кривая содержит почти вертикальные участки, особенно при больших t.

```
; vdp.asm
; Решение уравнения Ван-дер-Поля
; x(t)'' = -x(t) + m(1-x(t)²)x(t)'
; с t = 0, 1, 2, 3, 4, 5, 6, 7, 8.
;
; Программа выводит на экран решение с m = 1, нажатие клавиш 0-8 изменяет t.
; Esc - выход, любая другая клавиша - пауза до нажатия одной из Esc, 0-8.

        .model tiny
        .286                    ; Для команд pusha и popa.
        .287                    ; Для команд FPU.
        .code
        org     100h            ; COM-программа.

start   proc    near
        cld
        push    0A000h
        pop     es              ; Адрес видеопамяти в ES.

        mov     ax,0012h
        int     10h             ; Графический режим 640x480x16.

        finit                   ; Инициализировать FPU.

        xor     si, si          ; SI будет содержать координату t и меняться
                                ; от 0 до 640.
        fld1                    ; 1
        fild    word ptr hinv   ; 32, 1
        fdiv                    ; h             (h = 1/hinv)
; Установка начальных значений для _display:
; m = 1, x = h = 1/32, y = x' = 0
again:  fild    word ptr m              ; m, h
        fld     st(1)                   ; x, m, h       (x = h)
        fldz                            ; y, x, m, h    (y = 0)
        call    _display                ; Выводить на экран решение, пока
                                        ; не будет нажата клавиша.
g_key:  mov     ah,10h                  ; Чтение клавиши с ожиданием.
        int     16h                     ; Код нажатой клавиши в AL.
        cmp     al,1Bh                  ; Если это Esc,
        jz      g_out                   ; выйти из программы.
        cmp     al,'0'                  ; Если код меньше "0",
        jb      g_key                   ; пауза/ожидание следующей клавиши.
        cmp     al,'8'                  ; Если код больше "8",
        ja      g_key                   ; пауза/ожидание следующей клавиши.
        sub     al,'0'                  ; Иначе: AL = введенная цифра,
```

```
        mov     byte ptr m,al           ; m = введенная цифра.
        fstp    st(0)                   ; x, m, h
        fstp    st(0)                   ; m, h
        fstp    st(0)                   ; h
        jmp short  again

g_out:  mov     ax,0003h                ; Текстовый режим.
        int     10h
        ret                             ; Конец программы.
start   endp

; Процедура display_.
; Пока не нажата клавиша, выводит решение на экран, делая паузу после каждой -из
; 640 точек.
;
_display        proc    near
dismore:
        mov     bx,0                    ; Стереть предыдущую точку: цвет = 0.
        mov     cx,si
        shr     cx,1                    ; CX - строка.
        mov     dx,240
        sub     dx,word ptr ix[si]      ; DX - столбец.
        call    putpixel1b
        call    next_x                  ; Вычислить x(t) для следующего t.
        mov     bx,1                    ; Вывести точку: цвет = 1.
        mov     dx,240
        sub     dx,word ptr ix[si]      ; DX - столбец.
        call    putpixel1b
        inc     si
        inc     si                      ; SI = SI + 2 (массив слов).
        cmp     si,640*2                ; Если SI достигло конца массива IX,
        jl      not_endscreen           ; пропустить паузу.
        sub     si,640*2                ; Переставить SI на начало массива IX.
not_endscreen:
        mov     dx,5000 •,
        xor     cx,cx
        mov     ah,86h
        int     15h                     ; Пауза на CX:DX микросекунд.

        mov     ah,11h
        int     16h                     ; Проверить, была ли нажата клавиша.
        jz      dismore                 ; Если нет - продолжить вывод на экран.
        ret                             ; Иначе - закончить процедуру.
_display        endp

; Процедура next_x.
; Проводит вычисления по формулам:
; y = y + h(m(1-x^2)y-x)
; x = x + by
; Вход: st = y, st(1) = x, st(2) = m, st(3) = h.
; Выход: st = y, st(1) = x, st(2) = m, st(3) = h, x * 100 записывается в ix[si].
```

```
next_x          proc    near
        fld1                    ; 1, y, x, m, h
        fld     st(2)           ; x, 1, y, x, m, h
        fmul    st,st(3)        ; x2, 1, y, x, m, h
        fsub                    ; (1-x2), y, x, m, h
        fld     st(3)           ; m, (1-x2), y, x, m, h
        fmul                    ; M, y, x, m, h (M = m(1-x2))
        fld     st(1)           ; y, M, y, x, m, h
        fmul                    ; My, y, x, m, h
        fld     st(2)           ; x, My, y, x, m, h
        fsub                    ; My-x, y, x, m, h
        fld     st(4)           ; h, My-x, y, x, m, h
        fmul                    ; h(My-x), y, x, m, h
        fld     st(1)           ; y, h(My-x), y, x, m, h
        fadd                    ; Y, y, x, m, h (Y = y + h(My-x))
        fxch                    ; y, Y, x, m, h
        fld     st(4)           ; h, y, Y, x, m, h
        fmul                    ; yh, Y, x, m, h
        faddp   st(2),st        ; Y, X, m, h            (X = x + hy)
        fld     st(1)           ; X, Y, X, m, h
        fild    word ptr c_100  ; 100, X, Y, X, m, h
        fmul                    ; 100X, Y, X, m, h
        fistp   word ptr ix[si] ; Y, X, m, h
        ret
next_x          endp

; Процедура вывода точки на экран в режиме, использующем 1 бит на пиксел.
; DX = строка, CX = столбец, ES = A000h, BX = цвет (1 - белый, 0 - черный).
; Все регистры сохраняются.

putpixel1b      proc    near
        pusha                   ; Сохранить регистры.
        push    bx
        xor     bx,bx
        mov     ax,dx           ; AX = номер строки.
        imul    ax,ax,80        ; AX = номер строки x число байтов в строке.
        push    cx
        shr     cx,3            ; CX = номер байта в строке.
        add     ax,cx           ; AX = номер байта в видеопамяти.
        mov     di,ax           ; Поместить его в DI и SI.
        mov     si,di

        pop     cx              ; CX снова содержит номер столбца.
        mov     bx,0080h
        and     cx,07h          ; Последние три бита CX =
                                ; остаток от деления на 8 =
                                ; номер бита в байте, считая справа налево.
        shr     bx,cl           ; Теперь нужный бит в BL установлен в 1.
        lods    es:byte ptr ix  ; AL = байт из видеопамяти.
        pop     dx
        dec     dx              ; Проверить цвет.
```

```
        is      black           ; Если 1 - '
        or      ax,bx           ; установить выводимый бит в 1.
        jmp     short white
black:  not     bx              ; Если 0 -
        and     ax,bx           ; установить выводимый цвет в 0
white:
        stosb                   ; и вернуть байт на место.
        popa                    ; Восстановить регистры.
        ret                     ; Конец.
putpixel1b      endp

m               dw      1       ; Начальное значение т.
c_100           dw      100     ; Масштаб по вертикали.
hinv            dw      32      ; Начальное значение 1/h.
ix:                             ; Начало буфера для значений x(t)
                                ; (всего 1280 байт за концом программы).
        end     start
```

5.7. Популярные алгоритмы

5.7.1. Генераторы случайных чисел

Самый часто применяемый тип алгоритмов генерации псевдослучайных последовательностей - линейные конгруэнтные генераторы, описываемые общим рекуррентным соотношением:

$$I_{j+1} = (aI_j + c) \text{ MOD } m.$$

При правильно выбранных числах a и c эта последовательность возвращает все числа от нуля до m−1 псевдослучайным образом и ее периодичность сказывается только на последовательностях порядка т. Такие генераторы очень легко реализуются и работают быстро, но им присущи и некоторые недостатки: самый младший бит намного менее случаен, чем, например, самый старший, а также, если попытаться использовать результаты работы этого генератора для заполнения k-мерного пространства, начиная с некоторого k, точки будут лежать на параллельных плоскостях. Оба недостатка можно устранить, используя так называемое перемешивание данных: числа, получаемые при работе последовательности, не выводятся сразу, а помещаются в случайно выбранную ячейку небольшой таблицы (8–16 чисел); число, находившееся в этой ячейке раньше, возвращается как результат работы функции.

Если число a подобрано очень тщательно, может оказаться, что число c равно нулю. Так, классический стандартный генератор Льюиса, Гудмана и Миллера использует a = 16 807 (7^5) при m = 2^{31}−1, а генераторы Парка и Миллера используют a = 48 271 и a = 69 621 (при том же т). Любой из этих генераторов можно легко применить в ассемблере для получения случайного 32-битного числа, достаточно всего двух команд - MUL и DIV.

```
; Процедура rand.
; Возвращает в EAX случайное положительное 32-битное число (от 0 до 2^31-2).
;
```

```
rand    proc    near
        push    'edx
        mov     eax,dword ptr seed      ; Считать последнее
                                        ; случайное число.
        test    eax,eax                 ; Проверить его, если это -1,
        js      fetch_seed              ; функция еще ни разу не
                                        ; вызывалась и надо создать
                                        ; начальное значение.
randomize:
        mul     dword ptr rand_a        ; Умножить на число a.
        div     dword ptr .rand_m       ; Взять остаток от деления на 2^31-1.
        mov     eax,edx
        mov     dword ptr seed,eax      ; Сохранить для следующих вызовов.
        pop     edx
        ret

fetch_seed:
        push    ds
        push    0040h
        pop     ds
        mov     eax,dword ptr ds:006Ch  ; Считать двойное слово из области
        pop     ds                      ; данных BIOS по адресу
                                        ; 0040:006C - текущее число
        jmp     short randomize         ; тактов таймера.

rand_a          dd      69621
randjn          dd      7FFFFFFFh
seed            dd      -1
rand            endp
```

Если период этого генератора (порядка 10^9) окажется слишком мал, можно скомбинировать два генератора с разными a и т, не имеющими общих делителей, например: a_1 = 400 014 с m_1 = 2 147 483 563 и a_2 = 40 692 с m_2 = 2 147 483 399. Генератор, работающий по уравнению

$$I_{j+1} = (a_1 I_j + a_2 I_j) \text{ MOD } m,$$

где m - любое из m_1 и m_2, имеет период 2,3 x 10^{18}.

Очевидный недостаток такого генератора - команды MUL и DIV относятся к самым медленным. От DIV можно избавиться, используя один из генераторов с ненулевым числом **c и m**, равным степени двойки (тогда DIV m заменяется на AND m−1), например: a = 25 173, c = 13849, m = 2^{16} или a = 1 664 525, c = 1 013904 223, m = 2^{32}, однако проще перейти к методам, основанным на сдвигах или вычитаниях.

Алгоритмы, основанные на вычитаниях, не так подробно изучены, как конгруэнтные, но из-за большой скорости широко используются и, по-видимому, не имеют заметных недостатков. Детальное объяснение алгоритма этого генератора (а также алгоритмов многих других генераторов случайных чисел) приведено в книге Кнута Д. Е. «Искусство программирования» (т. 2).

```
; Процедура srand_init.
; Инициализирует кольцевой буфер для генератора, использующего вычитания.
; Вход: EAX - начальное значение, например из области
; данных BIOS, как в предыдущем примере.
srand_init      proc    near
        push    bx
        push    si
        push    edx
        mov     edx,1
; Засеять кольцевой буфер.
        mov     bx,216
do_0:   mov     word ptr ablex[bx],dx
        sub     eax,edx
        xchg    eax,edx
        sub     bx,4
        jge     do_0

; Разогреть генератор.
        mov     bx,216
do_1:   push    bx
do_2:   mov     si,bx
        add     si,120
        cmp     si,216
        jbe     skip
        sub     si,216
skip:   mov     eax,dword ptr tablex[bx]
        sub     eax,dword ptr tablex[si]
        mov     dword ptr tablex[bx],eax
        sub     bx,4
        jge     do_2
        pop     bx
        sub     bx,4
        jge     do_1

; Инициализировать индексы.
        sub     ax,ax
        mov     word ptr index0,ax
        mov     ax,124
        mov     index1,ax

        pop     edx
        pop     si
        pop     bx
        ret
srand_init      endp

; Процедура srand.
; Возвращает случайное 32-битное число в EAX (от 0 до 2^32-1).
; Перед первым вызовом этой процедуры должна быть один раз вызвана процедура srand_init.
srand           proc    near
        push    bx
        push    si
```

```
        mov     bx,word ptr index0
        mov     si,word ptr index1      ; Считать индексы.
        mov     eax,dword ptr tablex[bx]
        sub     eax,dword ptr tablex[si] ; Создать новое случайное число.
        mov     dword ptr tablex[si],eax ; Сохранить его в кольцевом
                                         ; буфере.
        sub     si,4                    ; Уменьшить индексы,
        jl      fix_si                  ; перенося их на конец буфера,
fixed_si:mov    word ptr index1,si      ; если они выходят за начало.
        sub     bx,4
        jl      fix_bx
fixed_bx:mov    index0,bx
        pop     si
        pop     bx
        ret

fix_SI: mov     si,216
        jmp     short fixed_SI
fix_BX: mov     bx,216
        jmp     short fixed_BX

srand           endp

tablex          dd      55 dup (?)      ; Кольцевой буфер случайных чисел.
index0          dw      7               ; Индексы для кольцевого буфера.
index1          dw      9
```

Часто необходимо получить всего один или несколько случайных битов, а генераторы, работающие с 32-битными числами, оказываются неэффективными. В таком случае удобно применять алгоритмы, основанные на сдвигах:

```
; rand8
; Возвращает случайное 8-битное число в AL.
; Переменная seed должна быть инициализирована заранее,
; например из области данных BIOS, как в примере для конгруэнтного генератора.
rand8           proc    near
        mov     ax, word ptr seed
        mov     cx,8
newbit: mov     bx,ax
        and     bx,002Dh
        xor     bh,bl
        clc
        jpe     shift
        stc
shift:  rcr     ax,1
        loop    newbit
        mov     word ptr seed, ax
        mov     ah,0
        ret
rand8           endp
seed            dw      1
```

5.7.2.Сортировки

Еще одна часто встречающаяся задача при программировании — сортировка данных. Все существующие алгоритмы сортировки можно разделить на сортировки перестановкой, в которых на каждом шаге алгоритма меняется местами пара чисел; сортировки выбором, в которых на каждом шаге выбирается наименьший элемент и дописывается в отсортированный массив; и сортировки вставлением, в которых элементы массива рассматривают последовательно и каждый вставляют на подходящее место в отсортированном массиве. Самая простая сортировка перестановкой - пузырьковая, в которой более легкие элементы «всплывают» к началу массива: сначала второй элемент сравнивается с первым и, если нужно, меняется с ним местами; затем третий элемент сравнивается со вторым и только в том случае, когда они переставляются, сравнивается с первым, и т. д. Этот алгоритм также является и самой медленной сортировкой - в худшем случае для сортировки массива N чисел потребуется $N^2/2$ сравнений и перестановок, а в среднем - $N^2/4$.

```
; Процедура bubble_sort.
; Сортирует массив слов методом пузырьковой сортировки.
; Вход: DS:DI = адрес массива
;       DX = размер массива (в словах)
bubble_sort     proc    near
        pusha
        cld
        cmp     dx,1
        jbe     so'rt_exit          ; Выйти, если сортировать нечего.
        dec     dx
sb_loop1:mov    cx,dx               ; Установить длину цикла.
        xor     bx,bx               ; BX будет флагом обмена.
        mov     si,di               ; SI будет указателем на
                                    ; текущий элемент.
sn_loop2:lodsw                      ; Прочитать следующее слово.
        cmp     ax,word ptr [si]
        jbe     no_swap             ; Если элементы не в порядке,
        xchg    ax,word ptr [si]    ; поменять их местами
        mov     word ptr [si-2],ax
        inc     bx                  ; и установить флаг в 1.
no_swap: loop   sn_loop2
        cmp     bx,0                ; Если сортировка не закончилась,
        jne     sn_loop1            ; перейти к следующему элементу.
sort_exit:popa
        ret
bubble_sort     endp
```

Пузырьковая сортировка осуществляется так медленно потому, что сравнения выполняются лишь между соседними элементами. Чтобы получить более быстрый метод сортировки перестановкой, следует выполнять сравнение и перестановку элементов, отстоящих далеко друг от друга. На этой идее основан алгоритм, который называется *быстрой сортировкой*. Он работает следующим образом: делается предположение, что первый элемент является средним по отношению

к остальным. На основе такого предположения все элементы разбиваются на две группы - больше и меньше предполагаемого среднего. Затем обе группы отдельно сортируются таким же методом. В худшем случае быстрая сортировка массива из N элементов требует N^2 операций, но в среднем случае - только $2n\log_2 n$ сравнений и еще меньшее число перестановок.

```
; Процедура quick_sort.
; Сортирует массив слов методом быстрой сортировки.
; Вход: DS:BX - адрес массива
;       DX = число элементов массива

quick_sort      proc    near
        cmp     dx,1            ; Если число элементов 1 или 0,
        jle     qsort_done      ; то сортировка уже закончилась.
        xor     di,di           ; Индекс для просмотра сверху (DI = 0).
        mov     si,dx           ; Индекс для просмотра снизу (SI = DX).
        dec     si              ; SI = DX-1, так как элементы
                                ; нумеруются с нуля,
        shl     si,1            ; и умножить на 2, так как
                                ; это массив слов.
        mov     ax,word ptr [bx] ; AX = элемент X. объявленный средним.

step_2: ; Просмотр массива снизу, пока не встретится элемент,
        ; меньший или равный X.
        cmp     word ptr [bx][si],ax    ; Сравнить X_DI и X_1.
        jle     step_3                  ; Если X_SI больше, перейти
        sub     si,2                    ; к следующему снизу элементу
        jmp     short step_2            ; и продолжить просмотр.

step_3: ; Просмотр массива сверху, пока не встретится элемент
        ; меньше X_1 или оба просмотра не придут в одну точку.
        cmp     si,di                   ; Если просмотры встретились,
        je      step_5                  ; перейти к шагу 5.
        add     di,2                    ; Иначе: перейти
                                        ; к следующему сверху элементу.
        cmp     word ptr [bx][di],ax    ; Если он меньше X_1,
        jl      step_3                  ; продолжить шаг 3.

step_4: ; DI указывает на элемент, который не должен быть в верхней части,
        ; SI указывает на элемент, который не должен быть в нижней части.
        ; Поменять их местами.
        mov     cx,word ptr [bx][di]    ; CX = X_DI
        xchg    cx,word ptr [bx][si]    ; CX = X_SI, X_SI = X_DI
        mov     word ptr [bx][di],cx    ; X_DI = CX
        jmp     short step_2

step_5: ; Просмотры встретились. Все элементы в нижней группе больше X_1,
        ; все элементы в верхней группе и текущий - меньше или равны X_1.
        ; Осталось поменять местами X_1 и текущий элемент:
        xchg    ax,word ptr [bx][di]    ; AX = X_DI, X_DI = X_1
        mov     word ptr [bx],ax        ; X. = AX
```

```
        ; Теперь можно отсортировать каждую из полученных групп.
        push    dx
        push    di
        push    bx

        mov     dx,di                   Длина массива X1...XDI-1
        shr     dx, 1                   в DX.
        call    quick_sort              Сортировка.

        pop     bx
        pop     di
        pop     dx

        add     bx,di                ; Начало массива XDI+1...XN
        add     bx,2                 ; в BX.
        shr     di,1                 ; Длина массива XDI+1...XN
        inc     di
        sub     dx,di                ; в DX.
        call    quick_sort           ; Сортировка.
qsort_done:ret
quick_sort      endp
```

Помимо того, что быстрая сортировка - самый известный пример алгоритма, использующего рекурсию, то есть вызывающего самого себя, это еще и самая быстрая из сортировок «на месте», то есть сортировка, применяющая только ту память, в которой хранятся элементы сортируемого массива. Можно доказать, что сортировку нельзя выполнить быстрее, чем за $n\log_2 n$ операций, ни в худшем, ни в среднем случаях, и быстрая сортировка хорошими темпами приближается к этому пределу в среднем случае. Сортировки, достигающие теоретического предела, тоже существуют - это сортировки *турнирным выбором* и сортировки *вставлением в сбалансированные деревья,* но для их работы требуется резервирование дополнительной памяти, так что, например, работа со сбалансированными деревьями будет происходить медленно из-за дополнительных затрат на поддержку сложных структур данных в памяти.

Приведем в качестве примера самый простой вариант сортировки вставлением, использующей линейный поиск и затрачивающей порядка $n^2/2$ операций. Ее так же просто реализовать, как и пузырьковую сортировку, и она тоже имеет возможность выполняться «на месте». Кроме того, из-за высокой оптимальности кода этой процедуры она может оказаться даже быстрее рассмотренной нами «быстрой» сортировки на подходящих массивах.

```
; Процедура linear_selection_sort.
; Сортирует массив слов методом сортировки линейным выбором.
; Вход: DS:SI (и ES:SI) = адрес массива
;       DX = число элементов в массиве

do_swap:lea     bx, word ptr [di-2]
        mov     ax, word ptr [bx]       ; Новое минимальное число.
        dec     cx                      ; Если поиск минимального закончился,
        jcxz    tail                    ; перейти к концу.
```

```
loop1:  scasw                   ; Сравнить минимальное в AX
                                ; со следующим элементом массива.
        ja      do_swap         ; Если найденный элемент еще меньше -
                                ; выбрать его как минимальный.
        loop    loop1           ; Продолжить сравнения
                                ; с минимальным элементом в AX.
tail:   xchg    ax,word ptr [si-2]      ; Обменять минимальный элемент
        mov     word ptr [bx],ax ; с элементом, находящимся
                                        ; в начале массива.

linear_selection_sort   proc    near    ; Точка входа в процедуру.
        mov     bx, si          ; BX содержит адрес минимального элемента.
        lodsw                   ; Пусть элемент, адрес
                                ; которого был в SI, минимальный,
        mov     di, si          ; DI - адрес элемента, сравниваемого
                                ; с минимальным.
        dec     dx              ; Надо проверить DX-1 элементов массива.
        mov     cx, dx
        jg      loop1           ; Переход на проверку, если DX > 1.
        ret
linear_selection _sort  endp
```

5.8. Перехват прерываний

В архитектуре процессоров 80x86 предусмотрены особые случаи, когда процессор прекращает (прерывает) выполнение текущей программы и немедленно передает управление программе-обработчику, специально написанной для обработки подобной ситуации. Такие особые ситуации делятся на два типа: прерывания и исключения, в зависимости от того, вызвало ли эту ситуацию какое-нибудь внешнее устройство или выполняемая процессором команда. Исключения делятся далее на три типа: ошибки, ловушки и остановы, в зависимости от того, когда по отношению к вызвавшей их команде они происходят. Ошибки появляются перед выполнением команды, поэтому обработчик такого исключения получит в качестве адреса возврата адрес ошибочной команды (начиная с процессоров 80286). Ловушки происходят сразу после выполнения команды, так что обработчик получает в качестве адреса возврата адрес следующей команды. И наконец, остановы могут возникать в любой момент и вообще не предусматривать средств возврата управления в программу.

Команда INT (а также INTO и INT3) используется в программах как раз для того, чтобы вызывать обработчики прерываний (или исключений). Фактически они являются исключениями ловушки, поскольку адрес возврата, который передается обработчику, указывает на следующую команду, но так как эти команды были введены до разделения особых ситуаций на прерывания и исключения, их практически всегда называют командами вызова прерываний. Ввиду того, что обработчики прерываний и исключений в DOS обычно не различают механизм вызова, с помощью команды INT можно передавать управление как на обработчики прерываний, так и исключений.

Как показано в главе 4, программные прерывания, то есть передача управления при помощи команды INT, являются основным средством вызова процедур DOS и BIOS, потому что в отличие от вызова через команду CALL здесь не нужно знать адреса вызываемой процедуры - достаточно только номера. С другой стороны интерфейса рассмотрим, как строится обработчик программного прерывания.

5.8.1. Обработчики прерываний

Когда в реальном режиме выполняется команда INT, управление передается по адресу, который считывается из специального массива, таблицы векторов прерываний, начинающегося в памяти по адресу 0000h:0000h. Каждый элемент такого массива представляет собой дальний адрес обработчика прерывания в формате сегмент:смещение или 4 нулевых байта, если обработчик не установлен. Команда INT помещает в стек регистр флагов и дальний адрес возврата, поэтому, чтобы завершить обработчик, надо выполнить команды popf и retf или одну команду iret, которая в реальном режиме полностью им аналогична.

```
; Пример обработчика программного прерывания.
int_handler     proc    far
        mov     ax,0
        iret
int_handler     endp
```

После того как обработчик написан, следующий шаг - привязка его к выбранному номеру прерывания. Это можно сделать, прямо записав его адрес в таблицу векторов прерываний, например так:

```
        push    0       ; Сегментный адрес таблицы векторов прерываний -
        pop     es      ; в ES.
        pushf           ; Поместить регистр флагов в стек.
        cli             ; Запретить прерывания (чтобы не произошло
                        ; аппаратного прерывания между следующими командами,
; обработчик которого теоретически может вызвать INT 87h в тот момент,
; когда смещение уже будет записано, а сегментный адрес еще нет,
; что приведет к передаче управления в неопределенную область памяти).
; Поместить дальний адрес обработчика int_handler
; в таблицу векторов прерываний, в элемент номер 87h
; (одно из неиспользуемых прерываний).
        mov     word ptr es:[87h*4], offset int_handler
        mov     word ptr es:[87h*4+2], seg int_handler
        popf            ; Восстановить исходное значение флага IF.
```

Теперь команда INT 87h будет вызывать наш обработчик, то есть приводить к записи 0 в регистр AX.

Перед завершением работы программа должна восстанавливать все старые обработчики прерываний, даже если это были неиспользуемые прерывания типа 87h - автор какой-нибудь другой программы мог подумать точно так же. Для этого надо перед предыдущим фрагментом кода сохранить адрес старого обработчика, так что полный набор действий для программы, перехватывающей прерывание 87h, будет выглядеть следующим образом:

```
        push    0
        pop     es
; Скопировать адрес предыдущего обработчика в переменную old_handler.
        mov     eax,dword ptr es:[87h*4]
        mov     dword ptr old_handler,eax
; Установить наш обработчик.
        pushf
        cli
        mov     word ptr es:[87h«4], offset int_handler
        mov     word ptr es:[87h*4+2], seg int_handler
        popf
; Тело программы.
        [...]
; Восстановить предыдущий обработчик.
        push    0
        pop     es
        pushf
        cli
        mov     eax,word ptr old_handler
        mov     word ptr es:[87h*4],eax
        popf
```

Хотя прямое изменение таблицы векторов прерываний и кажется достаточно удобным, все-таки это не лучший подход к установке обработчика прерывания, и пользоваться им следует только в исключительных случаях, например внутри обработчиков прерываний. Для обычных программ DOS предоставляет две системные функции: 25h и 35h - установить и считать адрес обработчика прерывания, которые и рекомендуются к использованию в обычных условиях:

```
; Скопировать адрес предыдущего обработчика в переменную old_handler.
        mov     ax,3587h                ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Функция DOS: считать
                                        ; адрес обработчика прерывания.
        mov     word ptr old_handler, bx ; Возвратить смещение в BX
        mov     word ptr old_handler+2,es ; и сегментный адрес в ES.
                                        ; Установить наш обработчик.
        mov     ax,2587h                ; AH = 25h, AL = номер прерывания.
        mov     dx,seg int_handler      ; Сегментный адрес -
        mov     ds,dx                   ; в DS,
        mov     dx,offset int_handler   ; смещение в DX.
        int     21h                     ; Функция DOS: установить
                                        ; обработчик в тело программы
                                        ; (не забывайте, что ES
                                        ; изменился после вызова функции 35h!).
        [...]
; Восстановить предыдущий обработчик
        lds     dx,old_handler  ; Сегментный адрес в DS и смещение в DX.
        mov     ax,2587h        ; AH = 25h, AL = номер прерывания.
        int     21h             ; Установить обработчик.
```

Обычно обработчики прерываний применяют с целью обработки прерывания от внешних устройств или с целью обслуживания запросов других программ. Эти возможности рассмотрены далее, а здесь приведен пример использования обычного обработчика прерывания (или, в данном случае, исключения ошибки) для того, чтобы быстро найти минимум и максимум в большом массиве данных.

```
; Процедура minmax.
; Находит минимальное и максимальное значения в массиве слов.
; Вход: DS:BX = адрес начала массива
;       CX = число элементов в массиве
; Выход:
;       AX = максимальный элемент
;       BX = минимальный элемент
minmax          proc    near
; Установить наш обработчик прерывания 5.
        push    0
        pop     es
        mov     eax,dword ptr es:[5*4]
        mov     dword ptr old_int5,eax
        mov     word ptr es:[5*4],offset int5_handler
        mov     word ptr es:[5*4]+2,cs
; Инициализировать минимум и максимум первым элементом массива.
        mov     ax,word ptr [bx]
        mov     word ptr lower_bound,ax
        mov     word ptr upper_bound,ax
; Обработать массив.
        mov     di,2                    ; Начать со второго элемента.
bcheck:
        mov     ax,word ptr [bx][di]    ; Считать элемент в AX.
        bound   ax,bounds               ; Команда BOUND вызывает
                                        ; исключение - ошибку 5,
                                        ; если AX не находится в пределах
                                        ; lower_bound/upper_bound.
        add     di,2                    ; Следующий элемент.
        loop    bcheck                  ; Цикл на все элементы.
; Восстановить предыдущий обработчик.
        mov     eax,dword ptr old_int5
        mov     dword ptr es:[5*4],eax
; Вернуть результаты.
        mov     ax,word ptr upper_bound
        mov     bx,word ptr lower_bound
        ret
bounds:
lower_bound     dw      ?
upper_bound     dw      ?
old_int5        dd      ?

; Обработчик INT 5 для процедуры minmax.
; Сравнить AX со значениями upper_bound и lower_bound и копировать
; AX в один из них. Обработчик не обрабатывает конфликт между
```

```
; исключением BOUND и программным прерыванием распечатки экрана INT 5.
; Нажатие клавиши PrtScr в момент работы процедуры minmax приведет
; к ошибке. Чтобы это исправить, можно, например, проверять байт,
; на который указывает адрес возврата, если это 0CDh
; (код команды INT), то обработчик был вызван как INT 5,
int5_handler    proc    far
        cmp     ax,word ptr lower_bound  ; Сравнить AX с нижней границей.
        jl      its_lower                ; Если не меньше -
                                         ; это было нарушение
        mov     word ptr upper_bound,ax  ; верхней границы.
        iret
its_lower:
        mov     word ptr lower_bound,ax  ; Иначе это было нарушение
        iret                             ; нижней границы.
int5_handler    endp
minmax          endp
```

Разумеется, вызов исключения при ошибке занимает много времени, но, если массив достаточно большой и неупорядоченный, значительная часть проверок будет происходить без ошибок и быстро.

При помощи собственных обработчиков исключений можно справиться и с другими особыми ситуациями, например обрабатывать деление на ноль и остальные исключения, которые возникают в программе. В реальном режиме есть вероятность столкнуться всего с шестью исключениями:

- #DE (деление на ноль) - INT 0 - ошибка, появляющаяся при переполнении и делении на ноль. Как для любой ошибки, адрес возврата указывает на ошибочную команду;
- #DB (прерывание трассировки) - INT 1 - ловушка, возникающая после выполнения каждой команды, если флаг TF установлен в 1. Используется отладчиками, действующими в реальном режиме;
- #OF (переполнение) — INT 4 - ловушка, возникающая после выполнения команды INTO, если флаг OF установлен;
- #BR (переполнение при BOUND) - INT 5 - уже рассмотренная нами ошибка, которая происходит при выполнении команды BOUND;
- #UD (недопустимая команда) - INT 6 - ошибка, возникающая при попытке выполнить команду, отсутствующую на данном процессоре;
- #NM (сопроцессор отсутствует) - INT 7 - ошибка, появляющаяся при попытке выполнить команду FPU, если FPU отсутствует.

5.8.2. Прерывания от внешних устройств

Прерывания от внешних устройств, или аппаратные прерывания, - это то, что понимается под термином «прерывание». Внешние устройства (клавиатура, дисковод, таймер, звуковая карта и т. д.) подают сигнал, по которому процессор прерывает выполнение программы и передает управление на обработчик прерывания. Всего на персональных компьютерах используется 15 аппаратных прерываний, хотя теоретически возможности архитектуры позволяют довести их число до 64.

Рассмотрим их кратко в порядке убывания приоритетов («прерывание имеет более высокий приоритет» означает, что, пока не завершился его обработчик, прерывания с низкими приоритетами будут ждать своей очереди):

- IRQ0 (INT 8) - прерывание системного таймера, вызывается 18,2 раза в секунду. Стандартный обработчик этого прерывания вызывает INT 1Ch при каждом вызове, так что, если программе необходимо только регулярно получать управление, а не перепрограммировать таймер, рекомендуется использовать прерывание 1Ch;
- IRQ1 (INT 9) - прерывание клавиатуры, вызывается при каждом нажатии и отпускании клавиши на клавиатуре. Стандартный обработчик этого прерывания выполняет довольно много функций, начиная с перезагрузки по **Ctrl-Alt-Del** и заканчивая помещением кода клавиши в буфер клавиатуры BIOS;
- IRQ2 - к этому входу на первом контроллере прерываний подключены аппаратные прерывания IRQ8 - IRQ15, но многие BIOS перенаправляют IRQ9 на INT 0Ah;
- IRQ8 (INT 70h) - прерывание часов реального времени, вызывается часами реального времени при срабатывании будильника и если они установлены на генерацию периодического прерывания (в последнем случае IRQ8 вызывается 1024 раза в секунду);
- IRQ9 (INT 0Ah или INT 71h) - прерывание обратного хода луча, вызывается некоторыми видеоадаптерами при обратном ходе луча. Часто используется дополнительными устройствами (например, звуковыми картами, SCSI-адаптерами и т. д.);
- IRQ10 (INT 72h) - используется дополнительными устройствами;
- IRQ11 (INT 73h) - используется дополнительными устройствами;
- IRQ12 (INT 74h) - мышь на системах PS, используется дополнительными устройствами;
- IRQ13 (INT 02h или INT 75h) - ошибка математического сопроцессора. По умолчанию это прерывание отключено как на FPU, так и на контроллере прерываний;
- IRQ14 (INT 76h) - прерывание первого IDE-контроллера «операция завершена»;
- IRQ15 (INT 77h) - прерывание второго IDE-контроллера «операция завершена»;
- IRQ3 (INT 0Bh) - прерывание последовательного порта COM2, вызывается, если порт COM2 получил данные;
- IRQ4 (INT 0Ch) - прерывание последовательного порта COM1, вызывается, если порт COM1 получил данные;
- IRQ5 (INT 0Dh) - прерывание LPT2, используется дополнительными устройствами;
- IRQ6 (INT 0Eh) - прерывание дисковода «операция завершена»;
- IRQ7 (INT OFh) - прерывание LPT1, используется дополнительными устройствами.

Самые полезные для программ аппаратные прерывания - прерывания системного таймера и клавиатуры. Так как стандартные обработчики этих прерываний выполняют множество функций, от которых зависит работа системы, их нельзя заменять полностью, как мы поступали *с* обработчиком INT 5. Необходимо вызвать предыдущий обработчик, передав ему управление следующим образом (если его адрес сохранен в переменной old_handler - см. примеры ранее):

```
pushf
call    old_handler
```

Данные команды выполняют действие, аналогичное команде INT (сохранить флаги в стеке и передать управление подобно команде call), поэтому, когда обработчик завершится командой IRET, управление вернется в нашу программу. Так удобно вызывать предыдущий обработчик в начале собственного. Другой способ - простая команда jmp:

```
jmp     cs:old_handler
```

приводит к тому, что по выполнении команды IRET старым обработчиком управление сразу же перейдет к прерванной программе. Этот способ применяют, если нужно, чтобы сначала отработал новый обработчик, а потом он передал управление старому.

На следующем примере посмотрим, как осуществляется перехват прерывания от таймера:

```
; timer.asm
; Демонстрация перехвата прерывания системного таймера: вывод текущего времени
; в левом углу экрана.

        .model  tiny
        .code
        .186                            ; Для pusha/popa и сдвигов.
        org     100h
start           proc    near
;  Сохранить адрес предыдущего обработчика прерывания 1Ch.
        mov  -  ax,351Ch                ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Функция DOS: определить адрес обработчика
        mov     word ptr old_int1Ch,bx  ; прерывания
        mov     word ptr old_int1Ch+2,es ; (возвращается в ES:BX).
; Установить наш обработчик.
        mov     ax,251Ch                ; AH = 25h, AL = номер прерывания.
        mov     dx,offset int1Ch_handler ; DS:DX - адрес обработчика.
        int     21h                     ; Установить обработчик прерывания 1Ch.

; Здесь размещается собственно программа, например вызов command.com.
        mov     ah,1
        int     21h                     ; Ожидание нажатия на любую клавишу.
; Конец программы.

; Восстановить предыдущий обработчик прерывания 1Ch.
        mov     ax,251Ch                ; AH = 25h, AL = номер прерывания.
        mov     dx,word ptr old_int1Ch+2
```

```
        mov     ds,dx
        mov     dx,word ptr cs:old_int1Ch       ; DS:DX - адрес обработчика.
        int     21h
        ret

old_int1Ch      dd      ?       ; Здесь хранится адрес предыдущего обработчика.
start_position  dw      0       ; Позиция на экране, в которую выводится
                                ; текущее время.
start           endp

; Обработчик для прерывания 1Ch.
; Выводит текущее время в позицию start_position на экране (только в текстовом режиме).

int1Ch_handler  proc    far
        pusha                   ; Обработчик аппаратного прерывания
        push    es              ; должен сохранять ВСЕ регистры.
        push    ds

        push    cs              ; На входе в обработчик известно только
        pop     ds              ; значение регистра CS.
        mov     ah,02h          ; Функция 02h прерывания 1Ah:.
        int     1Ah             ; Чтение времени из RTC.
        jc      exit_handler    ; Если часы заняты - в другой раз.

                                ; AL = час в BCD-формате.
        call    bcd2asc                         ; Преобразовать в ASCII.
        mov     byte ptr output_line[2],ah      ; Поместить их в
        mov     byte ptr output_line[4],al      ; строку output_line.

        mov     al,cl                           ; CL = минута в BCD-формате.
        call    bcd2asc
        mov     byte ptr output_line[10],ah
        mov     byte ptr output_line[12],al

        mov     al,dh                           ; DH = секунда в BCD-формате.
        call    bcd2asc
        mov     byte ptr output_line[16],ah
        mov     byte ptr output_line[18],al

        mov     cx,output_line_l                ; Число байтов в строке - в CX.
        push    0B800h
        pop     es                              ; Адрес в видеопамяти -
        mov     di,word ptr start_position      ; в ES:DI.
        mov     si,offset output_line           ; Адрес строки в DS:SI.
        cld
        rep     movsb                           ; Скопировать строку.
exit_handler:
        pop     ds                              ; Восстановить все регистры.
        pop     es
        popa
        jmp     cs:old_int1Ch   ; Передать управление предыдущему обработчику.

; Процедура bcd2asc.
; Преобразует старшую цифру упакованного BCD-числа из AL в ASCII-символ,
```

```
; который будет помещен в AH, а младшую цифру - в ASCII-символ в AL.
bcd2asc         proc    near
        mov     ah,al
        and     al,0Fh                          ; Оставить младшие 4 бита в AL.
        shr     ah,4                            ; Сдвинуть старшие 4 бита в AH.
        or      ax,3030h                        ; Преобразовать в ASCII-символы.
        ret
bcd2asc         endp

; Строка " 00h 00:00" с атрибутом 1Fh (белый на синем) после каждого символа.
output_line     db      ' ',1Fh,'0',1Fh,'0',1Fh,'h',1Fh
                db      ' ',1Fh,'0',1Fh,'0',1Fh,':',1Fh
                db      '0',1Fh,'0',1Fh,' ',1Fh
output_line_l   equ $-output_line

int1Ch_handler  endp

end             start
```

Если в этом примере вместо ожидания нажатия на клавишу поместить какую-нибудь программу, работающую в текстовом режиме, например tinyshell из раздела 4.10, она выполнится как обычно, но в правом верхнем углу будет постоянно показываться текущее время, то есть такая программа будет осуществлять два действия одновременно. Именно для этого и применяется механизм аппаратных прерываний - они позволяют процессору выполнять одну программу, в то время как отдельные программы следят за временем, считывают символы из клавиатуры и помещают их в буфер, получают и передают данные через последовательные и параллельные порты и даже обеспечивают многозадачность, переключая процессор между разными задачами по прерыванию системного таймера.

Разумеется, обработка прерываний не должна занимать много времени: если прерывание происходит достаточно часто (например, прерывание последовательного порта может происходить 28 800 раз в секунду), его обработчик обязательно должен выполняться за более короткое время. Если, например, обработчик прерывания таймера будет выполняться 1/32,4 секунды, то есть половину времени между прерываниями, вся система станет работать в два раза медленнее. А если еще одна программа с таким же долгим обработчиком перехватит это прерывание, система остановится совсем. Именно поэтому обработчики прерываний принято писать исключительно на ассемблере.

5.8.3. Повторная входимость

Пусть у нас есть собственный обработчик программного прерывания, который вызывают обработчики двух аппаратных прерываний, и пусть эти аппаратные прерывания произошли сразу одно за другим. В этом случае может получиться так, что второе аппаратное прерывание осуществится тогда, когда еще не закончится выполнение нашего программного обработчика. В большинстве случаев это не приведет ни к каким проблемам, но, если обработчик обращается к каким-либо переменным в памяти, могут произойти редкие, невоспроизводимые сбои в его работе. Например, пусть в обработчике есть некоторая переменная counter, используемая как счетчик, производящий подсчет от 0 до 99:

```
        mov     al,byte ptr counter      ; Считать счетчик в AL.
        cmp     al,100                   ; Проверить его на переполнение.
        jb      counter_ok               ; Если счетчик достиг 100,
; >>> здесь произошло второе прерывание <<<
        sub     al,100                   ; вычесть 100
        mov     byte ptr counter,al      ; и сохранить счетчик.
counter_ok:
```

Если значение счетчика было, например, 102, а второе прерывание произошло после проверки, но до вычитания 100, второй вызов обработчика получит то же значение 102 и уменьшит его на 100. Затем управление вернется, и следующая команда sub al,100 еще раз уменьшит AL на 100 и запишет полученное число - 98 на место. Если затем по значению счетчика вычисляется что-нибудь вроде адреса в памяти для записи, вполне возможно, что произойдет ошибка. О таком обработчике прерывания говорят, что он не является повторно входимым.

Чтобы защитить подобные критические участки кода, следует временно запретить прерывания, например так:

```
        cli                              ; Запретить прерывания.
        mov     al,byte ptr counter
        cmp     al,100
        jb      counter_ok
        sub     al,100
        mov     byte ptr counter,al
counter_ok:
        sti                              ; Разрешить прерывания.
```

Следует помнить, что, пока прерывания запрещены, система не отслеживает изменения часов, не получает данных с клавиатуры, поэтому прерывания надо обязательно, при первой возможности, разрешать. Всегда лучше пересмотреть используемый алгоритм и, например, хранить локальные переменные в стеке или применить специально разработанную команду CMPXCHG, которая позволяет одновременно провести сравнение и запись в глобальную переменную.

К сожалению, в MS DOS самый важный обработчик прерываний в системе - обработчик INT 21h - не является повторно входимым. В отличие от прерываний BIOS, обработчики которых используют стек прерванной программы, обработчик системных функций DOS записывает в SS:SP адрес дна одного из трех внутренних стеков DOS. Если функция была прервана аппаратным прерыванием, обработчик которого вызвал другую функцию DOS, она будет пользоваться тем же стеком, затирая все, что туда поместила прерванная функция. Когда управление вернется в прерванную функцию, в стеке окажется мусор и произойдет ошибка. Лучший выход - вообще не использовать прерывания DOS из обработчиков аппаратных прерываний, но если это действительно нужно, то принять необходимые меры предосторожности. Если прерывание произошло в тот Момент, когда не выполнялось никаких системных функций DOS, ими можно безбоязненно пользоваться. Чтобы определить, занята DOS или нет, надо сначала, до установки собственных обработчиков, выяснить адрес флага занятости DOS.

Функция DOS 34k. Определить адрес флага занятости DOS
Вход: AH = 34h
Выход: ES:BX = адрес однобайтного флага занятости DOS
ES:BX - 1 = адрес однобайтного флага критической ошибки DOS

Теперь обработчик прерывания может проверять состояние этих флагов и, если оба флага равны нулю, разрешается свободно пользоваться функциями DOS.

Если флаг критической ошибки не ноль, никакими функциями DOS пользоваться нельзя. Если флаг занятости DOS не ноль, можно пользоваться только функциями 01h - 0Ch, а чтобы воспользоваться какой-нибудь другой функцией, придется отложить действия до тех пор, пока DOS не освободится. Чтобы это выполнить, следует сохранить номер функции и параметры в каких-нибудь переменных в памяти и установить обработчик прерывания 8h или 1Ch. Этот обработчик будет при каждом вызове проверять флаги занятости и, если DOS освободилась, вызовет функцию с номером и параметрами, оставленными в переменных в памяти. Кроме того, участок программы после проверки флага занятости - критический, и прерывания должны быть запрещены. Не все функции DOS возвращаются быстро - функция чтения символа с клавиатуры может оставаться в таком состоянии минуты, часы или даже дни, пока пользователь не вернется и не нажмет на какую-нибудь клавишу, и все это время флаг занятости DOS будет установлен в 1. В DOS предусмотрена и такая ситуация. Все функции ввода символов с ожиданием вызывают INT 28h в том же цикле, в котором они опрашивают клавиатуру, так что, если установить обработчик прерывания 28h, из него можно вызывать все функции DOS, кроме 01 - 0Ch.

Пример вызова DOS из обработчика прерывания от внешнего устройства рассмотрен чуть ниже, в резидентных программах. А сейчас следует заметить, что функции BIOS, одну из которых мы вызывали в нашем примере timer.asm, также часто оказываются не повторно входимыми. В частности, этим отличаются обработчики программных прерываний 5, 8, 9, 0Bh, 0Ch, ODh, OEh, 10h, 13h, 14h, 16h, 17h. Поскольку BIOS не предоставляет какого-либо флага занятости, придется создать его самим:

```
int10_handler   proc    far
        inc     cs:byte ptr int10_busy    ; Увеличить флаг занятости.
        pushf                             ; Передать управление старому
                                          ; обработчику INT 10h,
        call    cs:dword ptr old_int10    ; эмулируя команду INT.
        dec     cs:byte ptr int10_busy    ; Уменьшить флаг занятости.
        iret
int10_busy      db      0
int10_handler   endp
```

Теперь обработчики аппаратных прерываний могут пользоваться командой INT 10h, если флаг занятости int10_busy равен нулю, и это не приведет к ошибкам, если не найдется чужой обработчик прерывания, который тоже станет обращаться к INT 10h и не будет ничего знать о нашем флаге занятости.

5.9. Резидентные программы

Программы, остающиеся в памяти после того, как управление возвращается в DOS, называются *резидентными*. Превратить программу в резидентную просто - достаточно вызвать специальную системную функцию DOS.

Функция DOS 31h: Оставить программу резидентной
Вход: AH = 31h
AL = код возврата
DX = размер резидента в 16-байтных параграфах (больше 06h), считая от начала PSP

Кроме того, существует и иногда используется предыдущая версия этой функции - прерывание 27h:

INT 27h: Оставить программу резидентной
Вход: AH = 27h
DX = адрес последнего байта программы (считая от начала PSP) + 1

Эта функция не может оставлять резидентными программы размером больше 64 Кб, но многие программы, написанные на ассемблере, соответствуют этому условию. Так как резидентные программы уменьшают объем основной памяти, их всегда пишут на ассемблере и оптимизируют для достижения минимального размера.

Никогда не известно, по каким адресам в памяти оказываются загруженные в разное время резидентные программы, поэтому единственным несложным способом получения управления является механизм программных и аппаратных прерываний. Принято разделять резидентные программы на активные и пассивные, в зависимости от того, перехватывают ли они прерывания от внешних устройств или получают управление, только если программа специально вызовет Команду INT с нужным номером прерывания и параметрами.

5.9.1. Пассивная резидентная программа

В качестве первой резидентной программы рассмотрим именно пассивный резидент, который будет активизироваться при попытке программ вызывать INT 21h и запрещать удаление файлов с указанного диска.

```
; tsr.asm
; Пример пассивной резидентной программы.
; Запрещает удаление файлов на диске, указанном в командной строке, всем
; программам, использующим средства DOS.
        .model  tiny
        .code
        org     2Ch
envseg          dw      ?       ; Сегментный адрес копии окружения DOS.
        org     80h
cmd_len         db      ?       ; Длина командной строки.
cmd_line        db      ?       ; Начало командной строки.
        org     100h    .       ; COM-программа.
```

```
start:
old_int21h:
        jmp     short  initialize       ; Эта команда занимает 2 байта, так что
                dw      0                       ; вместе с ними получим
                                        ; old_int21h dd ?.
int21h_handler  proc    far             ; Обработчик прерывания 21h.
        pushf '                         ; Сохранить флаги.
        cmp     ah,41h                  ; Если вызвали функцию 41h (удалить
        je      fn41h                   ; файл)
        cmp     ax,7141h                ; или 7141h (удалить файл с длинным именем),
        je      fn41h   ,               ; начать наш обработчик.
        jmp     short not_fn41h         ; Иначе - передать управление
                                        ; предыдущему обработчику.
fn41h:
        push    ax                      ; Сохранить модифицируемые
        push    bx                      ; регистры.
        mov     bx,dx
        cmp     byte ptr ds:[bx+1], ' :'  ; Если второй символ ASCIZ-строки,
                                        ; переданной INT 21h, двоеточие - первый
                                        ; символ должен быть именем диска.
        je      full_spec
        mov     ah,19h                  ; Иначе -
        int     21h                     ; функция DOS 19h - определить текущий диск.
        add     al,'A'                  ; Преобразовать номер диска
                                        ; к заглавной букве.
        jmp     short  compare          ; Перейти к сравнению.
full_spec:
        mov     al,byte ptr [bx]        ; AL = имя диска из ASCIZ-строки.
        and     al,11011111b            ; Преобразовать к заглавной букве.
compare:
        cmp     al,byte ptr cs:cmd_line[1] ; Если диски
        je      access_denied           ; совпадают - запретить доступ.
        pop     bx                      ; Иначе - восстановить
        pop     ax                      ; регистры
not_fn41h:
        popf                            ; и флаги
        jmp     dword ptr cs:old_int21h ; и передать управление
                                        ; предыдущему обработчику INT 21h.
access_denied:
        pop     bx                      ; Восстановить регистры.
        pop     ax
        popf
        push    bp
        mov     bp,sp
        or      word ptr [bp+6],1       ; Установить флаг переноса
                                        ; (бит 0) в регистре флагов,
                                        ; который поместила команда INT в стек
                                        ; перед адресом возврата.
        pop     bp
```

```
        mov     ax,5                    ; Возвратить код ошибки "доступ запрещен".
        iret                            ; Вернуться в программу.
int21h_handler  endp

initialize      proc near
        cmp     byte ptr cmd_len,3      ; Проверить размер командной строки
        jne     not_install             ; (должно быть 3 - пробел, диск, двоеточие).
        cmp     byte ptr cmd_line[2],':' ; Проверить третий символ
        jne     not_install             ; командной строки (должно быть двоеточие).
        mov     al,byte ptr cmd_line[1]
        and     al,11011111b            ; Преобразовать второй
                                        ; символ к заглавной букве.
        cmp     al,'A'                  ; Проверить, что это не
        jb      not_install             ; меньше "A" и не больше
        cmp     al,'Z'                  ; "Z".
        ja      not_install             ; Если хоть одно из этих условий
                                        ; не выполняется - выдать информацию
                                        ; о программе и выйти.
                                        ; Иначе - начать процедуру инициализации.

        mov     ax,3521h                ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Получить адрес обработчика INT 21h
        mov     word ptr old_int21h,bx  ; и поместить его в old_int21h.
        mov     word ptr old_int21h+2,es

        mov     ax,2521h                ; AH = 25h, AL = номер прерывания.
        mov     dx,offset int21h_handler ; DS:DX - адрес нашего обработчика.
        int     21h                     ; Установить обработчик INT 21h.

        mov     ah,49h                  ; AH = 49h.
        mov     es,word ptr envseg      ; ES = сегментный адрес блока с нашей
                                        ; копией окружения DOS.
        int     21h                     ; Освободить память из-под окружения.

        mov     dx,offset initialize    ; DX - адрес первого байта за концом
                                        ; резидентной части программы.
        int     27h                     ; Завершить выполнение, оставшись
                                        ; резидентом.

not_install:
        mov     ah,9                    ; AH = 09h
        mov     dx,offset usage         ; DS:DX = адрес строки с информацией об
                                        ; использовании программы.
        int     21h                     ; Вывод строки на экран.
        ret                             ; Нормальное завершение программы.

; Текст, который выдает программа при запуске с неправильной командной строкой:
usage           db      "Использование: tsr.com D:",0Dh,0Ah
                db      "Запрещает удаление на диске D:",0Dh,0Ah
                db      "$"

initialize      endp
        end     start
```

Если запустить эту программу с командной строкой D:, никакой файл на диске D нельзя будет удалить командой **Del,** средствами оболочек типа Norton Commander и большинством программ для DOS. Действие этого запрета, однако, не будет распространяться на оболочку Far, которая использует системные функции Windows API, и на программы типа Disk Editor, обращающиеся с дисками при помощи функций BIOS (INT 13h). Несмотря на то что мы освободили память, занимаемую окружением DOS (а это могло быть лишних 512 или даже 1024 байта), наша программа все равно занимает в памяти 352 байта потому, что первые 256 байт отводятся для блока PSP. Существует возможность оставить программу резидентной без PSP - для этого инсталляционная часть программы должна скопировать резидентную часть с помощью, например, movs в начало PSP. Но при этом возникает сразу несколько проблем: во-первых, команда INT 27h, так же как и функция DOS 31h, использует данные из PSP для своей работы; во-вторых, код резидентной части должен быть написан для работы с нулевого смещения, а не со 100h, как обычно; и, в-третьих, некоторые программы, исследующие выделенные блоки памяти, определяют конец блока по адресу, находящемуся в PSP программы — владельца блока со смещением 2. С первой проблемой можно справиться вручную, создав отдельные блоки памяти для резидентной и инсталляционной частей программы, новый PSP для инсталляционной части и завершив программу обычной функцией 4Ch или INT 20h. Реальные программы, делающие это, существуют (например, программа поддержки нестандартных форматов дискет PU_1700), но мы не будем чрезмерно усложнять наш первый пример и скопируем резидентную часть не в позицию 0, а в позицию 80h, то есть, начиная с середины PSP, оставив в нем все значения, необходимые для нормальной работы функций DOS.

Прежде чем это сделать, заметим, что и номер диска, и адрес предыдущего обработчика INT 21h изменяются только при установке резидента и являются константами во время всей его работы. Более того, каждое из этих чисел используется только по одному разу. В таких условиях оказывается, что можно вписать номер диска и адрес перехода на старый обработчик прямо в код программы. Кроме того, после этого наш резидент не будет больше ссылаться ни на какие переменные с конкретными адресами, а значит, его код становится *перемещаемым,* то есть его можно выполнять, скопировав в любую область памяти.

```
; tsrpsp.asm
; Пример пассивной резидентной программы с переносом кода в PSP.
; Запрещает удаление файлов на диске, указанном в командной строке,
; всем программам, использующим средства DOS.

                .model  tiny
                .code
                org     2Ch
envseg          dw      7               ; Сегментный адрес копии окружения DOS.
                org     80h
cmd_len         db      ?               ; Длина командной строки.
cmd_line        db      7               ; Начало командной строки.
                org     100h            ; COM-программа.
```

```
start:
old_int21h:
        jmp'    short  initialize ; Переход на инициализирующую часть.

int21h_handler  proc     far      ; Обработчик прерывания 21h.
        pushf                     ; Сохранить флаги.
        cmp     ah,41h            ; Если вызвали функцию 41h
                                  ; (удалить файл)
        je      fn41h
        cmp     ax,7141h          ; или 7141h (удалить файл
                                  ; с длинным именем),
        je      fn41h             ; начать наш обработчик.
        .jmp    short not_fn41h   ; Иначе - передать
                                  ; управление предыдущему обработчику.
fn41h:
        push    ax                ; Сохранить модифицируемые
        push    bx                ; регистры.
        mov     bx,dx             ; Можно было бы использовать
                                  ; адресацию [edx+1], но в старшем
                                  ; слове EDX совсем необязательно 0.
        cmp     byte ptr [bx+1],':' ; Если второй символ ASCIZ-строки,
                                  ; переданной INT 21h, двоеточие, первый
                                  ; символ должен быть именем диска.
        je      full_spec
        mov     ah,19h            ; Иначе:
        int     21h               ; функция DOS 19h - определить
                                  ; текущий диск.
        add     al,'A'            ; Преобразовать номер диска
                                  ; к заглавной букве.
        jmp     short compare     ; Перейти к сравнению.
full_spec:
        mov     al,byte ptr [bx]; AL = имя диска из ASCIZ-строки.
        and     al,11011111b      ; Преобразовать к заглавной букве.
compare:
                db      3Ch       ; Начало кода команды CMP AL, число.
drive_letter:   db      'Z'       ; Сюда процедура инициализации
                                  ; впишет нужную букву.
        pop     bx                ; Эти регистры больше не
        pop     ax                ; понадобятся. Если диски совпадают -
        je      access_denied     ; запретить доступ.
not_fn41h:
        popf                      ; Восстановить флаги и передать
                                  ; управление предыдущему
                                  ; обработчику INT 21h:.
                db      0EAh      ; Начало кода команды
                                  ; JHP, число FAR.
old_int21h      dd      0         ; Сюда процедура инициализации
                                  ; запишет адрес предыдущего
                                  ; обработчика INT 21h. '
```

```
access_denied:
        popf
        push    bp
        mov     bp,sp                   ; Чтобы адресоваться в стек
                                        ; в реальном режиме,
        or      word ptr [bp+6],1       ; установить флаг
                                        ; переноса (бит 0) в регистре
                                        ; флагов, который поместила команда INT
                                        ; в стек перед адресом возврата.
        pop     bp
        mov     ax,5                    ; Возвратить код ошибки "доступ запрещен".
        iret                            ; Вернуться в программу.
int21h_handler  endp

tsr_length      equ     $-int21h_handler

initialize      proc    near
        cmp     byte ptr cmd_len,3      ; Проверить размер
                                        ; командной строки
        jne     not_install             ; (должно быть 3 -
                                        ; пробел, диск, двоеточие).
        cmp     byte ptr cmd_line[2],':' ; Проверить
                                        ; третий символ командной
        jne     not_install             ; строки (должно быть двоеточие).
        mov     al,byte ptr cmd_line[1]
        and     al,11011111b            ; Преобразовать второй
                                        ; символ к заглавной букве.
        cmp     al,'A'                  ; Проверить, что это не меньше "A"
        jb      not_install             ;  и не больше
        cmp     al,'Z'                  ; "Z".
        ja      not_install             ; Если хоть одно из этих условий
                                        ; не выполняется - выдать
                                        ; информацию о программе и выйти.
                                        ; Иначе - начать процедуру
                                        ; инициализации.
        mov     byte ptr drive_letter,al ; Вписать имя
                                        ; диска в код резидента.
        push    es
        mov     ax,3521h                ; AH = 35h,
                                        ; AL = номер прерывания.
        int     21h                     ; Получить адрес
                                        ; обработчика INT 21h
        mov     word ptr old_int21h,bx  ; и вписать его в код резидента.
        mov     word ptr old_int21h+2,es
        pop     es

        cld                             ; Перенос кода резидента,
        mov     si,offset int21h_handler ; начиная с этого адреса,
        mov     di,80h                  ; в PSP:0080h.
        rep     movsb
```

```
        mov     ax,2521h                ; AH = 25h,
                                        ; AL = номер прерывания.
        mov     dx,0080h                ; DS:DX - адрес нашего обработчика.
        int     21h                     ; Установить обработчик INT 21h.
        mov     ah,49h                  ; AH = 49h
        mov     es,word ptr envseg      ; ES = сегментный адрес блока
                                        ; с нашей копией окружения DOS.
        int     21h                     ; Освободить память из-под
                                        ; окружения.

        mov     dx,80h+tsr_length       ; DX - адрес первого байта
                                        ; за концом резидентной части
                                        ; программы.
        int     27h                     ; Завершить выполнение,
                                        ; оставшись резидентом.

not_install:
        mov     ah,9                    ; AH = 09h.
        mov     dx,offset usage         ; DS:DX = адрес строки
                                        ; с информацией об
                                        ; использовании программы.
        int     21h                     ; Вывод строки на экран.
        ret                             ; Нормальное завершение
                                        ; программы.

; Текст, который выдает программа при запуске
; с неправильной командной строкой:
usage           db      "Usage: tsr.com D:",0Dh,0Ah
                db      "Denies delete on drive D:",0Dh,0Ah
                db      "$"

initialize      endp
                end     start
```

Теперь эта резидентная программа занимает в памяти только 208 байт.

5.9.2. Мультиплексорное прерывание

Если вы запустите предыдущий пример несколько раз, с разными или даже одинаковыми именами дисков в командной строке, объем свободной памяти DOS всякий раз будет уменьшаться на 208 байт, то есть каждый новый запуск устанавливает дополнительную копию резидента, даже если она идентична уже установленной. Разумеется, это неправильно - инсталляционная часть обязательно должна уметь определять, загружен ли уже резидент в памяти перед его установкой. В нашем случае это не приводит ни к каким последствиям, кроме незначительного уменьшения объема свободной памяти, но во многих чуть более сложных случаях могут возникать различные проблемы, например многократное срабатывание активного резидента по каждому аппаратному прерыванию, которое он перехватывает.

Для того чтобы идентифицировать себя в памяти, резидентные программы обычно или устанавливали обработчики для неиспользуемых прерываний, или

вводили дополнительную функцию в используемое прерывание. Например: наш резидент мог бы проверять в обработчике INT 21h AH на равенство какому-нибудь числу, не соответствующему функции DOS, и возвращать в, например, AL код, означающий, что резидент присутствует. Очевидная проблема, связанная с таким подходом, - вероятность того, что кто-то другой выберет то же неиспользуемое прерывание или что будущая версия DOS станет использовать ту же функцию. Именно для решения этой проблемы, начиная с версии DOS 3.3, был предусмотрен специальный механизм, позволяющий разместить до 64 резидентных программ в памяти одновременно, - *мультиплексорное прерывание.*

INT 2Fh: Мультиплексорное прерывание
Вход: AH = идентификатор программы
00h - 7Fh зарезервировано для DOS/Windows
OB8h - 0BFh зарезервировано для сетевых функций
0C0h - 0FFh отводится для программ
AL = код функции
OOh - проверка наличия программы
остальные функции - свои для каждой программы
BX, CX, DX = 0 (так как некоторые программы выполняют те или иные действия в зависимости от значений этих регистров)
Выход:

Для подфункции AL = OOh, если установлен резидент с номером AH, он должен вернуть OFFh в AL и какой-либо идентифицирующий код в других регистрах, например адрес строки с названием и номером версии. Оказалось, что такого уровня спецификации совершенно недостаточно и резидентные программы попрежнему работали по-разному, находя немало способов конфликтовать между собой. Поэтому появилась новая спецификация - AMIS (альтернативная спецификация мультиплексорного прерывания). Все резидентные программы, следующие этой спецификации, обязаны поддерживать базовый набор функций AMIS, а их обработчики прерываний должны быть написаны в соответствии со стандартом IBM ISP, который делает возможным выгрузку резидентных программ из памяти в любом порядке.

Начало обработчика прерывания должно выглядеть следующим образом:

+00h: 2 байта - 0EBh,10h (команда jmp short на первый байт после этого блока)
+02h: 4 байта — адрес предыдущего обработчика: именно по адресу, хранящемуся здесь, обработчик должен выполнять call или jmp
+06h: 2 байта - 424Bh - сигнатура ISP-блока
+08h: байт - 80h, если это первичный обработчик аппаратного прерывания (то есть он посылает контроллеру прерываний сигнал EOI)
OOh, если это обработчик программного или дополнительный обработчик аппаратного прерывания
+09h: 2 байта - команда jmp short на начало подпрограммы аппаратного сброса - обычно состоит из одной команды IRET
+0Bh: 7 байт - зарезервировано

Все стандартное общение с резидентной программой по спецификации AMIS происходит через прерывание 2Dh. При установке инсталляционная часть резидентной программы должна проверить, нет ли ее копии, просканировав все идентификаторы от 00 до 0FFh, и, если нет, установить обработчик на первый свободный идентификатор.

INT 2Dh: Мультиплексорное прерывание AMIS
Вход: AH = идентификатор программы
AL = 00: проверка наличия
AL = 01: получить адрес точки входа
AL = 02: деинсталляция
AL = 03: запрос на активизацию (для «всплывающих» программ)
AL = 04: получить список перехваченных прерываний
AL = 05: получить список перехваченных клавиш
AL = 06: получить информацию о драйвере (для драйверов устройств)
AL = 07 - 0Fh - зарезервировано для AMIS
AL = 1Fh - OFFh - свои для каждой программы
Выход: AL = 00h, если функция не поддерживается

Рассмотрим функции, описанные в спецификации AMIS как обязательные.

INT 2Dh AL = 00h: Функция AMIS - проверка наличия резидентной программы
Вход: AH = идентификатор программы
AL - 00h
Выход: AL = OOh, если идентификатор не занят
AL = OFFh, если идентификатор занят
CH = старший номер версии программы
CL = младший номер версии программы
DX:DI = адрес AMIS-сигнатуры, по первым 16 байтам которой и происходит идентификация.
Первые 8 байт - имя производителя программы; следующие 8 байт - имя программы; затем или 0 или ASCIZ-строка С описанием программы, не больше 64 байт.

INT 2Dh AL = 02h: Функция AMIS - выгрузка резидентной программы из памяти
Вход: AH = идентификатор программы
AL = 02h
DX:BX = адрес, на который нужно передать управление после выгрузки
Выход:
AL = 01h - выгрузка не удалась
AL = 02h - выгрузка сейчас невозможна, но произойдет чуть позже
AL = 03h - резидент не умеет выгружаться сам, но его можно выгрузить, резидент все еще активен
BX = сегментный адрес резидента
AL = 04h - резидент не умеет выгружаться сам, но его можно выгрузить, резидент больше неактивен

BX = сегментный адрес резидента
AL = 05h - сейчас выгружаться небезопасно - повторить запрос позже
AL = 06h - резидент был загружен из CONFIG.SYS и выгрузиться не-
может, резидент больше неактивен
AL = 07h - это драйвер устройства, который не умеет выгружаться сам
BX = сегментный адрес
AL = OFFh с передачей управления на DX:BX - успешная выгрузка

INT 2DhAL = 02h: Функция AMIS - запрос на активизацию
Вход: AH = идентификатор программы
AL = 03c
Выход: AL = 00h - резидент - «невсплывающая» программа
AL = 01h - сейчас «всплывать» нельзя - повторить запрос позже
AL = 02h - сейчас «всплыть» не могу, но «всплыву» при первой возмож-
ности
AL = 03h - уже «всплыл»
AL = 04h - «всплыть» невозможно
BX, CX — коды ошибки
AL = OFFh - программа «всплыла», отработала и завершилась
BX - код завершения

INT 2DhAL = 04h: Функция AMIS - получить список перехваченных прерываний
Вход: AH = идентификатор программы
AL - 04h
Выход: AL = 04h
DX:BX = адрес списка прерываний, состоящего из 3-байтных структур:
байт 1: номер прерывания (2Dh должен быть последним)
байты 2, 3: смещение относительно сегмента, возвращенного
в DX обработчика прерывания (по этому смещению должен
Находиться стандартный заголовок ISP)

INT 2DhAL = 05h: Функция AMIS - получить список перехваченных клавиш
Вход: AH = идентификатор программы
AL = 05h
Выход:
AL = OFFh - функция поддерживается
DX:BX = адрес списка клавиш:
+00h: 1 байт: тип проверки клавиши:
бит 0: проверка до обработчика INT 9
бит 1: проверка после обработчика INT 9
бит 2: проверка до обработчика INT 15h/AH = 4Fh
бит 3: проверка после обработчика INT 15h/AH = 4Fh
бит 4: проверка при вызове INT 16h/AH = O, 1, 2
бит 5: проверка при вызове INT 16h/AH = 10h, 11h, 12h
бит 6: проверка при вызове INT 16h/AH = 20h, 21h, 22h
бит 7: O

+01h: 1 байт: количество перехваченных клавиш
+02h: массив структур по 6 байт:
байт 1: скан-код клавиши (старший бит - отпускание клавиши, 00/80h - если срабатывание только по состоянию Shift-Ctrl-Alt и т, д.)
байты 2, 3: необходимое состояние клавиатуры (формат тот же, что и в слове состояния клавиатуры, только бит 7 соответствует нажатию любой клавиши **Shift)**
байты 4, 5: запрещенное состояние клавиатуры (формат тот же)
байт 6: способ обработки клавиши
бит 0: клавиша перехватывается до обработчиков
бит 1: клавиша перехватывается после обработчиков
бит 2: другие обработчики не должны «проглатывать» клавишу
бит 3: клавиша не сработает, если, пока она была нажата, нажимали или отпускали другие клавиши
бит 4: клавиша преобразовывается в другую
бит 5: клавиша иногда «проглатывается», а иногда передается дальше
биты 6, 7: О

Теперь можно написать резидентную программу, и она не загрузится дважды в память. В этой программе установим дополнительный обработчик на аппаратное прерывание от клавиатуры IRQ1 (INT 9) для отслеживания комбинации клавиш Alt-A; после их нажатия программа перейдет в активное состояние, выведет на экран свое окно и среагирует уже на большее количество клавиш. Такие программы, активизирующиеся при нажатии какой-либо клавиши, часто называют «всплывающими» программами, но наша программа на самом деле будет только казаться «всплывающей». Настоящая «всплывающая» программа после активизации в обработчике INT 9h не возвращает управление до окончания работы пользователя. В нашем случае управление возобновится после каждого нажатия клавиши, хотя сами клавиши будут поглощаться программой, так что ей можно пользоваться одновременно с работающими программами, причем на скорости их работы активный ascii.com никак не скажется.

Так же как и с предыдущим примером, программы, не использующие средства DOS/BIOS для работы с клавиатурой, например файловый менеджер FAR, будут получать все нажатые клавиши параллельно с нашей программой, что приведет к нежелательным эффектам на экране. Кроме того, в этом упрощённом примере отсутствуют некоторые необходимые проверки (например, текущий видеорежим) и функции (например, выгрузка программы из памяти), но тем не менее это реально используемая программа. С ее помощью легко посмотреть, какой символ соответствует какому ASCII-коду, и ввести любой символ, которого нет на клавиатуре, в частности псевдографику.

```
; ascii.asm
; Резидентная программа для просмотра и ввода ASCII-символов.
; HCI:
```

```
;       Alt-A - активизация программы.
;       Клавиши управления курсором - выбор символа.
;       Enter - выход из программы с вводом символа.
;       Esc - выход из программы без ввода символа.
; API:
;       Программа занимает первую свободную функцию прерывания 2Dh
;       в соответствии со спецификацией AMIS 3.6.
;       Поддерживаются функции AMIS 00h, 02h, 03h, 04h и 05h.
;       Обработчики прерываний построены в соответствии с IMB ISP.

; Адрес верхнего левого угла окна (23-я позиция в третьей строке).
START_POSITION  equ (80*2+23)*2

        .model  tiny
        .code
        .186                            ; Для сдвигов и команд pusha/popa.
        org     2Ch
envseg          dw      ?               ; Сегментный адрес окружения DOS.
        org     100h                    ; Начало COM-программы.

start:
        jmp     initialize              ; Переход на инициализирующую часть.

hw_reset9:
        retf                            ; ISP: минимальный hw_reset.

; Обработчик прерывания 09h (IRQ1)
;
int09h_handler  proc    far
        jmp     short actual_int09h_handler     ; ISP: пропустить блок.
old_int09h      dd      7               ; ISP: старый обработчик.
                dw      424Bh           ; ISP: сигнатура.
                db      00h             ; ISP: вторичный обработчик.
        jmp     short hw_reset9         ; ISP: ближний jmp на hw_reset.
                db      7 dup (0)       ; ISP: зарезервировано.
actual_int09h_handler:                  ; Начало обработчика INT 09h.

; Сначала вызовем предыдущий обработчик, чтобы дать BIOS возможность
; обработать прерывание и, если это было нажатие клавиши, поместить код
; в клавиатурный буфер, так как мы пока не умеем работать с портами клавиатуры
; и контроллера прерываний.
        pushf
        call    dword ptr cs:old_int09h

; По этому адресу обработчик INT 2Dh запишет код команды IRET
; для дезактивизации программы.
disable_point   label byte

        pusha                           ; Это аппаратное прерывание - надо
        push    ds                      ; сохранить все регистры.
        push    es
        cld                             ; Флаг для команд строковой обработки.
```

```
        push    0B800h
        pop     es                      ; ES = сегментный адрес видеопамяти.
        push    0040h
        pop     ds                      ; DS = сегментный адрес области данных BIOS.
        mov     di,word ptr ds:001Ah    ; Адрес головы буфера клавиатуры.
        cmp     di,word ptr ds:001Ch    ; Если он равен адресу хвоста,
        je      exit_09h_handler        ; буфер пуст и нам делать нечего
                                        ; (например если прерывание пришло по
                                        ; отпусканию клавиши).

        mov     ax,word ptr [di]        ; Иначе: считать символ из головы буфера.

        cmp     byte ptr cs:we_are_active,0     ; Если программа уже
        jne     already_active                  ; активизирована - перейти
                                                ; к обработке стрелок и т.п.

        cmp     ah,1Eh                  ; Если прочитанная клавиша не A
        jne     exit_09h_handler        ; (скан-код 1Eh) - выйти.

        mov     al,byte ptr ds:0017h    ; Иначе: считать байт
                                        ; состояния клавиатуры.
        test    al,08h                  ; Если не нажата любая Alt,
        jz      exit_09h_handler        ; выйти.

        mov     word ptr ds:001Ch,di    ; Иначе: установить адреса
                                        ; головы и хвоста буфера одинаковыми,
                                        ; пометив его тем самым как пустой.

        call    save_screen             ; Сохранить область экрана, которую
                                        ; накроет всплывающее окно.
        push    cs
        pop     ds                      ; DS = наш сегментный адрес.
        call    display_all             ; Вывести на экран окно программы.

        mov     byte ptr we_are_active,1 ; Установить флаг
        jmp     short exit_09h_handler   ; и выйти из обработчика.

; Сюда передается управление, если программа уже активизирована.
; При этом ES = 0B800h, DS = 0040h, DI = адрес головы буфера клавиатуры,
; AX = символ из головы буфера.
already_active:
        mov     word ptr ds:001Ch,di    ; Установить адреса
                                        ; головы и хвоста буфера одинаковыми,
                                        ; пометив его тем самым как пустой.
        push    cs
        pop     ds                      ; DS = наш сегментный адрес.

        mov     al,ah           ; Команды cmp al, ? короче команд cmp ah, ?.
        mov     bh,byte ptr current_char ; Номер выделенного в данный момент
                                         ; ASCII-символа.
        cmp     al,48h                  ; Если нажата стрелка вверх (скан-код 48h),
        jne     not_up
        sub     bh,16                   ; уменьшить номер символа на 16.
```

```
not_up:
        cmp     al,50h                  ; Если нажата стрелка вниз (скан-код 50h),
        jne     not_down
        add     bh,16                   ; увеличить номер символа на 16.
not_down:
        cmp     al,4Bh                  ; Если нажата стрелка влево,
        jne     not_left
        dec     bh                      ; уменьшить номер символа на 1.
not_left:
        cmp     al,4Dh                  ; Если нажата стрелка вправо,
        jne     not_right
        inc     bh                      ; увеличить номер символа на 1.
not_right:
        cmp     al,1Ch                  ; Если нажата Enter (скан-код 1Ch),
        je      enter_pressed           ; перейти к его обработчику.
        dec     al                      ; Если не нажата клавиша Esc (скан-код 1),
        jnz     exit_with_display       ; выйти из обработчика, оставив
                                        ; окно нашей программы на экране.
exit_after_enter:                       ; Иначе:
        call    restore_screen          ; убрать наше окно с экрана,
        mov     byte ptr we_are_active,0 ; обнулить флаг активности,
        jmp     short exit_09h_handler  ; выйти из обработчика.

exit_with_display:                      ; Выход с сохранением окна
                                        ; (после нажатия стрелок).
        mov     byte ptr current_char,bh ; Записать новое значение
                                        ; текущего символа.
        call    display_all             ; Перерисовать окно.

exit_09h_handler:                       ; Выход из обработчика INT 09h.
        pop     es
        pop     ds                      ; Восстановить регистры
        popa
        iret                            ; и вернуться в прерванную программу.

we_are_active   db      0               ; Флаг активности: равен 1, если
                                        ; программа активна.
current_char    db      37h             ; Номер ASCII-символа, выделенного
                                        ; в данный момент.

; Сюда передается управление, если в активном состоянии была нажата Enter.
enter_pressed:
        mov     ah,05h                  ; Функция 05h
        mov     ch,0                    ; CH = 0
        mov     cl,byte ptr current_char ; CL = ASCII-код
        int     16h                     ; Поместить символ в буфер клавиатуры.
        jmp     short exit_after_enter  ; Выйти из обработчика, стерев окно.

; Процедура save_screen.
; Сохраняет в буфере screen_buffer содержимое области экрана, которую
; закроет наше окно.
```

```
save_screen     proc    near
        mov     si,START_POSITION
        push    0B800h                  ; DS:SI - начало этой области
                                        ; в видеопамяти.
        pop     -ds
        push    es
        push    cs
        pop     es
        mov     di,offsetscreen_buffer  ; ES:DI - начало буфера в программе.
        mov     dx,18                   ; DX = счетчик строк.
save_screen_loop:
        mov     cx,33                   ; CX = счетчик символов в строке.
        rep     movsw                   ; Скопировать строку с экрана в буфер.
        add     si,(80-33)*2            ; Увеличить SI до начала следующей строки.
        dec     dx                      ; Уменьшить счетчик строк.
        jnz     save_screen_loop        ; Если он не ноль - продолжить цикл.
        pop     es
        ret
save_screen     endp

; Процедура restore_screen.
; Восстанавливает содержимое области экрана, которую закрывало наше
; всплывающее окно данными из буфера screen_buffer.

restore_screen  proc    near
        mov     di,START_POSITION       ; ES:DI - начало области в видеопамяти.
        mov     si,offset screen_buffer ; DS:SI - начало буфера.
        mov     dx,18                   ; Счетчик строк.
restore_screen_loop:
        mov     cx,33                   ; Счетчик символов в строке.
        rep     movsw                   ; Скопировать строку.
        add     di,(80-33)*2            ; Увеличить DI до начала следующей строки.
        dec     dx                      ; Уменьшить счетчик строк.
        jnz     restore_screen_loop     ; Если он не ноль - продолжить.
        ret
restore_screen  endp

; Процедура display_all.
; Выводит на экран текущее состояние всплывающего окна нашей программы.
display_all     proc    near

; Шаг 1: вписать значение текущего выделенного байта в нижнюю строку окна.
        mov     al,byte ptr current_char ; AL = выбранный байт.
        push    ax
        shr     al,4                    ; Старшие четыре байта.
        cmp     al,10                   ; Три команды,
        sbb     al,69h                  ; преобразующие цифру в AL
        das                             ; в ее ASCII-код (0 - 9, A - F).
        mov     byte ptr hex_byte1,al   ; Записать символ на его
                                        ; место в нижней строке.
        pop     ax
```

```
        and     al,0Fh                  ; Младшие четыре бита.
        cmp     al,10                   ; То же преобразование.
        sbb     al,69h
        das
        mov     byte ptr hex_byte2,al   ; Записать младшую цифру.

; Шаг 2: вывод на экран окна. Было бы проще хранить его как массив и выводить
; командой movsw, как и буфер в процедуре restore_screen, но такой массив займет еще
; 1190 байт в резидентной части. Код этой части процедуры display_all - всего 69 байт.
; Шаг 2.1: вывод первой строки.
        mov     ah,1Fh                  ; Атрибут белый на синем.
        mov     di,START_POSITION       ; ES:DI - адрес в видеопамяти.
        mov     si,offset display_line1 ; DS:SI - адрес строки.
        mov     cx,33                   ; Счетчик символов в строке.
display_loop1:
        mov     al,byte ptr [si]        ; Прочитать символ в AL
        stosw                           ; и вывести его с атрибутом из AH.
        inc     si                      ; Увеличить адрес символа в строке.
        loop    display_loop1

; Шаг 2.2: вывод собственно таблицы
        mov     dx,16                   ; Счетчик строк.
        mov     al,-1                   ; Выводимый символ.
display_loop4:                          ; Цикл по строкам.
        add     di,(80-33)*2            ; Увеличить DI до начала
        push    ax                      ; следующей строки.
        mov     al,0B3h
        stosw                           ; Вывести первый символ (0B3h).
        pop     ax
        mov     cx,16                   ; Счетчик символов в строке.
display_loop3:                          ; Цикл по символам в строке.
        inc     al                      ; Следующий ASCII-символ.
        stosw                           ; Вывести его на экран.
        push    ax
        mov     al,20h                  ; Вывести пробел.
        stosw
        pop     ax
        loop    display_loop3           ; И так 16 раз.
        push    ax
        sub     di,2                    ; Вернуться назад на 1 символ
        mov     al,0B3h                 ; и вывести 0B3h на месте
        stosw                           ; последнего пробела.
        pop     ax
        dec     dx                      ; Уменьшить счетчик строк.
        jnz     display_loop4

; Шаг 2.3: вывод последней строки.
        add     di,(80-33)*2            ; Увеличить DI до начала следующей строки.
        mov     cx,33                   ; Счетчик символов в строке.
        mov     si,offset display_line2 ; DS:SI - адрес строки.
```

```
display_loop2:
        mov     al,byte ptr [si]        ; Прочитать символ в AL.
        stosw                           ; Вывести его с атрибутом на экран.
        inc     si                      ; Увеличить адрес символа в строке.
        loop    display_loop2

; Шаг 3: подсветка (изменение атрибута) у текущего выделенного символа.
        mov     al,byte ptr current_char ; AL = текущий символ.
        mov     ah,0
        mov     di,ax
        and     di,0Fh                  DI = остаток от деления на 16 (номер в строке).
        shl     di,2                    Умножить его на 2, так как на экране
                                        ; используется слово на символ, и еще раз на 2,
                                        ; так как между символами - пробелы.
        shr     ax,4                    AX = частное от деления на 16 (номер строки).
        imul    ax,ax,80*2              Умножить его на длину строки на экране,
        add     di,ax                   сложить результаты,
        add     di,START_POSITION+2+80*2+1 ; добавить адрес начала окна + 2,
                                        ; чтобы пропустить первый столбец, + 80 x 2,
                                        ; чтобы пропустить первую строку, + 1, чтобы
                                        ; получить адрес атрибута, а не символа.
        mov     al,071h                 Атрибут - синий на сером.
        stosb                           ; Вывод на экран.

        ret
display_all     endp

int09h_handler  endp                    ; Конец обработчика INT 09h.

; Буфер для хранения содержимого части экрана, которая накрывается нашим окном.
screen_buffer   db      1190 dup(?)

; Первая строка окна.
display_line1   db      0DAh,11 dup (0C4h),'* ASCII *',11 dup (0C4h),0BFh

; Последняя строка окна.
display_line2   db      0C0h,11 dup (0C4h),'* Hex "
hex_byte1       db      ?               ; Старшая цифра текущего байта.
hex_byte2       db      ?               ; Младшая цифра текущего байта.
                db      " *",10 dup (0C4h),0D9h

hw_reset2D:     retf                    ; ISP: минимальный hw reset.

; Обработчик прерывания INT 2Dh.
; Поддерживает функции AMIS 3.6 00h, 02h, 03h, 04h и 05h.
int2Dh_handler  proc    far
        jmp     short actual_int2Dh_handler     ; ISP: пропустить блок.
old_int2Dh      dd      ?                       ; ISP: старый обработчик.
                dw      424Bh                   ; ISP: сигнатура.
                db      00h                     ; ISP: программное прерывание.
        jmp     short hw_reset2D                ; ISP: ближний jmp на hw_reset.
                db      7 dup (0)               ; ISP: зарезервировано.
```

```
actual_int2Dh_handler:                  ; Начало собственно обработчика INT 2Dh.
                db      80h,0FCh          Начало команды CMP AH, число.
mux_id          db      ?                 Идентификатор программы.
        je      its_us                  ; Если вызывают с чужим AH - это не нас.
        jmp     dword ptr cs:old_int2Dh
its_us:
        cmp     al,06                   ; Функции 06h и выше
        jae     int2D_no                ; не поддерживаются.
        cbw                             ; AX = номер функции.
        mov     di,ax                   ; DI = номер функции.
        shl     di,1                    ; Умножить его на 2, так как jumptable. -
                                        ; таблица слов.
        jmp     word ptr cs:jumptable[di]  Косвенный переход на обработчики функций.
jumptable       dw      offset int2D_00,offset int2D_no
                dw      offset int2D_02,offset int2D_03
                dw      offset int2D_04,offset int2D_05

int2D_00:                               ; Проверка наличия.
        mov     al,0FFh                 ; Этот номер занят.
        mov     cx,0100h                ; Номер версии 1.0.
        push    cs
        pop     dx                      ; DX:DI - адрес AMIS-сигнатуры.
        mov     di,offset amis_sign
        iret
int2D_no:                               ; Неподдерживаемая функция.
        mov     al,00h                  ; Функция не поддерживается.
        iret
int2D_02:                               ; Выгрузка программы.
        mov     byte ptr cs:disable_point,0CFh ; Записать код команды IRET
                                ; по адресу disable_point в обработчик INT 09h.
        mov     al,04h          ; Программа дезактивизирована, но сама
                                ; выгрузиться не может.
        mov     bx,cs           ; BX - сегментный адрес программы.
        iret
int2D_03:                       ; Запрос на активизацию для "всплывающих" программ.
        cmp     byte ptr we_are_active, 0 ; Если окно не на экране,
        je      already_popup
        call    save_screen             ; сохранить область экрана,
        push    cs
        pop     ds
        call    display_all             ; вывести окно
        mov     byte ptr we_are_active, 1 ; и поднять флаг.
already_popup:
        mov     al,03h                  ; Код 03: программа активизирована.
        iret

int2D_04:                               ; Получить список перехваченных прерываний.
        mov     dx,cs                   ; Список в DX:BX.
        mov     bx,offset amis_hooklist
        iret
```

```
int2D_05:                                   ; Получить список "горячих" клавиш.
        mov     al,0FFh                     ; Функция поддерживается.
        mov     dx,cs                       ; Список в DX:BX.
        mov     bx,offset amis_hotkeys
        iret
int2Dh_handler  endp

; AMIS: сигнатура для резидентных программ.
amis_sign       db      "Cubbi..."          ; 8 байт - имя автора.
                db      "ASCII..."          ; 8 байт - имя программы.
                db      "ASCII display and input utility",0 ; ASCIZ-комментарий
                                            ; не более 64 байт.

; AMIS: список перехваченных прерываний.
amis_hooklist   db      09h
                dw      offset int09h_handler
                db      2Dh
                dw      offset int2Dh_handler

; AMIS: список "горячих" клавиш.
amis_hotkeys    db      01h                 ; Клавиши проверяются после стандартного
                                            ; обработчика INT 09h.
                db      1                   ; Число клавиш.
                db      1Eh                 ; Скан-код клавиши (A).
                dw      08h                 ; Требуемые флаги (любая Alt).
                dw      0                   ; Запрещенные флаги.
                db      1                   ; Клавиша проглатывается.

; Конец резидентной части.
; Начало процедуры инициализации.

initialize      proc near
        mov     ah,9
        mov     dx,offset usage             ; Вывести информацию о программе.
        int     21h

; Проверить, не установлена ли уже наша программа.
        mov     ah,-1                       ; Сканирование номеров от 0FFh до 00h.
more_mux:
        mov     al,00h                      ; Функция 00h - проверка наличия программы.
        int     2Dh                         ; Мультиплексорное прерывание AMIS.
        cmp     al,00h                      ; Если идентификатор свободен,
        jne     not_free
        mov     byte ptr mux_id,ah          ; записать его номер прямо в код
                                            ; обработчика int 2Dh.
        jmp short next_mux
not_free:
        mov     es,dx                       ; Иначе - ES:DI = адрес их сигнатуры,
        mov     si,offset amis_sign         ; DS:SI = адрес нашей сигнатуры.
        mov     cx,16                       ; Сравнить первые 16 байт.
        repe    cmpsb
        jcxz    already_loaded              ; Если они не совпадают,
```

```
next_mux:
        dec     ah                      ; перейти к следующему идентификатору,
        jnz     more_mux                ; пока это не 0
; (на самом деле в нашем примере сканирование происходит от OFFh до 01h,
; так как 0 мы используем в качестве признака отсутствия свободного номера
; в следующей строке).
free_mux_found:
        cmp     byte ptr mux_id,0       ; Если мы ничего не записали,
        je      no_more_mux               идентификаторы кончились.

        mov     ax,352Dh                ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Получить адрес обработчика INT 2Dh
        mov     word ptr old_int2Dh,bx  ; и поместить его в old_int2Dh.
        mov     word ptr old_int2Dh+2,es
        mov     ax,3509h                ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Получить адрес обработчика INT 09h
        mov     word ptr old_int09h,bx  ; и поместить его в old_int09h.
        mov     word ptr old_int09h+2,es

        mov     ax,252Dh                ; AH = 25h, AL = номер прерывания.
        mov     dx,offset int2Dh_handler ; DS:DX - адрес нашего
        int     21h                     ; обработчика.
        mov     ax,2509h                ; AH = 25h, AL = номер прерывания.
        mov     dx,offset int09h_handler ; DS:DX - адрес нашего
        int     21h                     ; обработчика.

        mov     ah,49h                  ; AH = 49h.
        mov     es,word ptr envseg      ; ES = сегментный адрес среды DOS.
        int     21h                     ; Освободить память.

        mov     ah,9
        mov     dx,offset installed_msg ; Вывод строки об успешной
        int     21h                     ; инсталляции.

        mov     dx,offset initialize    ; DX - адрес первого байта за
                                        ; концом резидентной части.
        int     27h                     ; Завершить выполнение, оставшись
                                        ; резидентом.

; Сюда передается управление, если наша программа обнаружена в памяти.
already_loaded:
        mov.    ah,9                    ; AH = 09h
        mov     dx,offset already_msg   ; Вывести сообщение об ошибке
        int     21h
        ret                             ; и завершиться нормально.

; Сюда передается управление, если все 255 функций мультиплексора заняты
; резидентными программами.
no_more_mux:
        mov     ah,9
        mov     dx,offset no_more_mux_msg
        int     21h
        ret
```

```
; Текст, который выдает программа при запуске:
usage           db      "ASCII display and input program"
                db      " v1.0",0Dh,0Ah
                db      "Alt-A   - активизация",0Dh,0Ah
                db      "Стрелки - выбор символа",0Dh,0Ah
                db      "Enter   - ввод символа",0Dh,0Ah
                db      "Escape  - выход",0Dh,0Ah
                db      "$"
; Текст, который выдает программа, если она уже загружена:
already_msg     db      "Ошибка: программа уже загружена",0Dh,0Ah,'$'
; Текст, который выдает программа, если все функции мультиплексора заняты:
no_more_mux_msg db      "Ошибка: Слишком много резидентных программ"
                db      0Dh,0Ah,'$'
; Текст, который выдает программа при успешной установке:
installed_msg   db      "Программа загружена в память",0Dh,0Ah,'$'

initialize      endp
        end     start
```

Резидентная часть этой программы занимает в памяти целых 2064 байта, из которых на собственно коды команд приходится только 436. Это вполне терпимо, учитывая, что обычно программа вроде ascii.com запускается перед простыми текстовыми редакторами для DOS (edit, multiedit, встроенные редакторы оболочек типа Norton Commander и т. д.), которые не требуют для своей работы полностью свободной памяти. В других случаях, как, например, при создании программы, копирующей изображение с экрана в файл, может оказаться, что на счету Каждый байт. Такие программы часто применяют для сохранения изображений из компьютерных игр, которые задействуют все ресурсы компьютера по максимуму. Здесь резидентным программам приходится размещать данные, а иногда и часть кода в старших областях памяти, пользуясь спецификациями HMA, UMB, EMS или XMS. В следующей главе рассмотрен простой пример именно такой программы.

5.9.3. Выгрузка резидентной программы из памяти

Чтобы выгрузить резидентную программу из памяти, необходимо сделать три вещи: закрыть открытые программой файлы и устройства, восстановить все перехваченные векторы прерываний, и наконец, освободить всю занятую программой память. Трудность может вызвать второй шаг, так как после нашего резидента могли быть загружены другие программы, перехватившие те же прерывания. Если в такой ситуации восстановить вектор прерывания в значение, которое он имел до загрузки нашего резидента, программы, загруженные позже, не будут получать управление. Более того, они не будут получать управление только по тем Прерываниям, которые у них совпали с прерываниями, перехваченными нашей программой, в то время как другие векторы прерываний будут все еще указывать на их обработчики, что почти наверняка приведет к ошибкам. Поэтому, если хоть один вектор прерывания не указывает на наш обработчик, выгружать резидентную программу нельзя. Это всегда было главным вопросом, и спецификации AMIS и IBM ISP (см. предыдущий раздел) являются возможным решением обозначенной

проблемы. Если вектор прерывания не указывает на нас, имеет смысл проверить, не указывает ли он на ISP-блок (первые два байта должны быть 0EBh 10h, а байты 6 и 7 - K и B), и, если это так, взять в качестве вектора значение из этого блока и т. д. Кроме того, программы могут изменять порядок, в котором обработчики одного и того же прерывания вызывают друг друга.

Последний шаг в выгрузке программы - освобождение памяти - можно выполнить вручную, вызывая функцию DOS 49h на каждый блок памяти, который программа выделяла через функцию 48h, на блок с окружением DOS, если он не освобождался при загрузке, и наконец, на саму программу. Однако есть способ заставить DOS сделать все это (а также закрыть открытые файлы и вернуть код возврата) автоматически, вызвав функцию 4Ch и объявив резидент текущим процессом. Посмотрим, как это делается на примере резидентной программы, занимающей много места в памяти. Кроме того, этот пример реализует все приемы, использующиеся для вызова функций DOS из обработчиков аппаратных прерываний, о которых рассказано в разделе 5.8.3.

```
; scrgrb.asm
; Резидентная программа, сохраняющая изображение с экрана в файл.
; Поддерживается только видеорежим 13h (320x200x256) и только один файл.

; HCI:
; Нажатие Alt-G создает файл scrgrb.bmp в текущей директории с изображением,
; находившимся на экране в момент нажатия клавиши.
; Запуск с командной строкой /u выгружает программу из памяти.

; API:
; Программа занимает первую свободную функцию прерывания 2Dh (кроме нуля)
; в соответствии со спецификацией AMIS 3.6.
; Поддерживаемые подфункции AMIS: 00h, 02h, 03h, 04h, 05h.
; Все обработчики прерываний построены в соответствии с IMB ISP.

; Резидентная часть занимает в памяти 1056 байт, если присутствует EMS,
; и 66 160 байт, если EMS не обнаружен.

        .model  tiny
        .code
        .186                    ; Для сдвигов и команд pusha/popa.
        org     2Ch
envseg          dw      ?       ; Сегментный адрес окружения.
        org     80h
cmd_len         db      ?       ; Длина командной строки.
cmd_line        db      ?       ; Командная строка.
        org     100h            ; COM-программа.
start:
        jmp     initialize      ; Переход на инициализирующую часть.

; Обработчик прерывания 09h (IRQ1).

int09h_handler  proc    far
        jmp     short actual_int09h_handler     ; Пропустить ISP.
```

```
old_int09h      dd      ?
                dw      424Bh
                db      00h
        jmp     short hw_reset
                db      7 dup (0)
actual_int09h_handler:          ; Начало собственно обработчика INT 09h.
        pushf
        call    dword ptr cs:old_int09h ; Сначала вызвать старый
                                ; обработчик, чтобы он завершил аппаратное
                                ; прерывание и передал код в буфер.

        pusha                   ; Это аппаратное прерывание - надо
        push    ds              ; сохранить все регистры.
        push    es

        push    0040h
        pop     ds              ; DS = сегментный адрес области данных BIOS.
        mov     di,word ptr ds:001Ah    ; Адрес головы буфера клавиатуры.
        cmp     di,word ptr d's:001Ch   ; Если он равен адресу хвоста,
        je      exit_09h_handler        ; буфер пуст и нам делать нечего.

        mov     ax,word ptr [di]        ; Иначе: считать символ.
        cmp     ah,22h                  ; Если это не G (скан-код 22h),
        jne     exit_09h_handler        ; выйти.

        mov     al,byte ptr ds:0017h    ; Байт состояния клавиатуры.
        test    al,08h                  ; Если Alt не нажата,
        jz      exit_09h_handler        ; выйти.

        mov     word ptr ds:001Ch,di    ; Иначе: установить адреса головы и хвоста
                                        ; буфера равными, то есть опустошить его.
        call    do_grab                 ; Подготовить BMP-файл с изображением.
        mov     byte ptr cs:io_needed,1 ; Установить флаг
                                        ; требующейся записи на диск.
        cli
        call    safe_check              ; Проверить, можно ли вызвать DOS.
        jc      exit_09h_handler
        sti
        call    do_io                   ; Если да - записать файл на диск.

exit_09h_handler:
        pop     es
        pop     ds                      ; Восстановить регистры
        popa
        iret                            ; и вернуться в прерванную программу.
int09h_handler  endp

hw_reset:retf

; Обработчик INT 08h (IRQ0)

int08h_handler  proc    far
        jmp     short actual_int08h_handler ; Пропустить ISP.
```

```
old_int08h      dd      ?
                dw      424Bh
                db      00h
        jmp     short hw_reset
                db      7 dup (0)
actual_int08h_handler:                  ; Собственно обработчик.
        pushf
        call    dword ptr cs:old_int08h ; Сначала вызвать стандартный обработчик,
                                        ; чтобы он завершил аппаратное прерывание
                                        ; (пока оно не завершено, запись на диске невозможна).
        pusha
        push    ds
        cli             ; Между любой проверкой глобальной переменной и принятием
                        ; решения по ее значению - не повторно входимая область,
                        ; прерывания должны быть запрещены.
        cmp     byte ptr cs:io_needed,0 ; Проверить,
        je      no_io_needed            ; нужно ли писать на диск.

        call    safe_check              ; Проверить,
        jc      no_io_needed            ; можно ли писать на диск.
        sti                             ; Разрешить прерывания на время записи.

        call    do_io                   ; Запись на диск.
no_io_needed:
        pop     ds
        popa
        iret
int08h_handler  endp

; Обработчик INT 13h.
; Поддерживает флаг занятости INT 13h, который тоже надо проверять перед записью на диск.

int13h_handler  proc    far
        jmp     short actual_int13h_handler     ; Пропустить ISP.
old_int13h      dd      ?
                dw      424Bh
                db      00h
        jmp     short hw_reset
                db      7 dup (0)
actual_int13h_handler:                  ; Собственно обработчик.
        pushf
        inc     byte ptr cs:bios_busy   ; Увеличить счетчик занятости INT 13h.
        cli
        call    dword ptr cs:old_int13h
        pushf
        dec     byte ptr cs:bios_busy   ; Уменьшить счетчик.
        popf
        ret     2       ; Имитация команды IRET, не восстанавливающая флаги из стека,
                        ; так как обработчик INT 13h возвращает некоторые
                        ; результаты в регистре флагов, а не в его копии,
                        ; хранящейся в стеке. Он тоже завершается командой ret 2.
int13h_handler  endp
```

```
; Обработчик INT 28h.
; Вызывается DOS, когда она ожидает ввода с клавиатуры и функциями DOS можно
; пользоваться.

int28h_handler  proc    far
        jmp     short actual_int28h_handler     ; Пропустить ISP.
old_int28h      dd      ?
                dw      424Bh
                db      00h
        jmp     short hw_reset
                db      7 dup (0)
actual_int28h_handler:
        pushf
        push    di
        push    ds
        push    cs
        pop     ds
        cli
        cmp     byte ptr io_needed,0        Проверить,
        je      n.o_io_needed2              нужно ли писать на диск.
        lds     di,dword ptr in_dos_addr
        cmp     byte ptr [di+1],1           Проверить,
        ja      no_io_needed2               можно ли писать на диск (флаг
                                          ; занятости DOS не должен быть больше 1).
        sti
        call    do_io                       Запись на диск.
no_io_needed2:
        pop     ds
        pop     di
        popf
        jmp     dword ptr cs:old_int28h   ; Переход на старый обработчик INT 28h.
int28h_handler  endp

; Процедура do_grab.
; Помещает в буфер палитру и содержимое видеопамяти, формируя BMP-файл.
; Считает, что текущий видеорежим - 13h.
do_grab         proc    near
        push    cs
        pop     ds

        call    ems_init                ; Отобразить наш буфер в окно EMS.

        mov     dx,word ptr cs:buffer_seg
        mov     es,dx                   ; Поместить сегмент с буфером в ES и DS
        mov     ds,dx                   ; для следующих шагов процедуры.

        mov     ax,1017h                ; Функция 1017h - чтение палитры VGA
        mov     bx,0                    ; начиная с регистра палитры 0.
        mov     cx,256                  ; Все 256 регистров.
        mov     dx,BMP_header_length    ; Начало палитры в BMP.
        int     10h                     ; Видеосервис BIOS.
```

```
; Перевести палитру из формата, в котором ее показывает функция 1017h
; (три байта на цвет, в каждом байте 6 значимых бит),
; в формат, используемый в BMP-файлах
; (4 байта на цвет, в каждом байте 8 значимых бит)
        std                                     ; Движение от конца к началу.
        fflov   .si,BMP_header_length+256*3-1   ; SI - конец 3-байтной палитры.
        mov     di,BMP_header_length+256*4-1    ; DI - конец 4-байтной палитры.
        mov     cx,256                          ; CX - число цветов.
adj_pal: mov    al,0
        stosb                                   ; Записать четвертый байт (0).
        lodsb                                   ; Прочитать третий байт.
        shl     al,2                            ; Масштабировать до 8 бит.
        push    ax
        lodsb                                   ; Прочитать второй байт.
        shl     al,2                            ; Масштабировать до 8 бит.
        push    ax
        lodsb                                   ; Прочитать третий байт.
        shl     al,2                            ; Масштабировать до 8 бит
        stosb                                   ; и записать эти три байта
        pop     ax                              ; в обратном порядке.
        stosb
        pop     ax
        stosb
        loop    adj_pal

; Копирование видеопамяти в BMP.
; В формате BMP строки изображения записываются от последней к первой, так что
; первый байт соответствует нижнему левому пикселу.
        cld                     ; Движение от начала к концу (по строке).
        push    0A000h
        pop     ds
        mov     si,320*200      ;; DS:SI - начало последней строки на экране.
        mov     di,bfoffbits    ;; ES:DI - начало данных в BMP.
        mov     dx,200          ;; Счетчик строк.
bmp_write_loop:
        mov     cx,320/2        ; Счетчик символов в строке.
        rep     movsw           ; Копировать целыми словами, так быстрее.
        sub     si,320*2        ; Перевести SI на начало предыдущей строки.
        dec     dx              ; Уменьшить счетчик строк.
        jnz     bmp_write_loop  ; Если 0 - выйти из цикла.

        call    ems_reset       ; Восстановить состояние EMS до вызова do_grab.
        ret
do_grab         endp

; Процедура do_io.
; Создает файл и записывает в него содержимое буфера.

do_io           proc    near
        push cs
        pop ds
```

```
        mov     byte ptr io_needed,0    ; Сбросить флаг требующейся записи на диск.
        call    ems_init                ; Отобразить, в окно EMS наш буфер.

        mov     ah,6Ch                  ; Функция DOS 6Ch.
        mov     bx,2                    ; Доступ - на чтение/запись.
        mov     cx,0                    ; Атрибуты - обычный файл.
        mov     dx,12h                  ; Заменять файл, если он существует;
                                        ; создавать, если нет.
        mov     si,offset filespec      ; DS:SI - имя файла.
        int     21h                     ; Создать/открыть файл.
        mov     bx,ax                   ; Идентификатор файла - в BX.

        mov     ah,40h                  ; Функция DOS 40h.
        mov     cx,bfsize               ; Размер BMP-файла.
        mov     ds,word ptr buffer_seg
        mov     dx,0                    ; DS:DX - буфер для файла.
        int     21h                     ; Запись в файл или устройство.

        mov     ah,68h                  ; Сбросить буфера на диск.
        int     21h

        mov     ah,3Eh                  ; Закрыть файл.
        int     21h

        call    ems_reset

        ret
do_io           endp

; Процедура ems_init.
; Если буфер расположен в EMS, подготавливает его для чтения/записи.

ems_init        proc near

        cmp     dx,word ptr ems_handle  Если не используется EMS
        cmp     dx,0                    (EMS-идентификаторы начинаются с 1),
        je      ems_init_exit           ничего не делать.

        mov     ax,4700h                Функция EMS 47h:
        int     67h                     сохранить EMS-контекст.

        mov     ax,4100h                Функция EMS 41h:
        int     67h                     определить адрес окна EMS.
        mov     word ptr buffer_seg,bx  Сохранить его.

        mov     ax,4400h                Функция EMS 44h:
        mov     bx,0                    Начиная со страницы 0,
        int     67h                     отобразить страницы EMS в окно.
        mov     ax,4401h
        inc     bx
        int     67h                     ; Страница 1.
        mov     ax,4402h
        inc     bx
        int     67h                     ; Страница 2.
        mov     ax,4403h
```

```
        inc     bx
        int     67h                     ; Страница 3.
ems_init_exit:
        ret
ems_init        endp

; Процедура ems_reset.
; Восстанавливает состояние EMS.
ems_reset proc near
        mov     dx,word ptr cs:ems_handle
        cmp     dx,0
        je      ems_reset_exit

        mov     ax,4800h                ; Функция EMS 48h:
        int     67h                     ; восстановить EMS-контекст.
ems_reset_exit:
        ret
ems_reset       endp

; Процедура safe_check.
; Возвращает CF = 0, если в данный момент можно пользоваться функциями DOS,
; и CF = 1, если нельзя.
safe_check      proc    near
        push    es
        push    cs
        pop     ds

        les     di,dword ptr in_dos_addr ; Адрес флагов занятости DOS.
        cmp     word ptr es:[di],0      ; Если один из них не 0,
        pop     es
        jne     safe_check_failed       ; пользоваться DOS нельзя.

        cmp     byte ptr bios_busy,0    ; Если выполняется прерывание 13h,
        jne     safe_check_failed       ; тоже нельзя.

        clc                             ; CF = 0.
        ret
safe_check_failed:
        stc                             ; CF = 1.
        ret
safe_check      endp

in_dos_addr     dd      ?               ; Адрес флагов занятости DOS.
io_needed       db      0               ; 1, если надо записать файл на диск.
bios_busy       db      0               ; 1, если выполняется прерывание INT 13h.
buffer_seg      dw      0               ; Сегментный адрес буфера для файла.
ems_handle      dw      0               ; Идентификатор EMS.
filespec        db      'scrgrb.bmp',0  ; Имя файла.

; Обработчик INT 2Dh

hw_reset2D:retf
int2Dh_handler  proc    far
        jmp     short actual_int2Dh_handler     ; Пропустить ISP.
```

```
old_int2Dh      dd      ?
                dw      424Bh
                db      00h
        jmp     short hw_reset2D
                db      7 dup (0)
actual_int2Dh_handler:                  ; Собственно обработчик.
                db      80h,0FCh        ; Начало команды CMP AH, число.
mux_id          db      ?               ; Идентификатор программы.
        je      its_us                  ; Если вызывают с чужим AH - это не нас.
        jmp     dword ptr cs:old_int2Dh
its_us:
        cmp     al,06                     Функции AMIS 06h и выше
        jae     int2D_no                  не поддерживаются.
        cbw                             ; AX = номер функции.
        mov     di,ax                     DI = номер функции.
        shl     di,1                      x 2, так как jumptable - таблица слов.
        jmp     word ptr cs:jumptable[di] ; Переход на обработчик функции.
jumptable       dw      offset int2D_00,offset int2D_no
                dw      offset int2D_02,offset int2D_no
                dw      offset int2D_04,offset int2D_05

int2D_00:                               ; Проверка наличия.
        mov     al,0FFh                 ; Этот номер занят.
        mov     cx,0100h                ; Номер версии программы 1.0.
        push    cs
        pop     dx                      ; DX:DI - адрес AMIS-сигнатуры.
        mov     di,offset amis_sign
        iret
int2D_no:                               ; Неподдерживаемая функция.
        mov     al,00h                  ; Функция не поддерживается.
        iret

unload_failed:  ; Сюда передается управление, если хоть один из векторов
                ; прерываний был перехвачен кем-то после нас.
        mov     al,01h                  ; Выгрузка программы не удалась.
        iret
int2D_02:                               ; Выгрузка программы из памяти.
        cli                             ; Критический участок.
        push    0
        pop     ds                      ; DS - сегментный адрес
                                        ; таблицы векторов прерываний.
        mov     ax,cs                   ; Наш сегментный адрес.
; Проверить, все ли перехваченные прерывания по-прежнему указывают на нас.
; Обычно достаточно проверить только сегментные адреса (DOS не загрузит другую
; программу с нашим сегментным адресом).
        cmp     ax,word ptr ds:[09h*4+2]
        jne     unload_failed
        cmp     ax,word ptr ds:[13h*4+2]
        jne     unload_failed
        cmp     ax,word ptr ds:[08h*4+2]
```

```
        jne     unload_failed
        cmp     ax,word ptr ds:[28h*4+2]
        jne     unload_failed
        cmp     ax,word ptr ds:[2Dh*4+2]
        jne     unload_failed

        push    bx                      ; Адрес возврата - в стек.
        push    dx

; Восстановить старые обработчики прерываний.
        mov     ax,2509h
        lds     dx,dword ptr cs:old_int09h
        int     21h
        mov     ax,2513h
        lds     dx,dword ptr cs:old_int13h
        int     21h
        mov     ax,2508h
        lds     dx,dword ptr cs:old_int08h
        int     21h
        mov     ax,2528h .
        lds     dx,dword  ptr  cs:old_int28h
        int     21h
        mov     ax,252Dh
        lds     dx,dword ptr cs:old_int2Dh
        int     21h

        mov     dx,word ptr cs:ems_handle ; Если используется EMS.
        cmp     dx,0
        je      no_ems_to_unhook
        mov     ax,4500h                ; Функция EMS 45h:
        int     67h                     ; освободить выделенную память.
        jmp     short ems_unhooked
no_ems_to_unhook:
ems_unhooked:

; Собственно выгрузка резидента.
        mov     ah,51h                  ; Функция DOS 51h:
        int     21h                     ; получить сегментный адрес PSP прерванного
                                        ; процесса (в данном случае PSP - копии
                                        ; нашей программы, запущенной с ключом /и).
        mov     word ptr cs:[16h],bx    ; Поместить его в поле
                                        ; "сегментный адрес предка" в нашем PSP.
        pop     dx                      ; Восстановить адрес возврата из стека
        pop     bx
        mov     word ptr cs:[0Ch],dx    ; и поместить его в поле
        mov     word ptr cs:[0Ah],bx    ; "адрес перехода при
                                        ; завершении программы" в нашем PSP.
        push    cs
        pop     bx                      ; BX = наш сегментный адрес PSP.
        mov     ah,50h                  ; Функция DOS 50h:
        int     21h                     ; установить текущий PSP.
```

```
; Теперь DOS считает наш резидент текущей программой, а scrgrb.com /и - вызвавшим
; его процессом, которому и передаст управление после вызова следующей функции.
        mov     ax,4CFFh                ; Функция DOS 4Ch:
        int     21h                     ; завершить программу.

int2D_04:                               ; Получить список перехваченных
                                        ; прерываний.
        mov     dx,cs                   ; Список в DX:BX.
        mov     bx,offset amis_hooklist
        iret
int2D_05:                               ; Получить список "горячих" клавиш.
        mov     al,0FFh                 ; Функция поддерживается.
        mov     dx,cs                   ; Список в DX:BX.
        mov     bx,offset amis_hotkeys
        iret
int2Dh_handler  endp

; AMIS: Сигнатура для резидентной программы.
amis_sign       db      "Cubbi..."      ; 8 байт.
                db      "ScrnGrab"      ; 8 байт.
                db      "Simple screen grabber using EMS",0

; AMIS: Список перехваченных прерываний.
amis_hooklist   db      09h
                dw      offset int09h_handler
                db      08h
                dw      offset int08h_handler
                db      28h
                dw      offset int28h_handler
                db      2Dh
                dw      offset int2Dh_handler
; AMIS: Список "горячих" клавиш.
amis_hotkeys    db      1
                db      1
                db      22h             ; Скан-код клавиши (G).
                dw      08h             ; Требуемые флаги клавиатуры.
                dw      0
                db      1

; Конец резидентной части.
; Начало процедуры инициализации.

initialize      proc near
        jmp     short initialize_entry_point ; Пропустить различные
                ; варианты выхода без установки резидента, помещенные здесь
                ; потому, что на них передают управление команды условного
                ; перехода, имеющие короткий радиус действия.

exit_with_message:
        mov     ah,9                    ; Функция вывода строки на экран.
        int     21h
        ret                             ; Выход из программы.
```

```
already_loaded:                         ; Если программа уже загружена в память.
        cmp     byte ptr unloading,1    ; Если мы не были вызваны с /и.
        je      do_unload
        mov     dx,offset already_msg
        jmp     short exit_with_message

no_more_mux:    ; Если свободный идентификатор INT 2Dh не найден.
        mov     dx,offset no_more_mux_msg
        jmp     short exit_with_message

cant_unload1:                           ; Если нельзя выгрузить программу.
        mov     dx,offset cant_unload1_msg
        jmp     short exit_with_message

do_unload:      ; Выгрузка резидента: при передаче управления сюда AH содержит
                ; идентификатор программы - 1.
        inc     ah
        mov     al,02h                  AMIS-функция выгрузки резидента.
        mov     dx,cs                   ; Адрес возврата
        mov     bx,offset exit_point    ; в DX:BX.
        int     2Dh                     ; Вызов нашего резидента через мультиплексор.

        push    cs                      ; Если управление пришло сюда -
                                        ; выгрузки не было.
        pop     ds
        mov     dx,offset cant_unload2_msg
        jmp     short  exit_with_message

exit_point:     ; Если управление пришло сюда - выгрузка произошла.
        push    cs
        pop     ds
        mov     dx,offset unloaded_msg
        push    0                       ; Чтобы сработала команда RET для выхода.
        jmp     short  exit_with_message

initialize_entry_point:                 ; Сюда передается управление в самом начале.
        cld

        cmp     byte ptr cmd_line[1],'/'
        jne     not_unload
        cmp     byte ptr cmd_line[2],'u' ; Если нас вызвали с /и,
        jne     not_unload
        mov     byte ptr unloading,1    ; выгрузить резидент.
not_unload:

        mov     ah,9
        mov     dx,offset usage         ; Вывод строки с информацией о программе.
        int     21h

        mov     ah,-1                   ; Сканирование от FFh до 01h.
more_mux:
        mov     al,00h                  ; Функция AMIS 00h - проверка наличия
                                        ; резидента.
```

```
        int     2Dh                     ; Мультиплексорное прерывание.
        cmp     al,00h                  ; Если идентификатор свободен,
        jne     not_free
        mov     byte ptr mux_id,ah      ; вписать его сразу в код обработчика.
        jmp     short next_mux
not_free:
        mov     es,dx                   ; Иначе - ES:DI = адрес AMIS-сигнатуры
                                        ; вызвавшей программы.
        mov     si,offset amis_sign     ; DS:SI = адрес нашей сигнатуры.
        mov     cx,16                   ; Сравнить первые 16 байт.
        repe    cmpsb
        jcxz    already_loaded          ; Если они не совпадают,
next_mux:
        dec     ah                      ; перейти к следующему идентификатору.
        jnz     more_mux                ; Если это 0

free_mux_found:
        cmp     byte ptr unloading,1    ; и если нас вызвали для выгрузки,
        je      cant_unload1            ; а мы пришли сюда - программы нет
                                        ; в памяти.
        cmp     byte ptr mux_id,0       ; Если при этом mux_id все еще 0,
        je      no_more_mux             ; идентификаторы кончились.

; Проверка наличия устройства EMMXXXX0.
        mov     dx,offset ems_driver
        mov     ax,3D00h
        int     21h                     ; Открыть файл/устройство.
        jc      no_emmx
        mov     bx,ax
        mov     ax,4400h
        int     21h             ; IOCTL: получить состояние файла/устройства.
        jc      no_ems
        test    dx,80h          ; Если старший бит DX = 0, EMMXXXX0 - файл.
        jz      no_ems
; Выделить память под буфер в EMS.
        mov     ax,4100h                ; Функция EMS 41h:
        int     67h                     ; получить адрес окна EMS.
        mov     bp,bx                   ; Сохранить его пока в BP.
        mov     ax,4300h                ; Функция EMS 43h:
        mov     bx,4                    ; Нам надо 4 x 16 Кб.
        int     67h                     ; Выделить EMS-память (идентификатор в DX).
        cmp     ah,0                    ; Если произошла ошибка (нехватка памяти?),
        jnz     ems_failed              ; не будем пользоваться EMS.
        mov     word ptr ems_handle,dx  ; Иначе: сохранить идентификатор
                                        ; для резидента.
        mov     ax,4400h                ; Функция 44h - отобразить
                                        ; EMS-страницы в окно.
        mov     bx,0
        int     67h                     ; Страница 0.
        mov     ax,4401h
```

```
        inc     bx
        int     67h             ; Страница 1.
        mov     ax,4402h
        inc     bx
        int     67h             ; Страница 2.
        mov     ax,4403h
        inc     bx
        int     67h             ; Страница 3.

        mov     ah,9
        mov     dx,offset ems_msg; Вывести сообщение об установке в EMS.
        int     21h

        mov     ax,bp
        jmp     short ems_used

ems_failed:
no_ems:                         ; Если EMS нет или он не работает,
        mov     ah,3Eh
        int     21h             ; Закрыть файл/устройство EMMXXXX0.
no_emmx:
; Занять общую память.
        mov     ah,9
        mov     dx,offset conv_msg ; Вывод сообщения об этом.
        int     21h

        mov     sp,length_of_program+100h+200h ; Перенести стек.

        mov     ah,4Ah          ; Функция DOS 4Ah.
next_segment = length_of_program+100h+200h+0Fh
next_segment = next_segment/16  ; Такая запись нужна только для
                                ; WASM, остальным ассемблерам это
                                ; можно было записать в одну строчку.
        mov     bx,next_segment ; Уменьшить занятую память, оставив
                                ; текущую длину нашей программы +100h
                                ; на PSP +200h на стек.
        int     21h

        mov     ah,48h          ; Функция 48h - выделить память.
bfsize_p = bfsize+0Fh
bfsize_p = bfsize_p/16
        mov     bx,bfsize_p     ; Размер BMP-файла 320x200x256 в 16-байтных
        int     21h             ; параграфах.

ems_used:
        mov     word ptr buffer_seg,ax   ; Сохранить адрес буфера для резидента.

; Скопировать заголовок BMP-файла в начало буфера.
        mov     cx,BMP_header_length
        mov     si,offset BMP_header
        mov     di,0
        mov     es,ax
        rep     movsb
```

```
; Получить адреса флага занятости DOS и флага критической ошибки
; (считая, что версия DOS старше 3.0).
        mov     ah,34h              ; Функция 34h - получить флаг занятости.
        int     21h
        dec     bx                  ; Уменьшить адрес на 1, чтобы он указывал
                                    ; на флаг критической ошибки,
        mov     word ptr in_dos_addr,bx
        fflov   word ptr in_dos_addr+2,es ; и сохранить его для резидента.

; Перехват прерываний.
        mov     ax,352Dh                ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Получить адрес обработчика INT 2Dh
        mov     word ptr old_int2Dh,bx  ; и поместить его в old_int2Dh.
        mov     word ptr old_int2Dh+2,es
        mov     ax,3528h                ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Получить адрес обработчика INT 28h
        mov     word ptr old_int28h,bx  ; и поместить его в old_int28h.
        mov     word ptr old_int28h+2,es
        mov     ax,3508h                ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Получить адрес обработчика INT 08h
        mov     word ptr old_int08h,bx  ; и поместить его в old_int08h.
        mov     word ptr old_int08h+2,es
        mov     ax,3513h                ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Получить адрес обработчика INT 13h
        mov     word ptr old_int13h,bx  ; и поместить его в old_int13h.
        mov     word ptr old_int13h+2,es
        mov     ax,3509h                ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Получить адрес обработчика INT 09h
        mov     word ptr old_int09h,bx  ; и поместить его в old_int09h.
        mov     word ptr old_int09h+2,es

        mov     ax,252Dh                ; AH = 25h, AL = номер прерывания.
        mov     dx,offset int2Dh_handler ; DS:DX - адрес обработчика.
        int     21h                     ; Установить новый обработчик INT 20h.
        mov     ax,2528h                ; AH = 25h, AL = номер прерывания.
        mov     dx,offset int28h_handler ; DS:DX - адрес обработчика.
        int     21h                     ; Установить новый обработчик INT 2:8h.
        mov     ax,2508h                ; AH = 25h, AL = номер прерывания.
        mov     dx,offset int08h_handler ; DS:DX - адрес обработчика.
        int     21h                     ; Установить новый обработчик INT 08h.
        mov     ax,2513h                ; AH = 25h, AL = номер прерывания.
        mov     dx,offset int13h_handler ; DS:DX - адрес обработчика.
        int     21h                     ; Установить новый обработчик INT 13h.
        mov     ax,2509h                ; AH = 25h, AL = номер прерывания.
        mov     dx,offset int09h_handler ; DS:DX - адрес обработчика.
        int     21h                     ; Установить новый обработчик INT 09h.

; Освободить память из-под окружения DOS.
        mov     ah,49h                  ; Функция DOS 49h.
        mov     es,word ptr envseg      ; ES = сегментный адрес окружения DOS.
        int     21h                     ; Освободить память.
```

```
; Оставить программу резидентной.
        mov     dx,offset initialize    ; DX - адрес первого байта за концом
                                        ; резидентной части.
        int     27h                     ; Завершить выполнение, оставшись
                                        ; резидентом.
initialize      endp

ems_driver      db      'EMMXXXX0',0    ; Имя EMS-драйвера для проверки.

; Текст, который выдает программа при запуске:
usage           db      'Простая программа для копирования экрана только из'
                db      ' видеорежима 13h',0Dh,0Ah
                db      ' Alt-G      - записать копию экрана в scrgrb.bmp'
                db      0Dh,0Ah
                db      ' scrgrb.com /u - выгрузиться из памяти',0Dh,0Ah
                db      '$'
; Тексты, которые выдает программа при успешном выполнении:
ems_msg         db      'Загружена в EMS',0Dh,0Ah,'$'
conv_msg        db      'Не загружена в EMS',0Dh,0Ah,'$'
unloaded_msg    db      'Программа успешно выгружена из памяти',0Dh,0Ah,'$'
; Тексты, которые выдает программа при ошибках:
already_msg     db      'Ошибка: Программа уже загружена',0Dh,0Ah,'$'
no_more_mux_msg db      'Ошибка: Слишком много резидентных программ'
                db      0Dh,0Ah,'$'
cant_unload1_msg db     'Ошибка: Программа не обнаружена в памяти',0Dh,0Ah,'$'
cant_unload2_msg db     'Ошибка: Другая программа перехватила прерывания'
                db      0Dh,0Ah,'$'
unloading       db      0               ; 1, если нас запустили с ключом /u.

; BMP-файл (для изображения 320x200x256)
BMP_header      label   byte
; Файловый заголовок.
BMP_file_header db      "BM"            ; Сигнатура.
                dd      bfsize          ; Размер файла.
                dw      0,0             ; 0
                dd      bfoffbits       ; Адрес начала. BMP_data.
; Информационный заголовок
BMP_info_header dd      bi_size         ; Размер BMP_info_header.
                dd      320             ; Ширина.
                dd      200             ; Высота.
                dw      1               ; Число цветовых плоскостей.
                dw      8               ; Число битов на пиксел.
                dd      0               ; Метод сжатия данных.
                dd      320*200         ; Размер данных.
                dd      0B13h           ; Разрешение по X (пиксел на метр).
                dd      0B13h           ; Разрешение по Y (пиксел на метр).
                dd      0               ; Число используемых цветов (0 - все).
                dd      0               ; Число важных цветов (0 - все).
bi_size = $-BMP_info_header             ; Размер BMP_info_header.
BMP_header_length = $-BMP_header        ; Размер обоих заголовков.
bfoffbits = $-BMP_file_header+256*4     ; Размер заголовков + размер палитры.
```

```
bfsize = $-BMP_file_header+256*4+320*200   ; Размер заголовков +
                                           ; размер палитры + размер данных.
length_of_program = $-start
        end     start
```

В этом примере, достаточно сложном из-за необходимости избегать всех случаев повторного вызова прерываний DOS и BIOS, добавилась еще одна мера предосторожности - сохранение состояния EMS-памяти перед работой с ней и восстановление в исходное состояние. Действительно, если наш резидент активизируется в тот момент, когда какая-то программа работает с EMS, и не выполнит это требование, программа будет читать/писать уже не в свой EMS-страницы, а в наши. Аналогичные предосторожности следует предпринимать всякий раз, когда вызываются функции, затрагивающие какие-нибудь глобальные структуры данных. Например: функции поиска файлов используют буфер DTA, адрес которого надо сохранить (функция DOS 2Fh), затем создать собственный (функция DOS 1Ah) и в конце восстановить DTA прерванного процесса по сохраненному адресу (функция 1Ah). Таким образом надо сохранять/восстанавливать состояние адресной линии A20 (функции XMS 07h и 03h), если резидентная программа хранит часть своих данных или кода в области HMA, сохранять состояние драйвера мыши (INT 33h, функции 16h и 17h), сохранять информацию о последней ошибке DOS (функции DOS 59h и 5DOAh), и так с каждым ресурсом, который затрагивает резидентная программа.

Писать полноценные резидентные программы в DOS очень сложно, но, если не выходить за рамки реального режима, это самое эффективное средство управления системой и реализации всего, что только можно сделать в DOS.

5.9.4. Полурезидентные программы

Полурезидентные программы - это программы, которые загружают и Выполняют другую программу, оставаясь при этом в памяти, а затем, после того как загруженная программа заканчивается, они трже заканчиваются обычным образом. Полурезидентная программа может содержать обработчики прерываний, которые будут действовать все время, пока работает загруженная из-под нее обычная программа. Так что, с точки зрения этой дочерней программы, полурезидентная программа функционирует как обычная резидентная. Эти программы удобно использовать для внесения изменений и дополнений в существующие программы, если нельзя внести исправления прямо в их исполняемый код. Так создаются загрузчики для игр, которые хранят свой код в зашифрованном или упакованном виде. Такой загрузчик может отслеживать определенные комбинации клавиш и обманывать игру, добавляя игроку те или иные ресурсы, или, например, находить код проверки пароля и выключать его.

В качестве примера напишем простой загрузчик для игры «Tie Fighter», который устранит ввод пароля, требующийся при каждом запуске игры. Разумеется, это условный пример, поскольку игра никак не шифрует свои файлы, и того же эффекта можно было достигнуть, изменив всего два байта в файле front.ovl. Единственное преимущество нашего загрузчика будет заключаться в том, что он подходит для всех версий игры (от X-Wing до Tie Fighter: Defender of the Empire).

```
; tieload.asm
; Пример полурезидентной программы - загрузчик, устраняющий проверку пароля
; для игр компании Lucasarts:
; X-Wing, X-Wing: Imperial Pursuit, B-Wing,
; Tie Fighter, Tie Fighter: Defender of the Empire
;
        .model  tiny
        .code
        .386                            ; Для команды LSS.
        org     100h                    ; COM-программа.
start:
; Освободить память после конца программы (+ стек).
        mov     sp,length_of_program    ; Перенести стек.
        mov     ah,4Ah                  ; Функция DOS 4Ah.
        mov     bx,par_length           ; Размер в параграфах.
        int     21h                     ; Изменить размер выделенной памяти.

; Заполнить поля EPB, содержащие сегментные адреса.
        mov     ax,cs
        mov     word ptr EPB+4,ax
        mov     word ptr EPB+8,ax
        mov     word ptr EPB+0Ch,ax

; Загрузить программу без выполнения.
        mov     bx,offset EPB           ; ES:BX - EPB
        mov     dx,offset filename1     ; DS:DX - имя файла (TIE.EXE).
        mov     ax,4B01h                ; Функция DOS 4B01h.
        int     21h                     ; Загрузить без выполнения.
        jnc     program_loaded          ; Если TIE.EXE не найден,
        mov     byte ptr XWING,1        ; установить флаг для find_passwd
        mov     ax,4B01h
        mov     dx,offset filename2     ; и попробовать BWING.EXE.
        int     21h
        jnc     program_loaded          ; Если он не найден,
        mov     ax,4B01h
        mov     dx,offset filename3     ; попробовать XWING.EXE.
        int     21h
        jc      error_exit              ; Если и он не найден (или не загружается
                                        ; по какой-нибудь другой причине) -
                                        ; выйти с сообщением об ошибке.
program_loaded:
; Процедура проверки пароля не находится непосредственно в исполняемом файле
; tie.exe, bwing.exe или xwing.exe, а подгружается позже из оверлея front.ovl,
; bfront.ovl или frontend.ovl соответственно. Найти команды, выполняющие чтение
; из этого оверлея, и установить на них наш обработчик find_passwd.
        cld
        push    cs
        pop     ax
        add     ax,par_length
        mov     ds,ax
```

```
        xor     si,si   ; DS:SI - первый параграф после конца нашей программы
                        ; (то есть начало области, в которую была загружена
                        ; модифицируемая программа).
        mov     di,offset read_file_code ; ES:DI - код для сравнения.
        mov     cx,rf_code_l            ; CX - его длина.
        call    find_string             ; Поиск кода.
        jc      error_exit2             ; Если он не найден - выйти
                                        ; с сообщением об ошибке.
; Заменить 6 байт из найденного кода командами call find_passwd и пор.
        mov     byte ptr [si], 9Ah      ; CALL (дальний).
        mov     word ptr [si+1], offset find_passwd
        mov     word ptr [si+3], cs
        mov     byte ptr [si+5], 90h    ; NOP.

; Запустить загруженную программу.
; Надо записать правильные начальные значения в регистры для EXE-программы
; и заполнить некоторые поля ее PSP.
        mov     ah,51h                  ; Функция DOS 51h.
        int     21h                     ; BX = PSP-сегмент загруженной программы.
        mov     ds,bx                   ; Поместить его в DS
        mov     es,bx                   ; и ES. Заполнить также поля PSP:
        mov     word ptr ds:[0Ah],offset exit_without_msg
        mov     word ptr. ds:[0Ch],cs   ; "адрес возврата"
        mov     word ptr ds:[16h],cs    ; и "адрес PSP предка".
        lss     sp,dword ptr cs:EPB_SSSP ; Загрузить SS:SP
        jmp     dword ptr cs:EPB_CSIP   ; и передать управление на
                                        ; точку входа программы.

XWING           db      0           1/0: тип защиты X-wing/Tie-fighter

EPB             dw      0           Запускаемый файл получает среду DOS от
                                    ;tieload.com,
                dw      0080h,? ; и командную строку,
                dw      005Ch,? ; и первый FCB,
                dw      006Ch,? ; и второй FCB.
EPB_SSSP        dd      ?       ; Начальный SS:SP - заполняется DOS.
EPB_CSIP        dd      ?       ; Начальный CS:IP - заполняется DOS.

filename1       db      "tie.exe",0     ; Сначала пробуем запустить этот файл,
filename2       db      "bwing.exe",0   ; потом этот,
filename3       db      "xwing.exe",0   ; а затем этот.

; Сообщения об ошибках.
error_msg       db      "Ошибка: не найден ни один из файлов TIE.EXE, "
                db      "BWING.EXE, XWING.EXE",0Dh,0Ah,'$'
error_msg2      db      "Ошибка: участок кода не найден",0Dh,0Ah,'$'

; Команды, выполняющие чтение оверлейного файла в tie.exe/bwing.exe/xwing.exe:
read_file_code:
                db      33h,0D2h                ; xor dx,dx
                db      0B4h,3Fh                ; mov ah,3Fh
                db      0CDh,21h                ; int 21h
```

```
                db      72h             ; jz
                                        ;(на разный адрес в xwing и tie)
rf_code_l = $-read_file_code

; Команды, вызывающие процедуру проверки пароля.
; Аналогичный набор команд встречается и в других местах, поэтому find_passwd
; будет выполнять дополнительные проверки.
passwd_code:
                db      89h,46h,0FCh        mov [bp-4],ax
                db      89h,56h,0FEh        mov [bp-2],dx
                db      52h                 push dx
                db      50h                 push ax
                db      9Ah                 call far
passwd_l = $-passwd_code

error_exit:
        mov     dx,offset error_msg     ; Вывод сообщения об ошибке 1.
        jmp     short exit_with_msg
error_exit2:
        mov     dx,offset error_msg2    ; Вывод сообщения об ошибке 2.
exit_with_msg:
        mov     ah,9                    ; Функция DOS 09h:
        int     21h                     ; вывести строку на экран.
exit_without_msg:                       ; Сюда также передается управление после
                                        ; завершения загруженной программы (этот адрес был
                                        ; вписан в поле PSP "адрес возврата").
        mov     ah,4Ch                  ; Функция DOS 4Ch:
        int     21h                     ; конец программы.

; Эту процедуру вызывает программа tie.exe/bwing.exe/xwing.exe каждый раз, когда
; она выполняет чтение из оверлейного файла.
find_passwd     proc    far
; Выполнить три команды, которые мы заменили на call find_passwd.
        xor     dx,dx
        mov     ah,3Fh                  ; Функция DOS 3Fh:
        int     21h                     ; чтение из файла или устройства.

deactivation_point:                     ; По этому адресу мы запишем код команды RETF,
                                        ; когда наша задача будет выполнена.
        pushf                           ; Сохраним флаги
        push    ds                      ; и регистры.
        push    es
        pusha

        push    cs
        pop     es
        mov     si,dx                     DS:DX - начало только что прочитанного участка
                                        ; оверлейного файла.
        mov     di,offset passwd_code   ; ES:DI - код для сравнения.
        dec     si                        Очень скоро мы его увеличим обратно.
search_for_pwd:                         В этом цикле найденные вхождения эталонного кода
                                        ; проверяются на точное соответствие коду проверки
                                        ; пароля.
```

```
        inc     si                  ; Процедура find_string возвращает DS:SI
                                    ; указывающим на начало найденного кода - чтобы
                                    ; искать дальше, надо увеличить SI хотя бы на 1.
        mov     cx,passwd_l         ; Длина эталонного кода.
        call    find_string         ; Поиск его в памяти.
        jc      pwd_not_found       ; Если он не найден - выйти.
; find_string нашла очередное вхождение нашего эталонного кода вызова
; процедуры - проверим, точно ли это вызов процедуры проверки пароля.
        cmp     byte ptr [si+10],00h    ; Этот байт должен быть 00.
        jne     search_for_pwd
        cmp     byte ptr cs:XWING,1     ; В случае X-wing/B-wing
        jne     check_for_tie
        cmp     word ptr [si+53],0774h  ; команда je должна быть здесь,
        jne     search_for_pwd
        jmp     short pwd_found
check_for_tie:                          ; а в случае Tie Fighter -
        cmp     word ptr [si+42],0774h  ; здесь.
        jne     search_for_pwd
pwd_found:      ; Итак, вызов процедуры проверки пароля найден - отключить его
        mov     word ptr ds:[si+8],9090h ; NOP NOP
        mov     word ptr ds:[si+10],9090h ; NOP NOP
        mov     byte ptr ds:[si+12],90h   ; NOP
                ; и дезактивизировать нашу процедуру find_passwd.
        mov     byte ptr cs:deactivation_point,0CBh ; RETF

pwd_not_found:
        popa                            ; Восстановить регистры
        pop     es
        pop     ds
        popf                            ; и флаги
        ret                             ; и вернуть управление в программу.
find_passwd     endp

; Процедура find_string.
; Выполняет поиск строки от заданного адреса до конца всей общей памяти.
; Вход: ES:DI - адрес эталонной строки.
;       CX - ее длина
;       DS:SI - адрес, с которого начинать поиск
; Выход: CF = 1, если строка не найдена,
; иначе: CF = 0 и DS:SI - адрес, с которого начинается найденная строка.

find_string     proc    near
        push    ax
        push    bx
        push    dx                      ; Сохранить регистры.

do_cmp: mov     dx,1000h                ; Поиск блоками по 1000h (4096 байт).
cmp_loop:
        push    di
        push    si
        push    cx
```

```
        repe    cmpsb                   ; Сравнить DS:SI со строкой.
        pop     cx
        pop     si
        pop     di
        je      found_code              ; Если совпадение - выйти с CF = 0.
        inc     si                      ; Иначе - увеличить DS:SI на 1,
        dec     dx                      ; уменьшить счетчик в DX
        jne     cmp_loop                ; и, если он не ноль, продолжить.
; Пройден очередной 4-килобайтный блок.
        sub     si,1000h                ; Уменьшить SI на 1000h
        mov     ax,ds
        inc     ah                      ; и увеличить DS на 1.
        mov     ds,ax
        cmp     ax,9000h                ; Если мы добрались до
        jb      do_cmp                  ; сегментного адреса 9000h -

        pop     dx                      ; восстановить регистры.
        pop     bx
        pop     ax
        stc                             ; Установить CF = 1
        ret                             ; и выйти.
; Сюда передается управление, если строка найдена.
found_code:
        pop     dx                      ; Восстановить регистры.
        pop     bx
        pop     ax
        clc                             ; Установить CF = 0
        ret                             ; и выйти.
find_string     endp

end_of_program:
length_of_program = $-start+100h+100h   ; Длина программы в байтах.
par_length = length_of_program + 0Fh
par_length = par_length/16              ; Длина программы в параграфах.
        end     start
```

5.9.5. *Взаимодействие между процессами*

Даже несмотря на то, что DOS является однозадачной операционной системой, в ней одновременно могут быть задействованы несколько процессов. Это означает, что сама система не предоставляет никаких специальных возможностей для их одновременного выполнения, кроме возможности оставлять программы резидентными в памяти. Следовательно, чтобы организовать общую память для нескольких процессов, надо загрузить пассивную резидентную программу, которая будет поддерживать функции выделения блока памяти (возвращающая идентификатор), определения адреса блока (по его идентификатору) и освобождения блока - приблизительно так же, как работают драйверы EMS или XMS.

Чтобы реализовать многозадачность, придется запустить активную резидентную программу, которая перехватит прерывание IRQ0 и по каждому такту системного таймера будет по очереди отбирать управление от каждого из запущенных

процессов и передавать следующему. Практически никто не реализует полноценную многозадачность в DOS, когда каждый процесс имеет собственную память и не может обращаться к памяти другого процесса, - для этого существует защищенный режим, но встречаются довольно простые реализации для облегченного варианта многозадачности - переключение нитей.

Нить - это процесс, который использует тот же код и те же данные, что и остальные такие же процессы в системе, но отличается от них содержимым стека и регистров. Тогда резидентная программа (диспетчер) по каждому прерыванию таймера будет сохранять регистры прерванной нити в ее структуру, считывать регистры следующей нити в очереди и возвращать управление, а структуры и стеки всех нитей будут храниться в какой-нибудь специально выделенной общедоступной области памяти. Указанная программа также должна поддерживать несколько вызовов при помощи какого-нибудь программного прерывания - создание нити, удаление нити и, например, передача управления следующей нити, пока текущая нить находится в состоянии ожидания.

Эта простота оборачивается сложностью написания самих нитей. Так как все они используют общий код, абсолютно все в коде нити должно быть повторно входимым. Кроме того, нити создают множество проблем, связанных с синхронизацией, приводящих к тому, что либо в коде всех нитей, либо в основном резиденте придется реализовывать семафоры, очереди, сигналы, барьеры и все остальные структуры, которые встречаются в реальных пакетах для работы с нитями.

Попробуем создать элементарный прототип такой многозадачности в DOS (всего с двумя нитями) и посмотрим, со сколькими проблемами придется столкнуться.

```
; scrsvr.asm
; Пример простой задачи, реализующей нитевую многозадачность в DOS.
; Изображает на экране две змейки, двигающиеся случайным образом,
; каждой из которых управляет своя нить.
;
; Передача управления между нитями не работает в окне DOS (Windows 95).

        .model  tiny
        .code
        .386                    ; ГСЧ использует 32-битные регистры.
        org     100h            ; COM-программа.
start:
        mov .   ax,13h          ; Видеорежим 13h:
        int     10h             ; 320x200x256.

        call    init_threads    ; Инициализировать наш диспетчер.
; С этого места и до вызова shutdown_threads исполняются две нити с одним и тем
; же кодом и данными, но с разными регистрами и стеками
; (в реальной системе здесь был бы вызов fork или аналогичной функции).

        mov     bx,1            ; Цвет (синий).
        push    bp
        mov     bp,sp           ; Поместить все локальные переменные в стек,
                                ; чтобы обеспечить повторную входимость.
        push    1               ; Добавка к X на каждом шаге.
x_inc       '   equ word ptr [bp-2]
```

```
        push    0               ; Добавка к Y на каждом шаге.
y_inc           equ word ptr [bp-4]
        push    128-4           ; Относительный адрес головы буфера line_coords.
coords_head     equ word ptr [bp-6]
        push    0               ; Относительный адрес хвоста буфера line_coords.
coords_tail     equ word ptr [bp-8]
        sub     sp,64*2         ; line_coords - кольцевой буфер координат точек.
        mov     di,sp
        mov     cx,64
        mov     ax,10           ; Заполнить его координатами (10, 10).
        push    ds
        pop     es
        rep     stosw
line_coords     equ word ptr [bp-(64*2)-8]

        push    0A000h
        pop     es                      ; ES - адрес видеопамяти.

main_loop:                              ; Основной цикл.
        call    display_line            ; Изобразить текущее состояние змейки.

; Изменить направление движения случайным образом.
        push    bx
        mov     ebx,50                  ; Вероятность смены направления 2/50.
        call    z_random                ; Получить случайное число от 0 до 49.
        mov     ax,word ptr x_inc
        mov     bx,word ptr y_inc
        test    dx,dx                     Если это число - 0,
        jz      rot_right                 повернем направо,
        dec     dx                      а если 1 -
        jnz     exit_rot                  налево.
                                        ; Повороты
        neg     ax      /                 налево на 90 градусов.
        xchg    ax,bx                   dY = -dX, dX = dY.
        jmp     short exit_rot
rot_right:
        neg     bx                      ; Направо на 90 градусов.
        xchg    ax,bx                   ; dY = dX, dX = dY.
exit_rot:
        mov     word ptr x_inc,ax       ; Записать новые значения инкрементов.
        mov     word ptr y_inc,bx
        pop     bx                              Восстановить цвет в BX.

; Перемещение змейки на одну позицию вперед.
        mov     di,word ptr coords_head         DI - адрес головы.
        mov     cx,word ptr line_coords[di]     CX - строка.
        mov     dx,word ptr line_coords[di+2]   DX - столбец.
        add     cx,word ptr y_inc               Добавить инкременты.
        add     dx,word ptr x_lnc
        add     di,4                            DI - следующая точка в буфере.
        and     di,127                          Если DI > 128, DI = DI - 128.
        mov     word ptr coords_head,di         Теперь голова здесь.
```

```
        mov     word ptr line_coords[di],cx     ; Записать ее координаты.
        mov     word ptr line_coords[di+2],dx
        mov     di,word ptr coords_tail         ; Прочитать адрес хвоста.
        add     di,4                            ; Переместить его на одну
        and     di,127                          ; позицию вперед
        mov     word ptr coords_tail,di         ; и записать на место.

; Пауза.
; Из-за особенностей нашего диспетчера (см. ниже) мы не можем пользоваться
; прерыванием BIOS для паузы, поэтому сделаем просто пустой цикл. Длину цикла
; придется изменить в зависимости от скорости процессора.
        mov     cx,-1
        loop    $                       ; 65 535 команд loop.
        mov     cx,-1
        loop    $
        mov     cx,-1
        loop    $

        mov     ah,1
        int     16h                     ; Если ни одна из клавиш не была нажата,
        jz      main_loop               ; продолжить основной цикл.
        mov     ah,0                    ; Иначе - прочитать клавишу.
        int     16h
        leave                           ; Освободить стек от локальных переменных.
        call    shutdown_threads        ; Выключить многозадачность.
; С этого момента у нас снова только один процесс.
        mov     ax,3                    ; Видеорежим 3:
        int     10h                     ; 80x24.
        int     20h                     ; Конец программы.

; Процедура вывода точки на экран в режиме 13h.
; CX = строка, DX = столбец, BL = цвет, ES = 0A000h
putpixel        proc    near
        push    di
        lea     ecx,[ecx*4+ecx]         ; CX = строка x 5.
        shl     cx,6                    ; CX = строка x 5 x 64 = строка x 320.
        add     dx,cx                   ; DX = строка x 320 + столбец = адрес.
        mov     di,dx
        mov     al,bl
        stosb                           ; Записать байт в видеопамять.
        pop     di
        ret
putpixel        endp

; Процедура display_line.
; Выводит на экран нашу змейку по координатам из кольцевого буфера line_coords.
display_line    proc near
        mov     di,word ptr coords_tail ; Начать вывод с хвоста.
continue_line_display:
        cmp     di,word ptr coords_head ; Если DI равен адресу головы,
        je      line_displayed          ; вывод закончился.
```

```
        call    display_point           ; Иначе - вывести точку на экран.
        add     di,4                    ; Установить DI на следующую точку.
        and     di,127
        jmp     short continue_line_display     ; И так далее.
line_displayed:
        call    display_point
        mov     di,word ptr coords_tail ; Вывести точку в хвосте
        push    bx
        mov     bx,0                    ; нулевым цветом,
        call    display_point           ; то есть стереть.
        pop     bx
        ret
display_line    endp

; Процедура display_point.
; Выводит точку из буфера line_coords с индексом DI.
display_point   proc    near
        mov     cx,word ptr line_coords[di]     ; Строка.
        mov     dx,word ptr line_coords[di+2]   ; Столбец.
        call    putpixel                        ; Вывод точки.
        ret
display_point   endp

; Процедура z_random.
; Стандартный конгруэнтный генератор случайных чисел (неоптимизированный).
; Вход: EBX - максимальное число.
; Выход: EDX - число от 0 до EBX-1.
z_random:
        push    ebx
        cmp     byte ptr zr_init_flag,0 ; Если еще не вызывали,
        je      zr_init                 ; инициализироваться.
        mov     eax,zr_prev_rand        ; Иначе - умножить предыдущее
zr_cont:
        mul     rnd_number              ; на множитель
        div     rnd_number2             ; и разделить на делитель.
        mov     zr_prev_rand,edx        ; Остаток -от деления - новое число.
        pop     ebx
        mov     eax,edx
        xor     edx,edx
        div     ebx                     ; Разделить его на максимальное
        ret                             ; и вернуть остаток в EDX.
zr_init:
        push    0040h                   ; Инициализация генератора.
        pop     fs                      ; 0040h:006Ch -
        mov     eax,fs:[006Ch]          ; счетчик прерываний таймера BIOS,
        mov     zr_prev_rand,eax        ; он и будет первым случайным числом.
        mov     byte ptr zr_init_flag,1
        jmp     zr_cont
rnd_number      dd      16807           ; Множитель.
rnd_number2     dd      2147483647      ; Делитель.
```

```
zr_init_flag    db      0               ; Флаг инициализации генератора.
zr prev rand    dd      0               ; Предыдущее случайное число.

; Здесь начинается код диспетчера, обеспечивающего многозадачность.

; Структура данных, в которой мы храним регистры для каждой нити.
thread_struc struc
_ax             dw      ?
_bx             dw      ?
_cx             dw      ?
_dx             dw      ?
_si             dw      ?
_di             dw      ?
_bp             dw      ?
_sp             dw      ?
_ip             dw      ?
_flags          dw      ?
thre'acLstruc           ends

; Процедура init_threads.
; Инициализирует обработчик прерывания 08h и заполняет структуры, описывающие
; обе нити, .
init_threads    proc    near
        pushf
        pusha
        push    es
        mov     ax,3508h                ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Определить адрес обработчика.
        mov     word ptr old_int08h,bx .  ; Сохранить его.
        mov     word ptr old_int08h+2,es
        mov     ax,2508h                ; AH = 25h, AL = номер прерывания.
        mov     dx,offset int08h_handler  ; Установить наш.
        int     21h
        pop     es
        popa            ; Теперь регистры те же, что и при вызове процедуры.
        popf

        mov     thread1._ax,ax          ; Заполнить структуры
        mov     thread2._ax,ax          ; thread1 и thread2,
        mov     thread1._bx,bx          ; в которых хранится содержимое
        mov     thread2._bx,bx          ; всех регистров (кроме сегментных -
        mov     thread1._cx,cx          ; они в этом примере не изменяются).
        mov     thread2._cx,cx
        mov     thread1._dx,dx
        mov     thread2._dx,dx
        mov     thread1._si,si
        mov     thread2._si,si
        mov     thread1._di,di
        mov     thread2._di,di
        mov     thread1._bp,bp
        mov     thread2._bp,bp
```

```
        mov     thread1._sp,offset thread1_stack+512
        mov     thread2._sp,offset thread2_stack+512
        pop     ax                      ; Адрес возврата (теперь стек пуст).
        mov     thread1._ip,ax
        mov     thread2._ip,ax
        pushf
        pop     ax                      ; Флаги.
        mov     thread1._flags,ax
        mov     thread2._flags,ax
        mov     sp,thread1._sp          ; Установить стек нити 1
        jmp     word ptr thread1._ip    ; и передать ей управление.
init_threads    endp

current_thread  db      1               ; Номер текущей нити.

; Обработчик прерывания INT08h (IRQ0)
; переключает нити
int08h_handler proc far
        pushf                           ; Сначала вызвать старый обработчик.
        db      9Ah                     ; Код команды call far.
old_int08h      dd      0               ; Адрес старого обработчика.
; Определить, произошло ли прерывание в момент исполнения нашей нити или
; какого-то обработчика другого прерывания. Это важно, так как мы не собираемся
; возвращать управление чужой программе, по крайней мере сейчас.
; Вот почему нельзя пользоваться прерываниями для задержек в наших нитях
; и программа не работает в окне DOS (Windows 95).
        mov     save_di,bp              ; Сохранить BP.
        mov     bp,sp
        push    ax
        push    bx
        pushf
        mov     ax,word ptr [bp+2]      ; Прочитать сегментную часть
        mov     bx,cs                   ; обратного адреса.
        cmp     ax,bx                   ; Сравнить ее с CS.
        jne     called_far              ; Если они не совпадают - выйти.
        popf
        pop     bx                      ; Иначе - восстановить регистры.
        pop     ax
        mov     bp,save_di

        mov     save_di,di                Сохранить DI, SI
        mov     save_si,si
        pushf                             и флаги.
; Определить, с какой нити на какую надо передать управление.
        cmp     byte ptr current_thread,1 ; Если с первой,
        je      thread1_to_thread2        ; перейти на thread1_to_thread2.
        mov     byte ptr current_thread,1 ; Если с 2 на 1, записать в номер 1
        mov     si,offset thread1         ; и установить SI и DI
        mov     di,offset thread2         ; на соответствующие структуры.
        jmp     short order_selected
```

```
thread1_to_thread2:                     ; Если с 1 на 2,
        mov     byte ptr current_thread,2 ; записать в номер нити 2
        mov     si,offset thread2       ; и установить SI и DI.
        mov     di,offset thread1
order_selected:

; Записать все текущие регистры в структуру по адресу [DI]
; и загрузить все регистры из структуры по адресу [SI].
; начать с SI и DI:
        mov     ax,[si]._si     ; Для MASM все выражения [reg]._reg надо
        push    save_si         ; заменить (thread_struc ptr [reg])._reg.
        pop     [di]._si
        mov     save_si,ax
        mov     ax,[si]._di
        push    save_di
        pop     [di]._di
        mov     save_di,ax
; Теперь все основные регистры.
        mov     [di._ax],ax
        mov     ax,[si._ax]
        mov     [di._bx],bx
        mov     bx,[si._bx]
        mov     [di._cx],cx
        mov     cx,[si._cx]
        mov     [di._dx],dx
        mov     dx,[si._dx]
        mov     [di._bp],bp
        mov     bp,[si._bp]
; Флаги.
        pop     [di._flags]
        push    [si._flags]
        popf
; Адрес возврата.
        pop     [di._ip]        ; Адрес возврата из стека.
        add     sp,4            ; CS и флаги из стека - теперь он пуст.
; Переключить стеки.
        mov     [di._sp],sp
        mov     sp,[si._sp]
        push    [si._ip]        ; Адрес возврата в стек (уже новый).
        mov     di,save_di      ; Загрузить DI и SI
        mov     si,save_si
        retn                    ; и перейти по адресу в стеке.
; Управление переходит сюда, если прерывание произошло в чужом коде.
called_far:
        popf                    ; Восстановить регистры
        pop     bx
        pop     ax
        mov     bp,save_di
        iret                    ; и завершить обработчик.
int08h_handler  endp
```

```
save_di         dw      ?               ; -Переменные для временного хранения
save_si         dw      ?               ; регистров.

; Процедура shutdown_threads.
; Выключает диспетчер.
shutdown_threads proc near
        mov     ax,2508h        ; Достаточно просто восстановить прерывание.
        lds     dx,dword ptr old_int08h
        int     21h
        ret
shutdown_threads endp

; Структура, описывающая первую нить.
thread1 thread_struc <>
; И вторую.
thread2 thread_struc <>
; Стек первой нити.
thread1_stack db 512 dup(?)
; И второй.
thread2_stack db 512 dup(?)
end     start
```

Как мы видим, этот пример не может работать в Windows 95 и в некоторых других случаях, когда DOS расширяют до более совершенной операционной системы. Фактически в этом примере мы именно этим и занимались — реализовывали фрагмент операционной системы, который отсутствует в DOS.

Действительно, применяя механизм обработчиков прерываний, можно создать операционную систему для реального режима, аналогичную DOS, но очень быстро окажется, что для этого нужно общаться напрямую с аппаратным обеспечением компьютера, то есть использовать порты ввода-вывода.

5.10. Программирование на уровне портов ввода-вывода

Как видно из предыдущей главы, использование системных функций DOS и прерываний BIOS может быть небезопасным из-за отсутствия в них повторной входимости. Теперь самое время спуститься на следующий уровень и научиться работе с устройствами компьютера напрямую, через порты ввода-вывода, как это и делают системные функции. Кроме того, многие возможности компьютера могут быть реализованы только программированием на уровне портов.

5.10.1. Клавиатура

Контроллеру клавиатуры соответствуют порты с номерами от 60h до 6Fh, хотя для всех стандартных операций достаточно портов 60h и 61h.

64h для чтения - регистр состояния клавиатуры, возвращает следующий байт:

бит 7: ошибка четности при передаче данных с клавиатуры

бит 6: тайм-аут при приеме

бит 5: тайм-аут при передаче

бит 4: клавиатура закрыта ключом
бит 3: данные, записанные в регистр ввода, - команда
бит 2: самотестирование закончено
бит 1: в буфере ввода есть данные (для контроллера клавиатуры)
бит 0: в буфере вывода есть данные (для компьютера)

При записи в этот порт он играет роль дополнительного регистра управления клавиатурой, но его команды сильно различаются для разных плат и разных BIOS, и мы не будем его подробно рассматривать.

61h для чтения и записи - регистр управления клавиатурой. Если в старший бит этого порта записать значение 1, клавиатура будет заблокирована, если 0 - разблокирована. Другие биты этого порта менять нельзя, так как они управляют иными устройствами (в частности динамиком). Чтобы изменить состояние клавиатуры, надо считать байт из порта, изменить бит 7 и снова записать в порт 61h этот байт.

60h для чтения — порт данных клавиатуры. При чтении из него можно получить скан-код последней нажатой клавиши (см. приложение 1) - именно так лучше всего реализовывать резидентные программы, перехватывающие прерывание IRQ1, потому что по этому коду можно определять момент нажатия и отпускания любой клавиши, включая такие клавиши, как **Shift, Ctrl,** Alt или даже **Pause** (скан-код отпускания клавиши равен скан-коду нажатия плюс 80h):

```
int09h_handler:
        in      al,60h          ; Прочитать скан-код клавиши.
        cmp     al,hot_key      ; Если это наша "горячая" клавиша,
        jne     not_our_key     ; перейти к нашему обработчику.
        [...]                   ; Наши действия здесь.
not_our_key:
        jmp     old_int09h      ; Вызов старого обработчика.
```

Мы пока не можем завершить обработчик просто командой IRET, потому что, во-первых, обработчик аппаратного прерывания клавиатуры должен установить бит 7 порта 61h, а затем вернуть его в исходное состояние, например так:

```
        in      al,61h
        push    ax
        or      al,80h
        out     61h,al
        pop     ax
        out     61h,al
```

Во-вторых, он должен сообщить контроллеру прерываний, что обработка аппаратного прерывания закончилась командами

```
        mov     al,20h
        out     20h,al
```

60h для записи - регистр управления клавиатурой. Байт, записанный в этот порт (если бит 1 в порту 64h равен 0), интерпретируется как команда. Некоторые

команды состоят из более чем одного байта - тогда следует дождаться обнуления этого бита еще раз перед тем, как посылать следующий байт. Перечислим наиболее стандартные команды.

Команда OEDh 0?h - изменить состояние светодиодов клавиатуры. Второй байт этой команды определяет новое состояние:

бит 0: состояние **Scroll Lock** (1 - включена, 0 - выключена)
бит 1: состояние Num **Lock**
бит 2: состояние **Caps Lock**

При этом состояние переключателей, которое хранит BIOS в байтах состояния клавиатуры, не изменяется, и при первой возможности обработчик прерывания клавиатуры BIOS восстановит состояние светодиодов.

Команда OEEh - эхо-запрос. Клавиатура отвечает скан-кодом 0EEh.

Команда 0F3h ??h - установить параметры режима автоповтора:

бит 7 второго байта команды: О
биты 6-5: устанавливают паузу перед началом автоповтора:
00b = 250ms, 01b = 500ms, 10b = 750ms, 11b = 1000ms
биты 4-0: устанавливают скорость автоповтора (символов в секунду):

00000b = 30,0	01111b = 8,0
00010b = 24,0	10010b = 6,0
00100b = 20,0	10100b = 5,0
00111b = 16,0	10111b = 4,0
01000b = 15,0	11010b = 3,0
01010b = 12,0	11111b = 2,0
01100b = 10,0	

Все промежуточные значения также имеют смысл и соответствуют промежуточным скоростям, например OOOOlb = 26,7.

Команда OF4h - включить клавиатуру.
Команда 0F5h - выключить клавиатуру.
Команда 0F6h - установить параметры по умолчанию.
Команда 0FEh — послать последний скан-код еще раз.
Команда 0FFh - выполнить самотестирование.

Клавиатура отвечает на все команды, кроме OEEh и OFEh, скан-кодом OFAh (подтверждение), который поглощается стандартным обработчиком BIOS, поэтому, если мы не замещаем его полностью, об обработке OFAh можно не беспокоиться.

В качестве примера работы с клавиатурой напрямую рассмотрим простую программу, выполняющую переключение светодиодов.

```
; mig.asm                    •                    • ,
; Циклически переключает светодиоды клавиатуры.

        .model  tiny
        .code
org     100h                        ; COM-программа.
```

```
start           proc    near
        mov     ah, 2               ; Функция 02 прерывания 1Ah:
        int     1Ah .               ; получить текущее время.
        mov     ch,dh               ; Сохранить текущую секунду в CH.
        mov     cl,0100b            ; CL = состояние светодиодов клавиатуры.

main_loop:
        call    change_LEDs         ; Установить светодиоды в соответствии с CL.
        shl     cl,1                ; Следующий светодиод.
        test    cl,1000b            ; Если единица вышла в бит 3,
        jz      continue
        mov     cl,0001b            ; вернуть ее в бит 0.
continue:
        mov     ah,1                ; Проверить, не была ли нажата клавиша.
        int     16h
        jnz     exit_loop           ; Если да - выйти из программы.
        push    cx
        mov     ah, 2               ; Функция 02 прерывания 1Ah.
        int     1Ah                 ; Получить текущее время.
        pop     cx
        cmp     ch,dh               ; Сравнить текущую секунду в DH с CH.
        mov     ch,dh               ; Скопировать ее в любом случае.
        je      continue            ; Если это была та же самая секунда -
                                    ; не переключать светодиоды.
        jmp     short main_loop     ; Иначе - переключить светодиоды.

exit_loop:
        mov     ah,0                ; Выход из цикла - была нажата клавиша.
        int     16h                 ; Считать ее
        ret                         ; и завершить программу.
start           endp

; Процедура change_LEDs.
; Устанавливает состояние светодиодов клавиатуры в соответствии с числом в CL.
change_LEDs     proc    near
        call    wait_KBin           ; Ожидание возможности посылки команды.
        mov     al,0EDh
        out     60h,al              ; Команда клавиатуры EDh.
        call    wait_KBin           ; Ожидание возможности посылки команды.
        mov     al,cl
        out     60h,al              ; Новое состояние светодиодов.
        ret
change_LEDs     endp

; Процедура wait_KBin.
; Ожидание возможности ввода команды для клавиатуры.
wait_KBin       proc    near
        in      al,64h              ; Прочитать слово состояния.
        test    al,0010b            ; Бит 1 равен 1?
        jnz     wait_KBin           ; Если нет - ждать.
        ret                         ; Если да - выйти.
wait_KBin       endp
        end     start
```

5.10.2. Последовательный порт

Каждый из последовательных портов обменивается данными с процессором через набор портов ввода-вывода: COM1 = 03F8h - 03FFh, COM2 = 02F8h - 02FFh, COM3 = 03E8h - 03EFh и COM4 = 02E8h - 02EFh. Имена портов COM1 - COM4 на самом деле никак не зафиксированы. BIOS просто называет порт COM1, адрес которого (03F8h по умолчанию) записан в области данных BIOS по адресу 0040h:0000h. Точно так же порт COM2, адрес которого записан по адресу 0040h:0002h, COM3 - 0040h:0004h и COM4 - 0040h:0006h. Рассмотрим назначение портов ввода-вывода на примере 03F8h - 03FFh.

03F8h для чтения и записи - если старший бит регистра управления линией = 0, то это регистр передачи данных (THR или RBR). Передача и прием данных через последовательный порт соответствуют записи и чтению именно в этот порт.
03F8h для чтения и записи - если старший бит регистра управления линией = 1, то это младший байт делителя частоты порта.
03F9h для чтения и записи - если старший бит регистра управления линией = 0, то это регистр разрешения прерываний (IER):

бит 3: прерывание по изменению состояния модема
бит 2: прерывание по состоянию BREAK или ошибке
бит 1: прерывание, если буфер передачи пуст
бит 0: прерывание, если пришли новые данные

03F9h для чтения и записи — если старший бит регистра управления линией = 1, то это старший байт делителя частоты порта. Значение скорости порта определяется по значению делителя частоты (см. табл. 20).
03FAh для чтения - регистр идентификации прерывания. Содержит информацию о причине прерывания для обработчика:

биты 7-6: 00 - FIFO отсутствует, 11 - FIFO присутствует
бит 3: тайм-аут FIFO приемника
биты 2–1: тип произошедшего прерывания:
    11b – состояние BREAK или ошибка.
    Сбрасывается после чтения из 03FDh
    10b - пришли данные.
    Сбрасывается после чтения из 03F8h
    01b - буфер передачи пуст.
    Сбрасывается после записи в 03F8h
    00b - изменилось состояние модема.
    Сбрасывается после чтения из 03FEh
бит 0: 0, если произошло прерывание; 1, если нет

03FAh для записи - регистр управления FIFO (FCR)

биты 7-6: порог срабатывания прерывания о приеме данных:
    00b - 1 байт
    01b - 4 байта
    10b - 8 байт
    11b - 14 байт

бит 2: очистить FIFO передатчика
бит 1: очистить FIFO приемника
бит 0: включить режим работы через FIFO

03FBh для чтения и записи - регистр управления линией (LCR)
бит 7: если 1 - порты 03F8h и 03F9h работают, как делитель частоты порта
бит 6: состояние BREAK - порт непрерывно посылает нули
биты 5-3: четность:
? ? О - без четности
О О 1 - контроль на нечетность
0 1 1 - контроль на четность
1 О 1 - фиксированная четность 1
1 1 1 - фиксированная четность О
? ? 1 - программная (не аппаратная) четность
бит 2: число стоп-бит:
0-1 стоп-бит
1–2 стоп-бита для 6-, 7-, 8-битных; 1,5 стоп-бита для 5-битных слов
биты 1 – 0: длина слова:
00-5 бит
01–6 бит
10-7 бит
11–8 бит

03FCh для чтения и записи - регистр управления модемом (MCR)
бит 4: диагностика (выход COM-порта замыкается на вход)
бит 3: линия OUT2 - должна быть 1, чтобы работали прерывания
бит 2: линия OUT1 - должна быть О
бит 1: линия RTS
бит 0: линия DTR

03FDh для чтения - регистр состояния линии (LSR)
бит 6: регистр сдвига передатчика пуст
бит 5: регистр хранения передатчика пуст - можно писать в 03F8h
бит 4: обнаружено состояние BREAK (строка нулей длиннее, чем старт-бит + слово + четность + стоп-бит)
бит 3: ошибка синхронизации (получен нулевой стоп-бит)
бит 2: ошибка четности
бит 1: ошибка переполнения (пришел новый байт, хотя старый не был прочитан из 03F8h, при этом старый байт теряется)
бит 0: данные получены и готовы для чтения из 03F8h

03FEh для чтения - регистр состояния модема (MSR)
бит 7: линия DCD (несущая)
бит 6: линия RI (звонок)
бит 5: линия DSR (данные готовы)
бит 4: линия CTS (разрешение на посылку)
бит 3: изменилось состояние DCD
бит 2: изменилось состояние RI

бит 1: изменилось состояние DSR
бит 0: изменилось состояние CTS

03FFh для чтения и записи - запасной регистр. Не используется контроллером последовательного порта, любая программа может им пользоваться.

Таблица 20. Делители частоты последовательного порта

Делитель частоты	Скорость
0001h	115 200
0002h	57 600
0003h	38 400
0006h	19 200
000Ch	9 600
0010h	7 200
0018h	4 800
0020h	3 600
0030h	2 400

Итак, первое, что должна сделать программа, работающая с последовательным портом, - проинициализировать его, записав в регистр управления линией (03FBh) число 80h, делитель частоты — в порты 03F8h и 03F9h, режим — в порт 03FBh, а также указав разрешенное прерывание в порту 03F9h. Если программа вообще не пользуется прерываниями - надо записать в этот порт 0.

Перед записью данных в последовательный порт можно проверить бит 5, а перед чтением - бит 1 регистра состояния линии, но, если программа использует прерывания, эти условия выполняются автоматически. Вообще говоря, реальная серьезная работа с последовательным портом возможна только при помощи прерываний. Посмотрим, как может быть устроена такая программа на следующем примере:

```
; term2.asm '
; Минимальная терминальная программа, использующая прерывания.
; Выход - Alt-X

        .model  tiny
        .code
        .186
        org     100h            ; COM-программа.

; Следующие четыре директивы определяют, для какого последовательного порта
; скомпилирована программа (никаких проверок не выполняется - не запускайте этот
; пример, если у вас нет модема на соответствующем порту). Реальная программа
; должна определять номер порта из конфигурационного файла или из командной строки.
COM             equ 02F8h       ; Номер базового порта (COM2).
IRQ             equ 0Bh         ; Номер прерывания (INT 0Bh для IRQ3).
E_BITMASK       equ 11110111b    ; Битовая маска для разрешения IRQ3.
D_BITMASK       equ 00001000b    ; Битовая маска для запрещения IRQ3.
```

```
start:
        call    init_everything         ; Инициализация линии и модема.
main_loop:                              ; Основной цикл.
; Реальная терминальная программа в этом цикле будет выводить данные из буфера
; приема (заполняемого из обработчика прерывания) на экран, если идет обычная
; работа, в файл, если пересылается файл, или обрабатывать как-то по-другому.
; В нашем примере мы используем основной цикл для ввода символов, хотя лучше это
; делать из обработчика прерывания от клавиатуры.
        mov     ah,8                    ; Функция DOS 08h:
        int     21h                     ; чтение с ожиданием и без эха.
        test    al,al                   ; Если введен обычный символ,
        jnz     send_char               ; послать его.
        int     21h                     ; Иначе - считать расширенный ASCII-код.
        cmp     al,2Dh                  ; Если это не Alt-X,
        jne     main_loop               ; продолжить цикл.

        call    shutdown_everything     ; Иначе - восстановить все в
                                        ; исходное состояние
        ret                             ; и завершить программу.

send_char:                              ; Отправка символа в модем.
; Реальная терминальная программа должна здесь только добавлять символ в буфер
; передачи и, если этот буфер был пуст, разрешать прерывания "регистр передачи
; пуст". Просто пошлем символ напрямую в порт.
        mov     dx,COM                  ; Регистр THR.
        out     dx,al
        jmp     short main_loop

old_irq         dd      ?               ; Здесь будет храниться адрес старого
                                        ;обработчика.

; Упрощенный обработчик прерывания от последовательного порта.
irq_handler     proc    far
        pusha                           ; Сохранить регистры.
        mov     dx,COM+2                ; Прочитать регистр идентификации
        in      al,dx                   ; прерывания.
repeat_handler:
        and     ax,00000110b            ; Обнулить все биты, кроме 1 и 2,
        mov     di,ax                   ; отвечающие за 4 основные ситуации.
        call    word ptr cs:handlers[di] ; Косвенный вызов процедуры
                                        ; для обработки ситуации.
        mov     dx,COM+2                ; Еще раз прочитать регистр идентификации
        in      al,dx                   ; прерывания.
        test    al,1                    ; Если младший бит не 1,
        jz      repeat_handler          ; надо обработать еще одно прерывание.
        mov     al,20h                  ; Иначе - завершить аппаратное прерывание
        out     20h,al                  ; посылкой команды EOI (см. раздел 5.10.10).
        popa
        iret
; Таблица адресов процедур, обслуживающих разные варианты прерывания.
handlers        dw      offset line_h, offset trans_h
                dw      offset recv_h, offset modem_h
```

```
; Эта процедура вызывается при изменении состояния линии.
line_h          proc    near
        mov     dx,COM+5        ; Пока не будет прочитан LSR,
        in      al,dx           ; прерывание считается незавершившимся.
; Здесь можно проверить, что случилось, и, например, прервать связь, если
; обнаружено состояние BREAK.
        ret
line_h          endp
; Эта процедура вызывается при приеме новых данных.
recv_h          proc    near
        mov     dx,COM          ; Пока не будет прочитан RBR,
        in      al,dx           ; прерывание считается незавершившимся.
; Здесь следует поместить принятый байт в буфер приема для основной программы,
; но мы просто сразу выведем его на экран.
        int     29h             ; Вывод на экран.
        ret
recv_h          endp
; Эта процедура вызывается по окончании передачи данных.
trans_h         proc    near
; Здесь следует записать в THR следующий символ из буфера передачи и, если
; буфер после этого оказывается пустым, запретить этот тип прерывания.
        ret
trans_h         endp
; Эта процедура вызывается при изменении состояния модема.
modem_h         proc    near
        mov     dx,COM+6        ; Пока MCR не будет прочитан,
        in      al,dx           ; прерывание считается незавершившимся.
; Здесь можно определить состояние звонка и поднять трубку, определить
; потерю несущей и перезвонить, и т. д.
        ret
modem_h         endp
irq_handler     endp

; Инициализация всего, что требуется инициализировать.
init_everything proc    near
; Установка нашего обработчика прерывания.
        mov     ax,3500h+IRQ            ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Получить адрес старого обработчика
        mov     word ptr old_irq,bx     ; и сохранить в old_irq.
        mov     word ptr old_irq+2,es
        mov     ax,2500h+IRQ            ; AH = 25h, AL = номер прерывания.
        mov     dx,offset irq_handler   ; DS:DX - наш обработчик.
        int     21h                     ; Установить новый обработчик.
                                        ; Сбросить все регистры порта.
        mov     dx,COM+1                ; Регистр IER.
        mov     al,0
        out     dx,al                   ; Запретить все прерывания.
        mov     dx,COM+4                ; MCR.
        out     dx,al                   ; Сбросить все линии модема в 0
        mov     dx,COM+5                ; и выполнить чтение из LSR,
```

```
        in      al,dx
        mov     dx,COM+0        ; из RBR
        in      al,dx
        mov     dx,COM+6        ; и из MSR
        in      al,dx           ; на тот случай, если они недавно изменялись,
        mov     dx,COM+2        ; а также послать 0 в регистр FCR,
        mov     al,0            ; чтобы выключить FIFO.
        out     dx,al

; Установка скорости COM-порта.
        mov     dx,COM+3        ; Записать в регистр LCR
        mov     al,80h          ; любое число со старшим битом 1.
        out     dx,al
        mov     dx,COM+0        ; Теперь записать в регистр DLL
        mov     al,2            ; младший байт делителя скорости,
        out     dx,al
        mov     dx,COM+1        ; а в DLH -
        mov     al,0            ; старший байт
        out     dx,al           ; (мы записали 0002h - скорость порта 57 600).

; Инициализациялинии.
        mov     dx,COM+3        ; Записать теперь в LCR
        mov     al,0011b        ; число, соответствующее режиму 8N1
        out     dx,al           ; (наиболее часто используемому).
; Инициализация модема.
        mov     dx,COM+4          Записать в регистр MCR
        mov     al,1011b        битовую маску, активизирующую DTR, RTS
        out     dx,al           и OUT2.
; Здесь следует выполнить проверку на наличие модема на этом порту (читать
; регистр MSR, пока не будут установлены линии CTS и DSR или не кончится время),
; а затем послать в модем (то есть поместить в буфер передачи) инициализирующую
; строку, например "ATZ",0Dh.

; Разрешение прерываний.
        mov     dx,COM+1        ; Записать в IER битовую маску, разрешающую
        mov     al,1101b        ; все прерывания, кроме "регистр передачи пуст".
        out     dx,al
        in      al,21h          ; Прочитать OCW1 (см. раздел 5.10.10).
        and     al,E_BITMASK    ; Размаскировать прерывание.
        out     21h,al          ; Записать OCW1.
        ret
init_everything endp

; Возвращение всего в исходное состояние.
shutdown_everything     proc    near
; Запрещение прерываний.
        in      al,21h          ; Прочитать OCW1.
        or      al,D_BITMASK    ; Замаскировать прерывание.
        out     21h,al          : Записать OCW1.
        mov     dx,COM+1        ; Записать в регистр IER
        mov     al,0            ; ноль.
```

```
        out     dx,al
; Сброс линий модема DTR и CTS.
        mov     dx,COM+4        ; Записать в регистр MCR
        mov     al,0            ; ноль.
        out     dx,al
; Восстановление предыдущего обработчика прерывания.
        mov     ax,2500h+IRQ    ; AH = 25h, AL = номер прерывания.
        lds     dx,old_irq      ; DS:DX - адрес обработчика.
        int     21h
        ret
shutdown_everything     endp
        end     start
```

5.10.3. Параллельныйпорт

BIOS автоматически обнаруживает только три параллельных порта - с адресами 0378h - 037Ah (LPT1 или LPT2), 0278h - 027Ah (LPT2 или LPT3) и 03BCh - 03BFh (LPT1, если есть) - и записывает номера их базовых портов ввода-вывода в область данных BIOS по адресам 0040h:0008h, 0040h:OOOAh, 0040h:000Ch соответственно. Если в системе установлен еще один параллельный порт, придется дополнительно записывать его базовый номер в 0040h:OOOEh, чтобы BIOS воспринимала его как LPT4. Рассмотрим назначение портов ввода-вывода, управляющих параллельными портами на примере 0278h — 027Ah.

0278h для записи - порт данных. Чтение и запись в этот порт приводят к приему или отправке байта в принтер или другое присоединенное устройство.

0279h для чтения - порт состояния

- бит 7: принтер занят, находится в offline или произошла ошибка
- бит 6: нет подтверждения (1 - принтер не готов к приему следующего байта)
- бит 5: нет бумаги
- бит 4: принтер в режиме online
- бит 3: нет ошибок
- бит 2: IRQ не произошло
- биты 1-0: O

027Ah для чтения и записи - порт управления

- бит 5: включить двунаправленный обмен данными (этот режим не поддерживается BIOS)
- бит 4: включить генерацию аппаратного прерывания (по сигналу подтверждения)
- бит 3: установить принтер в online
- бит 2: 0 в этом бите инициализирует принтер
- бит 1: режим посылки символа LF (OAh) после каждого CR (0Dh)
- бит 0: линия STROBE

Чтобы послать байт в принтер, программа должна убедиться, что линия BUSY (бит 7 порта состояния) равна нулю, а линия ACK (бит 6 порта состояния) - единице. Затем надо послать символ на линии DATA (порт данных), не ранее чем

через 0,5 мкс установить линию STROBE (бит 0 порта управления) в 0, а затем, не менее чем через 0,5 мкс, - в 1. В отличие от последовательных портов, параллельные хорошо поддерживаются BIOS и DOS, так что программирование их на уровне портов ввода-вывода может потребоваться только при написании драйвера для какого-нибудь необычного устройства, подключаемого к параллельному порту, или, например, при написании драйвера принтера для новой операционной системы.

5.10.4. Видеоадаптеры VGA

VGA-совместимые видеоадаптеры управляются при помощи портов ввода-вывода 03C0h - 03CFh, 03B4h, 03B5h, 03D4h, 03D5h, 03DAh, причем реальное число внутренних регистров видеоадаптера, к которым можно обращаться через это окно, превышает 50. Так как BIOS предоставляет хорошую поддержку для большинства стандартных функций, мы не будем подробно говорить о программировании видеоадаптера на уровне портов, а только рассмотрим основные действия, для которых принято обращаться к видеоадаптеру напрямую.

Внешние регистры контроллера VGA (03C2h - 03CFh)

Доступ к этим регистрам осуществляется прямым обращением к соответствующим портам ввода-вывода.

Регистр состояния ввода 0 (ISRO) - доступен для чтения из порта 03C2
- бит 7: произошло прерывание обратного хода луча IRQ2
- бит 6: дополнительное устройство 1 (линия FEAT1)
- бит 5: дополнительное устройство 0 (линия FEATO)
- бит 4: монитор присутствует

Регистр вывода (MOR) - доступен для чтения из порта 3CCh и для записи как 3C2h
- биты 7-6: полярность сигналов развертки: (01, 10, 11) = (400, 350, 480) линий
- бит 5: 1/0: нечетная/четная страница видеопамяти
- биты 3-2: частота: (00, 01) = (25,175 МГц, 28,322 МГц)
- бит 1: 1/0: доступ CPU к видеопамяти разрешен/запрещен
- бит 0: 1/0: адрес порта контроллера CRT = 03D4h/03B4h

Регистр состояния ввода 1 (ISR1) - доступен для чтения из порта 03DAh
- бит 3: происходит вертикальный обратный ход луча
- бит 0: происходит любой обратный ход луча

Лучшим моментом для вывода данных в видеопамять является тот, когда электронный луч двигается от конца экрана к началу и экран не обновляется, то есть вертикальный обратный ход луча. Перед копированием в видеопамять полезно вызывать, например, следующую процедуру:

```
; Процедура wait_retrace.
; Возвращает управление в начале обратного вертикального хода луча.
;
wait_retrace    proc    near
        push    ax
```

```
        push    dx
        mov     dx,03DAh        ; Порт регистра ISR1.
wait_retrace_end:
        in      al,dx
        test    al,1000b        ; Проверить бит 3.
                                ; Если не ноль -
        jnz     wait_retrace_end ; подождать конца текущего обратного хода,
wait_retrace_start:
        in      al,dx
        test    al,1000b        ; а теперь подождать начала следующего.
        jz      wait_retrace_start
        pop     dx
        pop     ax
        ret
wait_retrace    endp
```

Регистры контроллера атрибутов (03C0h - 03C1h)

Контроллер атрибутов преобразовывает значения байта атрибута символа в цвета символа и фона. Надо записать в порт 03C0h номер регистра, а затем (второй командой out) - данные для этого регистра. Чтобы убедиться, что 03C0h находится в состоянии приема номера, а не данных, надо выполнить чтение из ISR1 (порт 03DAh). Порт 03C1h можно использовать для чтения последнего записанного индекса или данных.

00h - 0Fh: Регистры палитры EGA
биты 5-0: номер регистра в текущей странице VGA DAC, соответствующего данному EGA-цвету

10h: Регистр управления режимом
бит 7: разбиение регистров VGA DAC для 16-цветных режимов:
1 = 16 страниц по 16 регистров, 0 = 4 страницы по 64 регистра
бит 6: 1 = 8-битный цвет, 0 = 4-битный цвет
бит 5: горизонтальное панорамирование разрешено
бит 3: 1/0: бит 7 атрибута управляет миганием символа/цветом фона
бит 2: девятый пиксел в каждой строке повторяет восьмой
бит 1: 1/0: генерация атрибутов для монохромных/цветных режимов
бит 0: 1/0: генерация атрибутов для текстовых/графических режимов

11h: Регистр цвета бордюра экрана (по умолчанию OOh)
биты 7-0: номер регистра VGA DAC

12h: Регистр разрешения использования цветовых плоскостей
бит 3: разрешить плоскость 3
бит 2: разрешить плоскость 2
бит 1: разрешить плоскость 1
бит 0: разрешить плоскость О

13h: Регистр горизонтального панорамирования
биты 3-0: величина сдвига по горизонтали в пикселах (деленная на 2 для режима 13h)

14h: Регистр выбора цвета (по умолчанию OOh)

Функции INT 10h AX = 1000h - 1009h позволяют использовать большинство из этих регистров, но кое-что, например панорамирование, оказывается возможным только при программировании на уровне портов.

Регистры графического контроллера (03CEh - 03CFh)

Для обращения к регистрам графического контроллера следует записать индекс нужного регистра в порт 03CEh, после чего можно будет читать и писать данные для выбранного регистра в порт 03CFh. Если требуется только запись в регистры, можно просто поместить индекс в AL, посылаемый байт - в AH и выполнить команду вывода слова в порт 03CEh. Этот контроллер, в первую очередь, предназначен для обеспечения передачи данных между процессором и видеопамятью в режимах, использующих цветовые плоскости, как, например, режим 12h (640x480x16).

00h: Регистр установки/сброса
биты 3-0: записывать FFh в цветовую плоскость 3-0 соответственно
01h: Регистр разрешения установки/сброса
биты 3-0: включить режим установки/сброса для цветовой плоскости 3-0
В этом режиме данные для одних цветовых слоев получают от CPU, а для других - из регистра установки/сброса. Режим действует только в нулевом режиме работы (см. регистр 05h).
02h: Регистр сравнения цвета
биты 3–0: искомые биты для цветовых плоскостей 3-0
Используется для поиска пиксела заданного цвета, чтобы не обращаться по очереди во все цветовые слои.
03h: Регистр циклического сдвига данных
биты 4-3: выбор логической операции:
00 - данные от CPU записываются без изменений
01 - операция AND над CPU и регистром-защелкой
10 - операция OR над CPU и регистром-защелкой
11 - операция XOR над CPU и регистром-защелкой
биты 2-0: на сколько битов выполнять вправо циклический сдвиг данных перед записью в видеопамять
04h: Регистр выбора читаемой плоскости
биты 1-0: номер плоскости (0-3)
Запись сюда изменяет номер цветовой плоскости, данные из которой получает CPU при чтении из видеопамяти.
05h: Регистр выбора режима работы
бит 6: 1/0: 256/16 цветов
бит 4: четные адреса соответствуют плоскостям 0, 2; нечетные – 1, 3
бит 3: 1 режим сравнения цветов
биты 1-0: режим:
00: данные из CPU (бит на пиксел) + установка/сброс + циклический сдвиг + логические функции

01: данные в/из регистра-защелки (прочитать в него и записать в другую область памяти быстрее, чем через CPU)
10: данные из CPU, байт на пиксел, младшие 4 бита записываются в соответствующие плоскости
11: то же самое + режим битовой маски

06h: Многоцелевой регистр графического контроллера
биты 3–2: видеопамять:
00: OAOOOOh - OBFFFFh (128 Кб)
01: OAOOOOh - 0AFFFFh (64 Кб)
10: 0B0000h - 0B7FFFh (32 Кб)
11: 0B8000h - OBFFFFh (32 Кб)
бит 0: 1/0: графический/текстовый режим

07h: Регистр игнорирования цветовых плоскостей
биты 3-0: игнорировать цветовую плоскость 3–0

08h: Регистр битовой маски
Если бит этого регистра 0, то соответствующий бит будет браться из регистра-защелки, а не из CPU. (Чтобы занести данные в регистр-защелку, надо выполнить одну операцию чтения из видеопамяти, при этом в каждый из четырех регистров-защелок будет помещено по одному байту из соответствующей цветовой плоскости.)

Графический контроллер предоставляет весьма богатые возможности по управлению режимами, использующими цветовые плоскости. В качестве примера напишем процедуру, выводящую точку на экран в режиме 12h (640x480x16) с применением механизма установки/сброса.

```
; Процедура putpixel12h.
; Выводит на экран точку с заданным цветом в режиме 12h (640x480x16).
;       Вход:   DX = строка
;               CX = столбец
;               BP = цвет
;               ES = 0A000h

putpixel12h     proc    near
        pusha
; Вычислить номер байта в видеопамяти.
        xor     bx,bx
        mov     ax,dx           ; AX = строка.
        lea     eax,[eax+eax*4] ; AX = AX x 5.
        shl     ax,4            ; AX = AX x 16.
                                ; AX = строка x байт_в_строке
                                ; (строка x 80).
        push    cx
        shr     cx,3            ; CX = номер байта в строке.
        add     ax,cx           ; AX = номер байта в видеопамяти.
        mov     di,ax           ; Сохранить его в DI.
                                ; Вычислить номер бита в байте.
        pop     cx
```

```
        and     cx,07h          ; Остаток от деления на 8 - номер
                                ; бита в байте, считая справа налево.
        mov     bx,0080h
        shr     bx,cl           ; В BL теперь нужный бит установлен в 1.
                                ; Программирование портов.
        mov     dx,03CEh        ; Индексный порт
                                ; графического контроллера.
        mov     ax,0F01h        ; Регистр 01h: разрешение
                                ; установки/сброса.
        out     dx,ax           ; Разрешить установку/сброс для
                                ; всех плоскостей (эту часть лучше
; сделать однажды в программе, например сразу после установки
; видеорежима, и не повторять каждый раз при вызове процедуры).
        mov     ax,bp
        shl     ax,8            ; Регистр 00h: регистр
                                ; установки/сброса.
        out     dx,ax           ; AH = цвет.
        mov     al,08           ; Порт 08h: битовая маска.
        mov     ah,bl           ; Записать в битовую маску нули
                                ; всюду, кроме
        out     dx,ax           ; бита, соответствующего выводимому пикселу.
        mov     ah, byte ptr es:[di]    ; Заполнить
                                        ; регистры-защелки.
        mov     byte ptr es:[di],ah     ; Вывод на экран:
                                        ; выводится единственный бит
; в соответствии с содержимым регистра битовой маски, остальные
; биты берутся из защелки, то есть не изменяются. Цвет выводимого
; бита полностью определяется значением регистра установки/сброса.
        popa
        ret
putpixel12h     endp
```

Регистры контроллера CRT (03D4h - 03D5h)

Контроллер CRT управляет разверткой и формированием кадров на дисплее. Как и для графического контроллера, чтобы обратиться к регистрам контроллера CRT, следует записать индекс нужного регистра в порт 03D4h, после чего можно будет читать и писать данные для выбранного регистра в порт 03D5h. Если требуется только запись в регистры, можно просто поместить индекс в AL, посылаемый байт — в AH и выполнить команду вывода слова в порт 03D4h.

OOh: общая длина горизонтальной развертки
01h: длина отображаемой части горизонтальной развертки минус один
02h: начало гашения луча горизонтальной развертки
03h: конец гашения луча горизонтальной развертки
биты 6–5: горизонтальное смещение в текстовых режимах
биты 4-0: конец импульса
04h: начало горизонтального обратного хода луча

05h: конец горизонтального обратного хода луча
биты 7, 4-0: конец импульса
биты 6-5: горизонтальное смещение импульса
06h: число вертикальных линий растра без двух старших битов
07h: дополнительный регистр
бит 7: бит 9 регистра 10h
бит 6: бит 9 регистра 12h
бит 5: бит 9 регистра 06h
бит 4: бит 8 регистра 18h
бит 3: бит 8 регистра 15h
бит 2: бит 8 регистра 10h
бит 1: бит 8 регистра 12h
бит 0: бит 8 регистра 06h
08h: предварительная горизонтальная развертка
биты 6-5: биты 5 и 4 регистра горизонтального панорамирования
биты 4-0: номер линии в верхней строке, с которой начинается изображение
09h: высота символов
бит 7: двойное сканирование (400 линий вместо 200)
бит 6: бит 9 регистра 18h
бит 5: бит 9 регистра 15h
биты 4–0: высота символов минус один (от 0 до 31)
0Ah: начальная линия курсора (бит 5: гашение курсора)
0Bh: конечная линия курсора (биты 6-5: отклонение курсора вправо)
0Ch: старший байт начального адреса
ODh: младший байт начального адреса (это адрес в видеопамяти, начиная с которого выводится изображение)
OEh: старший байт позиции курсора
0Fh: младший байт позиции курсора
10h: начало вертикального обратного хода луча без старшего бита
11h: конец вертикального обратного хода луча без старшего бита
бит 7: защита от записи в регистры 00-07 (кроме бита 4 в 07h)
бит 6: 1/0: 5/3 цикла регенерации за время обратного хода луча
бит 5: 1/0: выключить/включить прерывание по обратному ходу луча
бит 4: запись нуля сюда заканчивает обработку прерывания
биты 3-0: конец вертикального обратного хода луча
12h: число горизонтальных линий минус один без двух старших битов
13h: логическая ширина экрана (в словах/двойных словах на строку)
14h: положение символа подчеркивания
бит 6: 1/0: адресация словами/двойными словами
бит 5: увеличение счетчика адреса регенерации на 4
биты 4-0: положение подчеркивания
15h: начало импульса гашения луча вертикальной развертки без двух старших битов
16h: конец импульса гашения вертикальной развертки
17h: регистр управления режимом

бит 7: горизонтальный и вертикальный ходы луча отключены
бит 6: 1/0 - адресация байтами/словами
бит 4: 1 - контроллер выключен
бит 3: 1/0 - счетчик адреса регенерации растет на 2/1 на каждый символ
бит 2: увеличение в 2 раза разрешения по вертикали
18h: регистр сравнения линий без двух старших битов
(от начала экрана до линии с номером из этого регистра отображается начало видеопамяти, а от этой линии до конца - видеопамять, начиная с адреса, указанного в регистрах 0Ch и ODh)
22h: регистр-защелка (только для чтения)
23h: состояние контроллера атрибутов
биты 7-3: текущее значение индекса контроллера атрибутов
бит 2: источник адреса палитры
бит 0: состояние порта контроллера атрибутов: 0/1 = индекс/данные

BIOS заполняет регистры этого контроллера соответствующими значениями при переключении видеорежимов. Поскольку одного контроллера CRT мало для полного переключения в новый видеорежим, мы вернемся к этому чуть позже, а пока посмотрим, как внести небольшие изменения в действующий режим, например: как превратить текстовый режим 80x25 в 80x30:

```
; 80x30.asm
; Переводит экран в текстовый режим 80x30 (размер символов 8x16)
; (Norton Commander 5.0 в отличие от, например, FAR восстанавливает режим по
; окончании программы, но его можно обмануть, если предварительно нажать Alt-F9).

        .model tiny
        .code
        .186                    ; Для команды outsw.
        org     100h            ; COM-программа.
start:
        mov     ax,3            ; Установить режим 03h (80X25),
        int     10h             ; чтобы только внести небольшие изменения.

        mov     dx,3CCh         ; Порт 3CCh: регистр вывода (MOR) на чтение.
        in      al,dx
        mov     dl,0C2h         ; Порт 03C2h: регистр вывода (MOR) на запись.
        or      al,0C0h         ; Установить полярности 1; 1 - для 480 строк.
        out     dx,al

        mov     dx,03D4h        ; DX = порт 03D4h: индекс CRT.
        mov     si,offset crt480 ; DS:SI = адрес таблицы данных для CRT.
        mov     cx,crt480_l     ; CX = ее размер.
        rep     outsw           ; Послать все устанавливаемые параметры в порты
                                ; 03D4h и 03D5h.

; Нельзя забывать сообщать BIOS об изменениях в видеорежиме.
        push    0040h
        pop     es              ; ES = 0040h.
```

```
        mov     byte ptr es:[84h],29      ; 0040h:0084h - число строк.
        ret

; Данные для контроллера CRT в формате индекс в младшем байте, данные
; в старшем - для записи при помощи команды outsw.
crt480          dw      0C11h     ; Регистр 11h всегда надо записывать первым,
                                  ; так как его бит 7 разрешает запись в другие
                dw      0B06h,3E07h,0EA10h,0DF12h,0E715h,0416h     ; регистры.
crt480_l = ($-crt480)/2
                end start
```

Еще одна интересная возможность, которую предоставляет контроллер CRT, – плавная прокрутка экрана при помощи регистра 08h:

```
; vscroll.asm
; Плавная прокрутка экрана по вертикали. Выход - клавиша Esc.
;
        .model  tiny
        .code
        .186                    ; Для push 0B400h.
        org100h                 ; COM-программа.
start:
        push    0B800h
        pop     es
        xor     si,si           ; ES:SI - начало видеопамяти.
        mov     di,80*25*2      ; ES:DI - начало второй страницы видеопамяти.
        mov     cx,di
        rep movs es:any_label,es:any_label ; Скопировать первую страницу во вторую.
        mov     dx,03D4h        ; Порт 03D4h: индекс CRT.
screen_loop:                    ; Цикл по экранам.
        mov     cx,80*12*2      ; CX = начальный адрес - адрес середины экрана.
line_loop:                      ; Цикл по строкам.
        mov     al,0Ch          ; Регистр 0Ch - старший байт начального адреса.
        mov     ah,ch           ; Байт данных - CH.
        out     dx,ax           ; Вывод в порты 03D4, 03D5.
        inc     ax              ; Регистр 0Dh - младший байт начального адреса.
        mov     ah,cl           ; Байт данных - CL.
        out     dx,ax           ; Вывод в порты 03D4, 03D5.

        mov     bx,15           ; Счетчик линий в строке.
        sub     cx,80           ; Переместить начальный адрес на начало
                                ; предыдущей строки (так как это движение вниз).
pel_loop:                       ; Цикл по линиям в строке.
        call    wait_retrace    ; Подождать обратного хода луча.

        mov     al,8            ; Регистр 08h - выбор номера линии в первой
                                ; строке, с которой начинается вывод изображения
        mov     ah,bl           ; (номер линии из BL).
        out     dx,ax
```

```
        dec     bx                  ; Уменьшить число линий.
        jge     pel_loop            ; Если больше или равно нулю - строка еще не
                                    ; прокрутилась до конца и цикл по линиям
                                    ; продолжается.

        in      al,60h              ; Прочитать скан-код последнего символа.
        cmp     al,81h              ; Если это 81h (отпускание клавиши Esc),
        jz      done                ; выйти из программы.

        cmp     cx,0                ; Если еще не прокрутился целый экран,
        jge     line_loop           ; продолжить цикл по строкам.
        jmp     short screen_loop;  Иначе: продолжить цикл по экранам.

done:                               ; Выход из программы.
        mov     ax,8                ; Записать в регистр CRT 08h
        out     dx,ax               ; байт 00 (никакого сдвига по вертикали),
        add     ax,4                ; а также 00 в регистр 0Ch
        out     dx,ax
        inc     ax                  ; и ODh (начальный адрес совпадает
        out     dx,ax               ; с началом видеопамяти).
        ret

wait_retrace    proc    near
        push    dx
        mov     dx,03DAh
VRTL1:  in      al,dx               ; Порт 03DAh - регистр ISR1.
        test    al,8
        jnz     VRTL1               ; Подождать конца текущего обратного хода луча,
VRTL2:  in      al,dx
        test    al,8
        jz      VRTL2               ; а теперь начала следующего.
        pop     dx
        ret
wait_retrace    endp
any_label       label byte          ; Метка для переопределения сегмента в MOVS.
        end     start
```

Горизонтальная прокрутка осуществляется аналогично, только с использованием регистра горизонтального панорамирования 13h из контроллера атрибутов.

Регистры синхронизатора (03C4h - 03C5h)

Для обращения к регистрам синхронизатора следует записать индекс нужного регистра в порт 03C4h, после чего можно будет читать и писать данные для выбранного регистра в порт 03C5h. Точно так же, если требуется только запись в регистры, можно просто поместить индекс в AL, посылаемый байт - в AH и выполнить команду вывода слова в порт 03CEh.

00h: регистр сброса синхронизации
бит 1: запись нуля сюда вызывает синхронный сброс
бит 0: запись нуля сюда вызывает асинхронный сброс

01h: регистр режима синхронизации
бит 5: 1; обмен данными между видеопамятью и дисплеем выключен
бит 3: 1: частота обновления для символов уменьшена в два раза
бит 0: 1/0: ширина символа 8/9 точек
02h: регистр маски записи
бит 3: разрешена запись CPU в цветовую плоскость 3
бит 2: разрешена запись CPU в цветовую плоскость 2
бит 1: разрешена запись CPU в цветовую плоскость 1
бит 0: разрешена запись CPU в цветовую плоскость О
03h: регистр выбора шрифта
бит 5: если бит 3 атрибута символа = 1, символ берется из шрифта 2
бит 4: если бит 3 атрибута символа = 0, символ берется из шрифта 2
биты 3-2: номер таблицы для шрифта 2
биты 1-0: номер таблицы для шрифта 1
(00, 01, 10, 11) = (О Кб, 16 Кб, 32 Кб, 48 Кб от начала памяти шрифтов VGA)
04h: регистр организации видеопамяти
бит 3: 1: режим CHAIN-4 (используется только в видеорежиме 13h)
бит 2: 0: четные адреса обращаются к плоскостям 0, 2, нечетные - к 1, 3
бит 1: объем видеопамяти больше 64 Кб

Даже несмотря на то, что BIOS позволяет использовать некоторые возможности этих регистров, в частности работу со шрифтами (INT 10h AH = 11h) и выключение обмена данными между видеопамятью и дисплеем (INT 10h, AH = 12h, BL = 32h), прямое программирование регистров синхронизатора вместе с регистрами контроллера CRT разрешает изменять характеристики видеорежимов VGA, вплоть до установки нестандартных видеорежимов. Наиболее популярными режимами являются так называемые режимы X с 256 цветами и с разрешением 320 или 360 пикселов по горизонтали и 200, 240, 400 или 480 пикселов по вертикали. Поскольку такие режимы не поддерживаются BIOS, для их реализации нужно написать все необходимые процедуры - установку видеорежима, вывод пиксела, чтение пиксела, переключение страниц, изменение палитры, загрузку шрифтов. При этом для всех режимов из этой серии, кроме 320x240x256, приходится также учитывать измененное соотношение размеров экрана по вертикали и горизонтали, чтобы круг, выведенный на экран, не выглядел как эллипс, а квадрат - как прямоугольник.

Установка нового режима выполняется почти точно так же, как и в предыдущем примере, - путем модификации существующего. Кроме того, нам придется изменять частоту кадров (биты 3-2 регистра MOR), а это приведет к сбою синхронизации, если мы не выключим синхронизатор на время изменения частоты (записью в регистр 00h):

```
; Процедура set_modex.
; Переводит видеоадаптер VGA в один из режимов X с 256 цветами.
; Вход: DI = номер режима
;       0: 320x200, соотношение сторон 1,2:1
;       1: 320x400, соотношение сторон 2,4:1
```

```
;       2: 360x200, соотношение сторон 1,35:1
;       3: 360x400, соотношение сторон 2,7:1
;       4: 320x240, соотношение сторон 1:1
;       5: 320x480, соотношение сторон 2:1
;       6: 360x240, соотношение сторон 1,125:1
;       7: 360x480, соотношение сторон 2,25:1
;       DS = CS
; Для вывода информации на экран в этих режимах
; см. процедуру putpixel_x
;
setmode_x       proc    near
        mov     ax,12h          ; Очистить все четыре цветовые
        int     10h             ; плоскости видеопамяти.
        mov     ax,13h          ; Установить режим 13h, который будем
        int     10h             ; модифицировать.

        cmp     di,7            ; Если нас вызвали с DI > 7,
        ja      exit_modex      ; выйти из процедуры
                                ; (оставшись в режиме 13h).
        shl     di,1            ; Умножить на 2, так как x_modes -
                                ; таблица слов.
        mov     di,word ptr x_modes[di]  ; Прочитать
                                ; адрес таблицы настроек для
                                ; выбранного режима.

        mov     dx,03C4h        ; Порт 03C4h - индекс синхронизатора.
        mov     ax,0100h        ; Регистр 00h, значение 01.
        out     dx,ax           ; Асинхронный сброс.
        mov     ax,0604h        ; Регистр 04h, значение 06h.
        out     dx,ax           ; Отключить режим CHAIN4.

        mov     dl,0C2h         ; Порт 03C2h - регистр
                                ; MOR на запись.
        mov     al,byte ptr [di] ; Записать в него
                                ; значение частоты кадров
        out     dx,al           ; и полярности развертки
                                ; для выбранного режима.

        mov     dl,0D4h         ; Порт 03D4h - индекс
                                ; контроллера CRT.
        mov     si,word ptr offset [di+2]
                                ; Адрес строки с настройками
                                ; для выбранной ширины в DS:SI.
        mov     cx,8            ; Длина строки настроек в CX.
        rep     outsw           ; Вывод строки слов
                                ; в порты 03D4/03D5.
        mov     si,word ptr offset [di+4] ; Настройки для
                                ; выбранной высоты в DS:SI.
        mov     cx,7            ; Длина строки настроек в CX/
        rep     outsw
```

```
        mov     si,word ptr offset [di+6]; Настройки
                                        ; для включения/выключения удвоения
                                        ; по вертикали (200/400 и 240/480 строк).
        mov     cx,3
        rep     outsw

        mov     ax, word ptr offset [di+8]      ; Число байтов в строке.
        mov     word ptr x_width,ax     ; Сохранить в переменной x_width.

        mov     dl,0C4h         ; Порт 03C4h - индекс синхронизатора.
        mov     ax,0300h        ; Регистр 00h, значение 03.
        out     dx,ax           ; Выйти из состояния сброса.
exit_modex:
        ret

; Таблица адресов таблиц с настройками режимов.
x_modes         dw      offset mode_0,offset mode_1
                dw      offset mode_2,offset mode_3
                dw      offset mode_4,offset mode_5
                dw      offset mode_6,offset mode_7

; Таблица настроек режимов: значение регистра MOR, адрес строки
; настроек ширины, адрес строки настроек высоты, адрес строки
; настроек удвоения по вертикали, число байтов в строке.
mode_0  dw      63h,offset mode_320w,offset mode_200h,offset mode_double,320/4
mode_1  dw      63h,offset mode_320w,offset mode_400h,offset mode_single,320/4
mode_2  dw      67h,offset mode_360w,offset mode_200h,offset mode_double,360/4
mode_3  dw      67h,offset mode_360w,offset mode_400h,offset mode_single,360/4
mode_4  dw      0E3h,offset mode_320w,offset mode_240h,offset mode_double,320/4
mode_5  dw      0E3h,offset mode_320w,offset mode_480h,offset mode_single,320/4
mode_6  dw      0E7h,offset mode_360w,offset mode_240h,offset mode_double,360/4
mode_7  dw      0E7h,offset mode_360w,offset mode_480h,offset mode_single,360/4

; Настройки CRT. В каждом слове младший байт - номер регистра,
; старший - значение, которое в этот регистр заносится.
mode_320w:                      ; Настройка ширины 320.
; Первый регистр обязательно 11h, хотя он и не относится к ширине,
; но разрешает запись в остальные регистры, если она была запрещена (!).
        dw      0E11h,5F00h,4F01h,5002h,8203h,5404h,8005h,2813h
mode_360w:                      ; Настройка ширины 360.
        dw      0E11h,6B00h,5901h,5A02h,8E03h,5E04h,8A05h,2D13h
mode_200h:
mode_400h:                      ; Настройка высоты 200/400.
        dw      0BF06h,1F07h,9C10h,0E11h,8F12h,9615h,0B916h
mode_240h:
mode_480h:                      ; Настройка высоты 240/480.
        dw      0D06h,3E07h,0EA10h,0C11h,0DF12h,0E715h,0616h
mode_single:                    ; Настройка режимов без удвоения.
        dw      4009h,0014h,0E317h
mode_double:                    ; Настройка режимов с удвоением.
        dw      4109h,0014h,0E317h
setmode_x       endp
```

```
x_width  dw      ?                ; Число байтов в строке.
; Эту переменную инициализирует setmode_x, а использует putpixel_x.

; Процедура putpixel_x.
; Выводит точку с заданным цветом в текущем режиме X.
; Вход: DX = строка
;       CX = столбец
;       BP = цвет
;       ES = 0A000h
;       DS = сегмент, в котором находится переменная x_width

putpixel_x      proc    near
        pusha

        mov     ax,dx
        mul     word ptr x_width ; AX = строка x число байтов в строке.
        mov     di,cx            ; DI = столбец.
        shr     di,2             ; DI = столбец/4 (номер байта в строке).
        add     di,ax            ; DI = номер байта в видеопамяти/

        mov     ax,0102h         ; AL = 02h (номер регистра).
                                 ; AH = 01 (битовая маска).
        and     cl,03h           ; CL = остаток от деления столбца на 4 =
                                 ; номер цветовой плоскости.
        shl     ah,cl            ; Теперь в AH выставлен в 1 бит,
                                 ; соответствующий нужной цветовой плоскости.
        mov     dx,03C4h         ; Порт 03C4h - индекс синхронизатора.
        out     dx,ax            ; Разрешить запись только
                                 ; в нужную плоскость.

        mov     ax,bp            ; Цвет в AL.
        stosb                    ; Вывод байта в видеопамять.

        popa
        ret
putpixel_x      endp
```

Регистры VGA DAC (3C6h - 3C9h)

Таблица цветов VGA на самом деле представляет собой 256 регистров, в каждом из которых записаны три 6-битных числа, соответствующих уровням красного, зеленого и синего цвета. Подфункции INT 10h AX = 1010h — 101Bh позволяют удобно работать с этими регистрами, но, если требуется максимальная скорость, программировать их на уровне портов ввода-вывода не намного сложнее.

03C6h для чтения/записи: регистр маскирования пикселов (по умолчанию 0FFh)
При обращении к регистру DAC выполняется операция AND над его номером и содержимым этого регистра.

03C7h для записи: регистр индекса DAC для режима чтения
Запись байта сюда переводит DAC в режим чтения, так что следующее чтение из 3C9h вернет значение регистра палитры с этим индексом.

03C7h для чтения: регистр состояния DAC
биты 1-0: 00b/11b - DAC в режиме записи/чтения

03C8h для чтения/записи: регистр индекса DAC для режима записи
Запись байта сюда переводит DAC в режим записи, поэтому дальнейший вывод в 03C9h будет приводить к записи новых значений в регистры палитры, начиная с этого индекса.

03C9h для чтения/записи: регистр данных DAC
Чтение отсюда считывает значение регистра палитры с индексом, помещенным предварительно в 03C8h, запись - записывает новое значение в регистр палитры с индексом, находящимся в 03C8h. На каждый регистр требуются три операции чтения/записи, передающие три 6-битных значения уровня цвета: красный, зеленый, синий. После третьей операции чтения/записи индекс текущего регистра палитры увеличивается на 1, так что можно считывать/записывать сразу несколько регистров

Команды insb/outsb серьезно облегчают работу с регистрами DAC в тех случаях, когда требуется считывать или загружать значительные участки палитры или всю палитру целиком, - такие процедуры оказываются и быстрее, и меньше аналогичных, написанных с использованием прерывания INT 10h. Посмотрим, как это реализуется на примере программы плавного гашения экрана.

```
; fadeout.asm
; Выполняет плавное гашение экрана.

        .model tiny
        .code
        .186                            ; Для команд insb/outsb.
        org     100h                    ; COM-программа.
start:
        cld                             ; Для команд строковой обработки.
        mov     di,offset palettes
        call    read_palette            ; Сохранить текущую палитру, чтобы
                                        ; восстановить в самом конце программы,
        mov     di,offset  palettes+256*3
        call    read_palette            ; а также записать еще одну копию
                                        ; текущей палитры, которую будем
                                        ; модифицировать.
        mov     cx,64                   ; Счетчик цикла изменения палитры.
main_loop:
        push    cx
        call    wait_retrace            ; Подождать начала обратного хода луча.
        mov     di,offset palettes+256*3
        mov     si,di
        call    dec_palette             ; Уменьшить яркость всех цветов.
        call    wait_retrace            ; Подождать начала следующего
        mov     si,offset palettes+256*3 ; обратного хода луча.
        call    write_palette           ; Записать новую палитру.
        pop     cx
```

```
        loop    main_loop               ; Цикл выполняется 64 раза - достаточно
                                        ; для обнуления самого яркого цвета
                                        ; (максимальное значение 6-битной
                                        ; компоненты - 63).
        mov     si,offset palettes
        call    write_palette           ; Восстановить первоначальную палитру.
        ret                             ; Конец программы.

; Процедура read_palette.
; Помещает палитру VGA в строку по адресу ES:DI.
read_palette    proc    near
        mov     dx,03C7h                ; Порт 03C7h - индекс DAC/режим чтения.
        mov     al,0                    ; Начинать с нулевого цвета.
        out     dx,al
        mov     dl,0C9h                 ; Порт 03C9h - данные DAC.
        mov     cx,256*3                ; Прочитать 256 x 3 байт
        rep     insb                    ; в строку по адресу ES:DI.
        ret
read_palette    endp

; Процедура write_palette.
; Загружает в DAC VGA палитру из строки по адресу DS:SI.
write_palette   proc    near
        mov     dx,03C8h                ; Порт 03C8h - индекс DAC/режим записи.
        mov     al,0                    ; Начинать с нулевого цвета.
        out     dx,al
        mov     dl,0C9h                 ; Порт 03C9h - данные DAC.
        mov     cx,256*3                ; Записать 256 x 3 байт
        rep     outsb                   ; из строки в DS:SI.
        ret
write_palette   endp

; Процедура dec_palette.
; Уменьшает значение каждого байта на 1 с насыщением (то есть, после того как
; байт становится равен нулю, он больше не уменьшается) из строки в DS:SI
; и записывает результат в строку в DS:SI.

dec_palette     proc    near
        mov     cx,256*3                ; Длина строки 256 x 3 байт.
dec_loop:
        lodsb                           ; Прочитать байт.
        test    al,al                   ; Если он ноль,
        jz      already_zero            ; пропустить следующую команду.
        dec     ax                      ; Уменьшить байт на 1.
already_zero:
        stosb                           ; Записать его обратно.
        loop    dec_loop                ; Повторить 256 x 3 раз.
        ret
dec_palette     endp

; Процедура wait_retrace.
; Ожидание начала следующего обратного хода луча.
```

```
wait_retrace    proc    near
        push    dx
        mov     dx,03DAh
VRTL1:  in      al,dx           ; Порт 03DAh - регистр ISR1.
        test    al,8
        jnz     VRTL1           ; Подождать конца текущего обратного хода луча.
VRTL2:  in      al,dx
        test    al,8
        jz      VRTL2           ; А теперь начала следующего.
        pop     dx
        ret
wait_retrace    endp

palettes:                       ; За концом программы мы храним две копии
                                ; палитры - всего 1,5 Кб.
        end start
```

5.10.5. Таймер

До сих пор о систменом таймере нам было известно лишь то, что он вызывает прерывание IRQ0 приблизительно 18,2 раза в секунду. На самом деле программируемый интервальный таймер - весьма сложная система, состоящая из трех частей - трех каналов таймера, каждый из которых можно запрограммировать для работы в одном из шести режимов. И более того, на многих современных материнских платах располагаются два таких таймера, следовательно, число каналов оказывается равным шести. Для своих нужд программы могут использовать канал 2 (если им не нужен динамик) и канал 4 (если присутствует второй таймер). При необходимости можно перепрограммировать и канал 0, но затем надо будет вернуть его в исходное состояние, чтобы BIOS и DOS могли продолжать работу.

В пространстве портов ввода-вывода для таймера выделена область от 40h до 5Fh:

- порт 40h - канал 0 (генерирует IRQ0);
- порт 41h - канал 1 (поддерживает обновление памяти);
- порт 42h - канал 2 (управляет динамиком);
- порт 43h - управляющий регистр первого таймера;
- порты 44h - 47h - второй таймер компьютеров с шиной MicroChannel;
- порты 48h - 4Bh — второй таймер компьютеров с шиной EISA.

Все управление таймером осуществляется путем вывода одного байта в 43h (для первого таймера). Рассмотрим назначение битов в этом байте.

биты 7-6: если не 11 - это номер канала, который будет программироваться
00, 01, 10 = канал О, 1, 2

бит 5-4: 00 - зафиксировать текущее значение счетчика для чтения (в этом случае биты 3-0 не используются)
01 - чтение/запись только младшего байта
10 - чтение/запись только старшего байта
11 - чтение/запись сначала младшего, а потом старшего байта

биты 3-1: режим работы канала
- 000 - прерывание IRQ0 при достижении нуля
- 001 - ждущий мультивибратор
- 010 - генератор импульсов
- OH - генератор прямоугольных импульсов (основной режим)
- 100 - программно запускаемый одновибратор
- 101 - аппаратно запускаемый одновибратор

бит 0: формат счетчика:
- 0 - двоичное 16-битное число (0000 - OFFFFh)
- 1 - двоично-десятичное число (0000 - 9999)

Если биты 7-6 равны 11, считается, что байт, посылаемый в 43h, - команда чтения счетчиков, формат которой отличается от команды программирования канала:

биты 7-6: 11 (код команды чтения счетчиков)

биты 5-4: что читать:
- 00 - сначала состояние канала, потом значение счетчика
- 01 - значение счетчика
- 10 - состояние канала

биты 3–1: команда относится к каналам 3-1

Если этой командой запрашивается состояние каналов, новые команды будут игнорироваться, пока не прочтется состояние из всех каналов, которые были заказаны битами 3-1.

Состояние и значение счетчика данного канала получают чтением из порта, соответствующего требуемому каналу. Формат байта состояния имеет следующий вид:

бит 7: состояние входа OUTx на момент выполнения команды чтения счетчиков. Так как в режиме 3 счетчик уменьшается на 2 за каждый цикл, состояние этого бита, замороженное командой фиксации текущего значения счетчика, укажет, в каком полуцикле находился таймер

бит 6: 1/0: состояние счетчика не загружено/загружено (используется в режимах 1 и 5, а также после команды фиксации текущего значения)

биты 5-0: совпадают с битами 5-0 последней команды, посланной в 43h

Для того чтобы запрограммировать таймер в режиме 3, в котором работают каналы 0 и 2 по умолчанию и который чаще всего применяют в программах, требуется выполнить следующие действия:

1. Вывести в регистр 43h команду (для канала 0) 001101 lb, то есть установить режим 3 для канала 0, и при чтении/записи будет пересылаться сначала младшее слово, а потом старшее.
2. Послать младший байт начального значения счетчика в порт, соответствующий выбранному каналу (42h для канала 2).
3. Послать старший байт начального значения счетчика в этот же порт.

После этого таймер немедленно начнет уменьшать введенное число от начального значения к нулю со скоростью 1 193 180 раз в секунду (четверть скорости

процессора 8088). Каждый раз, когда это число достигнет нуля, оно снова вернется к начальному значению. Кроме того, при достижении счетчиком нуля таймер выполняет соответствующую функцию - канал 0 вызывает прерывание IRQO, а канал 2, если включен динамик, посылает ему начало следующей прямоугольной волны, заставляя его работать на установленной частоте. Начальное значение счетчика для канала 0 по умолчанию составляет OFFFFh (65 535), то есть максимально возможное. Поэтому точная частота вызова прерывания IRQ0 равна 1 193 180/65 536 = 18,20648 раза в секунду.

Чтобы прочитать текущее значение счетчика, надо:

1. Послать в порт 43h команду фиксации значения счетчика для выбранного канала (биты 5-4 равны 00b).
2. Послать в порт 43h команду перепрограммирования канала без смены режима работы, если нужно воспользоваться другим способом чтения/записи (обычно не требуется).
3. Прочитать из порта, соответствующего выбранному каналу, младший байт зафиксированного значения счетчика.
4. Прочитать из того же порта старший байт.

Для таймера найдется много применений, единственное ограничение здесь следующее: таймер - это глобальный ресурс, и перепрограммировать его в многозадачных системах можно только с ведома операционной системы, если она вообще это позволяет.

Рассмотрим в качестве примера, как при помощи таймера измерить, сколько времени проходит между реальным аппаратным прерыванием и моментом, когда обработчик этого прерывания получает управление (почему это важно, см. примеры программ вывода звука из разделов 5.10.8 и 5.10.9). Так как IRQO происходит при нулевом значении счетчика, нам достаточно прочитать его значение во время старта обработчика и изменить его знак (потому что счетчик таймера постоянно уменьшается).

```
; latency.asm
; Измеряет среднее время, проходящее между аппаратным прерыванием и запуском
;  соответствующего обработчика.  Выводит среднее время в микросекундах после
; нажатия любой клавиши (на самом деле в 1/1 193 180).
;  Программа использует 16-битный сумматор для простоты,  так что может давать
; неверные результаты,  если подождать больше нескольких минут.

        .model tiny
        .code
        .386                                ; Для команды shld.
        org     100h                        ; COM-программа.
start:
        mov     ax,3508h                    ; AH = 35h,  AL = номер прерывания.
        int     21h                         ; Получить адрес обработчика
        mov     word ptr old_int08h,bx      ; и записать его в old_int08h.
        mov     word ptr old_int08h+2,es
        mov     ax,2508h                    ; AH = 25h,  AL = номер прерывания.
```

```
        mov     dx,offset int08h_handler ; DS:DX - адрес обработчика.
        int     21h                      ; Установить обработчик.
; С этого момента в переменной latency накапливается сумма.
        mov     ah,0
        int     16h                        Пауза до нажатия любой клавиши.

        mov     ax,word ptr latency      ; Сумма в AX.
        cmp     word ptr counter,0       ; Если клавишу нажали немедленно,
        jz      dont_divide              ; избежать деления на ноль.
        xor     dx,dx                    ; DX = 0.
        div     word ptr counter         ; Разделить сумму на число накоплений
dont_divide:
        call    print_ax                 ; и вывести на экран.

        mov     ax,2508h                 ; AH = 25h, AL = номер прерывания.
        lds     dx,dword ptr old_int08h  ; DS:DX = адрес обработчика.
        int     21h                      ; Восстановить старый обработчик.
        ret                              ; Конец программы.

latency         dw      0                ; Сумма задержек.
counter         dw      0                ; Число вызовов прерывания.

; Обработчик прерывания 08h (IRQ0).
; Определяет время, прошедшее с момента срабатывания IRQ0.
int08h_handler  proc    far
        push    ax              ; Сохранить используемый регистр.
        mov     al,0            ; Фиксация значения счетчика в канале 0.
        out     43h,al          ; Порт 43h: управляющий регистр таймера.
; Так как этот канал инициализируется BIOS для 16-битного чтения/записи, другие
; команды не требуются.
        in      al,40h          ; Младший байт счетчика -
        mov     ah,al           ; в AH.
        in      al,40h          ; Старший байт счетчика в AL.
        xchg    ah,al           ; Поменять их местами.
        neg     ax              ; Изменить его знак, так как счетчик уменьшается.
        add     word ptr cs:latency,ax  ; Добавить к сумме.
        inc     word ptr cs:counter     ; Увеличить счетчик накоплений.
        pop     ax
        db      0EAh            ; Команда jmp far.
old_int08h      dd      0       ; Адрес старого обработчика.
int08h_handler  endp

; Процедура print_ax.
; Выводит AX на экран в шестнадцатеричном формате.
print_ax        proc near
        xchg    dx,ax           ; DX = AX.
        mov     cx,4            ; Число цифр для вывода.
shift_ax:
        shld    ax,dx,4         ; Получить в AL очередную цифру.
        rol     dx,4            ; Удалить ее из DX.
        and     al,0Fh          ; Оставить в AL только эту цифру.
```

```
        cmp     al,0Ah          ; Три команды, переводящие
        sbb     al,69h          ; шестнадцатеричную цифру в AL
        das                     ; в соответствующий ASCII-код.
        int     29h             ; Вывод на экран.
        loop    shift_ax        ; Повторить для всех цифр.
        ret
print_ax endp
        end     start
```

Таймер можно использовать для управления динамиком, точных измерений отрезков времени, создания задержек, управления переключением процессов и даже для выбора случайного числа с целью запуска генератора случайных чисел - текущее значение счетчика канала 0 представляет собой идеальный вариант такого начального числа для большинства приложений.

5.10.6. Динамик

Как сказано в разделе 5.10.5, канал 2 системного таймера управляет динамиком компьютера - он генерирует прямоугольные импульсы с частотой, равной 1 193 180/начальное_значение_счетчика. При программировании динамика начальное значение счетчика таймера называется делителем частоты: подразумевается, что динамик работает с частотой 1 193 180/делитель герц. После программирования канала 2 таймера надо еще включить сам динамик. Это осуществляется путем установки битов 0 и 1 порта 61h в 1. Бит 0 фактически разрешает работу данного канала таймера, а бит 1 включает динамик.

```
; Процедура beep.
; Издает звук с частотой 261 Гц (нота "ми" средней октавы)
; длительностью 1/2 секунды на динамике.
beep            proc    near
                mov     al,10110110b    ; Канал 2, режим 3.
                out     43h,al
                mov     al,0Dh          ; Младший байт делителя частоты 11D0h.
                out     42h,al
                mov     al,11h          ; Старший байт делителя частоты.
                out     42h,al
                in      al,61h          ; Текущее состояние порта 61h в AL.
                or      al,00000011b    ; Установить биты 0 и 1 в 1.
                out     61h,al          ; Теперь динамик включен.
                mov     cx,0007h        ; Старшее слово числа микросекунд паузы.
                mov     dx,0A120h       ; Младшее слово числа микросекунд паузы.
                mov     ah,86h          ; Функция 86h:
                int     15h             ; пауза.

                in      al,61h
                and     al,11111100b    ; Обнулить младшие два бита.
                out     61h,al          ; Теперь динамик выключен.
                ret
beep            endp
```

В связи с повсеместным распространением звуковых плат обычный динамик PC сейчас практически никем не используется или используется для выдачи сообщений об ошибках. Вернемся к звуку чуть позже, а пока вспомним, что в разделе 4.7.1 рассматривалось еще одно устройство для определения текущего времени и даты — часы реального времени.

5.10.7. Часы реального времени и CMOS-память

В каждом компьютере есть микросхема, отвечающая за поддержку текущей даты и времени. Для того чтобы значения не сбрасывались при каждом выключении питания, на микросхеме расположена небольшая область памяти (от 64 до 128 байт), выполненная по технологии CMOS, позволяющей снизить энергопотребление до минимума (фактически энергия в таких схемах затрачивается только на зарядку паразитных емкостей при изменении состояния ячеек памяти). Микросхема получает питание от аккумулятора, расположенного на материнской плате, и не отключается при выключении компьютера. Для хранения собственно времени достаточно всего 14 байт такой энергонезависимой памяти, и остальная ее часть используется BIOS, чтобы хранить различную информацию, необходимую для корректного запуска компьютера. Для общения с CMOS и регистрами RTC выделяются порты ввода-вывода от 70h до 7Fh, но только назначение портов 70h и 71h одинаково для всех материнских плат.

Порт 70h для записи: индекс для выбора регистра CMOS:
бит 7: прерывание NMI запрещено на время чтения/записи
бит 6: собственно индекс
Порт 71h для чтения и записи: данные CMOS

После записи в порт 70h нужно осуществить запись или чтение из порта 71h, иначе RTC окажется в неопределенном состоянии. Содержимое регистров CMOS варьируется для разных BIOS, но первые 33h регистра обычно выполняют следующие функции:

00h: RTC: текущая секунда (00-59h или 00-3Bh) - формат выбирается регистром 0Bh, по умолчанию - BCD
01h: RTC: секунды будильника (00–59h или 00–3Bh или 0FFh (любая секунда))
02h: RTC: текущая минута (00–59h или 00-3Bh)
03h: RTC: минуты будильника (00–59h или 00-3Bh или FFh)
04h: RTC: текущий час:
00-23h/00-17h (24-часовой режим)
01-12h/01-lCh (12-часовой режим до полудня)
81h–92h/81–8Ch (12-часовой режим после полудня)
05h: RTC: часы будильника (то же или FFh, если любой час)
06h: RTC: текущий день недели (1–7, 1 - воскресенье)
07h: RTC: текущий день месяца (01–31h/01h–1Fh)
08h: RTC: текущий месяц (01–12h/01–0Ch)
09h: RTC: текущий год (00-99h/00-63h)

0Ah: RTC: регистр состояния A
бит 7: 1 - часы заняты (происходит обновление)
биты 4-6: делитель фазы (010 - 32 768 кГц - по умолчанию)
биты 3-0: выбор частоты периодического прерывания:
0000 - выключено
0011 - 122 мкс (минимум)
1111 – 500мс
0110 – 976,562мкс (1024 Гц)
0Bh: RTC: регистр состояния B
бит 7: запрещено обновление часов (устанавливают перед записью новых значений в регистры даты и часов)
бит 6: вызов периодического прерывания (IRQ8)
бит 5: вызов прерывания при срабатывании будильника
бит 4: вызов прерывания по окончании обновления времени
бит 3: включена генерация прямоугольных импульсов
бит 2: 1/0: формат даты и времени двоичный/BCD
бит 1: 1/0: 24-часовой/12-часовой режим
бит 0: автоматический переход на летнее время в апреле и октябре
0Ch только для чтения: RTC: регистр состояния C
бит 7: произошло прерывание
бит 6: разрешено периодическое прерывание
бит 5: разрешено прерывание от будильника
бит 4: разрешено прерывание по окончании обновления часов
ODh только для чтения: регистр состояния D
бит 7: питание RTC/CMOS есть
0Eh: результат работы POST при последнем старте компьютера:
бит 7: RTC сбросились из-за отсутствия питания CMOS
бит 6: неверная контрольная сумма CMOS-конфигурации
бит 5: неверная конфигурация
бит 4: размер памяти не совпадает с записанным в конфигурации
бит 3: ошибка инициализации первого жесткого диска
бит 2: RTC-время установлено неверно (например, 30 февраля)
OFh: состояние, в котором находился компьютер перед последней перезагрузкой
00h - **Ctr-Alt-Del**, 05h - INT 19h, OAh, OBh, 0Ch - jmp, iret, retf на адрес, хранящийся в 0040h:0067h. Другие значения указывают, что перезагрузка произошла в ходе POST или в других необычных условиях
10h: тип дисководов (биты 7-4 и 3-0 - типы первого и второго дисковода)
0000b - отсутствует
0001b – 360 Кб
0010b - 1,2 Мб
0011b – 720 Кб
0100b - 1,44 Мб
0101b - 2,88 Мб

12h: тип жестких дисков (биты 7-4 и 3-0 - типы первого и второго жестких дисков, 1111b, если номер типа больше 15)
14h: байт состояния оборудования
биты 7-6: число установленных жестких дисков минус один
биты 5-4: тип монитора (00, 01, 10, 11 = EGA/VGA, 40x25 CGA, 80x25 CGA, MDA)
бит 3: монитор присутствует
бит 2: клавиатура присутствует
бит 1: FPU присутствует
бит 0: дисковод присутствует
15h: младший байт размера базовой памяти в килобайтах (80h)
16h: старший байт размера базовой памяти в килобайтах (02h)
17h: младший байт размера дополнительной памяти (выше 1 Мб) в килобайтах
18h: старший байт размера дополнительной памяти (выше 1 Мб) в килобайтах
19h: тип первого жесткого диска, если больше 15
1Ah: тип второго жесткого диска, если больше 15
2Eh: старший байт контрольной суммы регистров 10h - 2Dh
2Fh: младший байт контрольной суммы регистров 10h - 2Dh
30h: младший байт найденной при POST дополнительной памяти в килобайтах
31h: старший байт найденной при POST дополнительной памяти в килобайтах
32h: первые две цифры года в BCD-формате

Данные о конфигурации, хранящиеся в защищенной контрольной суммой области, бывают нужны достаточно редко, а для простых операций с часами реального времени и будильником удобно использовать прерывание BIOS 1Ah. Однако, программируя RTC на уровне портов, можно активизировать периодическое прерывание - режим, в котором RTC вызывает прерывание IRQ8 с заданной частотой, что позволяет оставить IRQ0 для работы системы, если вас удовлетворяет ограниченный выбор частот периодического прерывания. В качестве примера посмотрим, как выполняются чтение и запись в CMOS-память.

```
; rtctime.asm
; Вывод на экран текущей даты и времени из RTC.
;
        .model  tiny
        .code
        .186                    ; Для shr al,4.
        org     100h            ; COM-программа.
start:
        mov     al,0Bh          ; CMOS 0Bh - управляющий регистр B.
        out     70h,al          ; Порт 70h - индекс CMOS.
        in      al,71h          ; Порт 71h - данные CMOS.
        and     al,11111011b    ; Обнулить бит 2 (форма чисел - BCD)
        out     71h,al          ; и записать обратно.

        mov     al,32h          ; CMOS 32h - две старшие цифры года.
        call    print_cmos      ; Вывод на экран.
        mov     al,9            ; CMOS 09h - две младшие цифры года.
```

```
        call    print_cmos
        mov     al,'-'          ; Минус.
        int     29h             ; Вывод на экран.
        mov     al,8            ; CMOS 08h - текущий месяц.
        call    print_cmos
        mov     al,'-'          ; Еще один минус.
        int     29h
        mov     al,7            ; CMOS 07h - день.
        call    print_cmos
        mov     al,' '          ; Пробел.
        int     29h
        mov     al,4            ; CMOS 04h - час.
        call    print_cmos
        mov     al,'h'          ; Буква "h".
        int     29h .
        mov     al,' '          ; Пробел.
        int     29h
        mov     al,2            ; CMOS 02h - минута.
        call    print_cmos
        mov     al,':'          ; Двоеточие.
        int     29h
        mov     al,0h           ; CMOS 00h - секунда.
        call    print_cmos
        ret

; Процедура print_cmos.
; Выводит на экран содержимое ячейки CMOS с номером в AL.
; Считает, что число, читаемое из CMOS, находится в формате BCD.
print_cmos      proc near
        out     70h,al          ; Послать AL в индексный порт CMOS.
        in      al,71h          ; Прочитать данные.
        push    ax
        shr     al,4            ; Выделить старшие четыре бита.
        add     al,'0'          ; Добавить ASCII-код цифры 0.
        int     29h             ; Вывести на экран.
        pop     ax
        and     al,0Fh          ; Выделить младшие четыре бита.
        add     al,30h          ; Добавить ASCII-код цифры 0.
        int     29h             ; Вывести на экран.
        ret
print_cmos      endp
        end     start
```

5.10.8. Звуковые платы

Звуковые платы, совместимые с теми или иными моделями Sound Blaster, выглядят как четыре независимых устройства:

- DSP (Digital Signal Processor) - устройство, позволяющее выводить и считывать оцифрованный звук;

а микшер (Mixer) - система регуляторов громкости для всех каналов платы;

- FM (Frequency Modulation) или AdLib (по названию первой звуковой платы) - устройство, позволяющее синтезировать звук из синусоидальных и треугольных волн. Слова типа OPL2 или OPL3 в описании платы - это и есть номера версии используемого FM-синтезатора;
- MIDI (Music Instrumental Digital Interface) - стандартный интерфейс передачи данных в музыкальной аппаратуре. Но в нашем случае рассматривается GMIDI (обобщенный MIDI) - более качественная система генерации музыки, в которой используются не искусственные синусоидальные сигналы, а образцы звучания различных инструментов. К сожалению, качество этих образцов в большинстве дешевых плат оставляет желать лучшего.

Номера портов ввода-вывода, предоставляющих доступ ко всем этим устройствам, отсчитываются от базового порта, обычно равного 220h, но допускаются также конфигурации с 210h, 230h, 240h, 250h, 260h и 280h. Кроме того, интерфейс GMIDI использует другую серию портов, которая может начинаться как с 300h, так и с 330h. В описаниях портов мы будем считать, что базовыми являются 220h и 300h. Область портов интерфейса с AdLib начинается с 388h.

Существует большое число модификаций плат Sound Blaster, отличающихся, помимо всего прочего, набором поддерживаемых команд и портов ввода-вывода. После названия каждой команды или порта мы будем указывать сокращенное название платы, начиная с которой эта команда или порт поддерживается:

- SB – Sound Blaster 1.0;
- a SB2 - Sound Blaster 2.0;
- SBPro - Sound Blaster Pro;
- SBPro2 - Sound Blaster Pro2;
- SB16 - Sound Blaster 16;
- ASP - Sound Blaster 16 ASP;
- AWE - Sound Blaster AWE32.

Программирование DSP

Цифровой процессор - наиболее важная часть звуковой платы. Именно с его помощью осуществляется вывод обычного оцифрованного звука, так же как и запись звука из внешнего источника в файл. Для своей работы, помимо описываемых в этом разделе портов, DSP использует прерывания и контроллер прямого доступа к памяти DMA. Программирование DMA с примером программы, использующей его для воспроизведения звука, мы рассмотрим в разделе 5.10.9.

DSP обслуживается при помощи следующих портов:

226h для записи: сброс DSP (SB)

Запись в этот порт осуществляет полную переинициализацию DSP, прерывая все происходящие процессы. Операцию сброса DSP необходимо выполнить, по крайней мере, один раз после перезагрузки системы, чтобы его можно было использовать.

Процедура сброса осуществляется следующим образом:

1. В порт 226h записывается число 1 (начало инициализации).
2. Выдерживается пауза как минимум 3,3 мкс.

3. В порт 226h записывается число 0 (конец инициализации).
4. Выдерживается пауза максимум 100 мкс. В течение паузы можно выполнять чтение порта 22Eh. Когда в считываемом числе будет установлен бит 7 (данные готовы), можно сразу переходить к пункту 5. В противном случае имеет смысл повторить процедуру, используя другой базовый порт.
5. Выполняется чтение из порта 22Ah. Если считанное число равно 0AAh - DSP был успешно инициализирован. В противном случае допускается вернуться к пункту 4, но по истечении 100 мкс после записи в 226h можно будет с уверенностью сказать, что DSP с базовым адресом 220h не существует или не работает.

22Ah для чтения: чтение данных из DSP (SB)

Чтение из этого порта используется для передачи всех возможных данных от DSP программам. Процедура чтения состоит из двух шагов:

1. Выполнять цикл чтения порта 22Eh, пока бит 7 считываемого байта не окажется равным единице.
2. Выполнить чтение из порта 22Ah.

22Ch для записи: запись данных и команд DSP (начиная с SB)

Этот единственный порт используется для передачи всего множества команд DSP и для пересылки в него данных (аргументов команд). Процедура записи:

1. Выполнять цикл чтения порта 22Ch, пока бит 7 считываемого байта не окажется равным нулю.
2. Выполнить запись в порт 22Ch.

22Ch для чтения: готовность DSP для приема команды (SB)

Если при чтении из этого порта бит 7 сброшен в ноль - DSP готов к приему очередного байта в порт 22Ch на запись. Значение остальных битов не определено.

22Eh для чтения: готовность DSP для посылки данных (начиная с SB)

Если при чтении из этого порта бит 7 установлен в единицу - DSP готов к передаче через порт 22Ah очередного байта.

22Eh для чтения (тот же порт!): подтверждение обработки 8-битного прерывания (SB)

Обработчик прерывания, сгенерированного звуковой платой по окончании 8-битной операции, обязательно должен выполнить одно чтение из этого порта перед завершением (помимо обычной процедуры посылки сигнала EOI в соответствующий контроллер прерываний).

22Fh для чтения: подтверждение обработки 16-битного прерывания (SB16)

Обработчик прерывания, сгенерированного звуковой платой по окончании 16-битной операции, обязательно должен выполнить одно чтение из этого порта перед завершением (помимо обычной процедуры посылки сигнала EOI в соответствующий контроллер прерываний).

Теперь рассмотрим команды DSP. Все они пересылаются в звуковую плату через порт 22Ch, как описано выше. После команды могут следовать аргументы,

которые передаются таким же образом (включая ожидание готовности к приему команды).

04h: состояние DSP (устаревшая) (SB2 - SBPro2)
Возвращает информацию о текущей операции DSP:
бит 0: динамик включен
бит 1: стерео АЦП включен
бит 2: всегда О
бит 3: происходит прямое воспроизведение 8-битного PCM
бит 4: происходит воспроизведение 2-битного ADPCM через DMA
бит 5: происходит воспроизведение 2,6-битного ADPCM через DMA
бит 6: происходит воспроизведение 4-битного ADPCM через DMA
бит 7: происходит воспроизведение 8-битного PCM через DMA

10h, NN: прямое воспроизведение 8-битного звука (SB)
Выводит очередной байт (NN) из несжатого 8-битного оцифрованного звука на воспроизведение. При использовании этого способа воспроизведения сама программа должна заботиться о том, чтобы новые данные всегда были наготове (то есть не считывать их с диска в ходе работы) и чтобы байты пересылались в DSP с необходимой частотой. (В этом режиме поддерживаются частоты до 23 кГц.) Процедура вывода проста:

1. Вывести в DSP команду 10h и очередной байт из оцифровки.
2. Подождать необходимое время и вернуться к пункту 1.

Чтобы выполнять пересылку байтов с заданной частотой, обычно перепрограммируют системный таймер, как будет показано в конце этого раздела. Но из-за ограничений по качеству звука и высокой ресурсоемкости такой способ воспроизведения практически не используется.

14h, LO, HI: прямое воспроизведение 8-битного PCM через DMA (SB)
Начинает процесс воспроизведения данных, на которые настроен соответствующий канал DMA (см. раздел 5.10.9):

1. Установить обработчик прерывания от звуковой платы (и разрешить его в контроллере прерываний).
2. Выполнить команду 40h или другим образом установить частоту оцифровки.
3. Выполнить команду 0D1h (включить динамик).
4. Настроить DMA (режим 48h + номер канала).
5. Выполнить команду 14h. Аргументы LO и HI - это младший и старший байты длины проигрываемого участка минус один.
6. Из обработчика прерывания подтвердить его чтением порта 22Eh и посылкой байта 20h в соответствующий контроллер прерываний.
7. Выполнить команду 0D3h (выключить динамик).

На платах, начиная с SB16, для этого режима рекомендуется пользоваться командами ОС?h.

16h, LO, HI: прямое воспроизведение 2-битного ADPCM через DMA (SB)
Начинает процесс воспроизведения данных аналогично команде 14h, но они должны храниться в сжатом формате Creative ADPCM 2 bit. Длина, указываемая в качестве аргументов этой команды, равна (число_байтов + 2)/4. В качестве нулевого байта в процедуре распаковки ADPCM используется значение, которое применялось последней командой 17h. В остальном процедура воспроизведения аналогична команде 14h.

17h, LO, HI: прямое воспроизведение 2-битного ADPCM через DMA с новым нулевым байтом (SB)
То же самое, что и 16h, но первый байт из данных будет рассматриваться как нулевой байт для процедуры распаковки ADPCM.

1Ch: воспроизведение 8-битного PCM через DMA с автоинициализацией (SB2)
Начинает режим воспроизведения с автоинициализацией - лучший из режимов, предлагаемых звуковыми платами. В этом режиме DSP воспроизводит в цикле содержимое указанного участка памяти, мгновенно возвращаясь на начало, пока он не будет остановлен командой 0DAh или новой командой воспроизведения через DMA. Весь секрет заключается в том, что плата генерирует прерывание не только при достижении конца блока, но и при достижении его середины. Таким образом, пока DSP проигрывает вторую половину буфера, мы можем прочитать следующие несколько килобайтов в первую половину, не останавливая воспроизведение ни на миг:

1. Установить обработчик прерывания звуковой платы и разрешить его в контроллере прерываний.
2. Выполнить команду 40h или другим образом установить частоту оцифровки.
3. Выполнить команду 48h
   (установить размер DMA-буфера = (число_байтов + 1)/2 — 1).
4. Выполнить команду 0D1h (включить динамик).
5. Настроить DMA (режим 58h + номер канала).
6. Выполнить команду 1Ch.
7. В обработчике прерывания: заполнить следующую половину буфера.
8. В обработчике прерывания: подтвердить прерывание чтением из 22Eh и записью 20h в контроллер прерываний.
9. Подождать, пока не будут воспроизведены все данные.
10. Выполнить команду OD3h (выключить динамик).
11. Выполнить команду 0D0h (остановить 8-битную DMA-передачу).
12. Выполнить команду ODAh (завершить режим автоинициализации).
13. Выполнить команду ODOh (остановить 8-битную DMA-передачу).

На платах, начиная с SB16, для этого режима рекомендуется пользоваться командами OC?h.

1Fh: воспроизведение 2-битного ADPCM через DMA с автоинициализацией (SB2)

Аналог команды 1Ch, но данные хранятся в 2-битном формате ADPCM с нулевым байтом. Длина блока рассчитывается так:

длина = (число_байтов + 3)/4 + 1
длина блока = (длина + 1)/2 - 1

20h: прямое чтение 8-битных данных из АЦП (SB)
Команда предназначена для чтения оцифрованного звука из внешнего источника. Используется следующая процедура:

1. Выполнить команду 20h.
2. Прочитать очередной байт.
3. Подождать необходимое время и вернуться к пункту 1.

Проблемы с этой командой точно такие же, как и с 10h.

24h, **LO, HI:** чтение 8-битного PCM через DMA (SB)
Аналог команды 14h, но выполняет не воспроизведение, а запись звука. Последовательность действий идентична случаю с 14h, но используемый режим DMA - 44h + номер канала.

2Ch: запись 8-битного PCM через DMA с автоинициализацией (SB2)
Аналог команды 1Ch, но выполняет не воспроизведение, а запись звука. Последовательность действий идентична случаю с 1Ch, но используемый режим DMA - 54h + номер канала.

30h: прямое чтение MIDI (SB)
Выполняет чтение очередного MIDI-события:

1. Выполнить команду 30h.
2. Прочитать MIDI-событие (до 64 байт).

31h: чтение MIDI с прерыванием (SB)
Включает генерацию прерывания от звуковой платы при поступлении нового MIDI-события. Для этого необходимо:

1. Установить обработчик прерывания.
2. Выполнить команду 31h.
3. В обработчике прерывания: прочитать MIDI-событие.
4. В обработчике прерывания: подтвердить прерывание чтением из 22Eh и записью 20h в контроллер прерываний.
5. Выполнить команду 31h еще раз, чтобы отменить генерацию прерывания.

32h: прямое чтение MIDI-события с дельта-временем (SB)
Выполняет чтение очередного MIDI-события и 24-битного дельта-времени, то есть времени в микросекундах, наступившего после предшествующего MIDI-события. (Считываются данные в следующем порядке: младший байт времени, средний байт времени, старший байт времени, MIDI-команда.) Именно в виде последовательности MIDI-событий, перед каждым из которых указано дельта-время, и записывается музыка в MIDI-файлах.

32h: чтение MIDI-события с дельта-временем с прерыванием (SB)
Включает/выключает генерацию прерываний от звуковой платы аналогично команде 31h, но при чтении MIDI-события передаются вместе с дельта-временами, как в команде 32h.

34h: режим прямого доступа к UART (SB2)
Отключает DSP, после чего все команды записи/чтения в его порты (используя тот же механизм проверки готовности) рассматриваются как MIDI-события. Вывести DSP из этого режима можно только с помощью полной переинициализации.

37h: режим прямого доступа к UART с прерыванием (SB2)
Переключает порты DSP на UART аналогично команде 34h, но каждый раз, когда новое MIDI-событие готово для чтения, вызывается прерывание звуковой платы.

38h, MIDI: прямая запись MIDI (SB)
Посылает одно MIDI-событие.

40h, TC: установить временную константу (SB)
Устанавливает частоту оцифровки, используя однобайтную константу, рассчитываемую следующим образом:

TC = 256 - (1000000 / (число_каналов X частота)),

где число_каналов - 1 для моно и 2 для стерео.

41h, LO, HI: установить частоту оцифровки (SB16)
Аналогично 40h, но указывается истинное значение частоты (сначала младший, потом старший байты). Число каналов определяется автоматически. Реальная частота тем не менее округляется до ближайшего возможного значения TC.

45h: продолжить остановленное 8-битное воспроизведение через DMA (SB16)
Продолжает остановленное командой ODAh воспроизведение 8-битного звука через DMA с автоинициализацией.

47h: продолжить остановленное 16-битное воспроизведение через DMA (SB16)
Продолжает остановленное командой 0D9h воспроизведение 16-битного звука через DMA с автоинициализацией.

48h, LO, HI: установить размер буфера DMA (SB2)
Устанавливает число байтов минус один для следующей команды передачи через DMA (сначала младший байт, затем старший).

74h, LO, HI: прямое воспроизведение 4-битного ADPCM через DMA (SB)
Аналог 16h, но используется 4-битный вариант формата Creative ADPCM.

75h, LO, HI: прямое воспроизведение 4-битного ADPCM через DMA с новым нулевым байтом (SB)
Аналог 17h, но используется 4-битный вариант формата Creative ADPCM.

76h, LO, HI: прямое воспроизведение 2,6-битного ADPCM через DMA (SB)
Аналог 16h, но используется 2,6-битный вариант формата Creative ADPCM.

77h, **LO, HI:** прямое воспроизведение 2,6-битного ADPCM через DMA с новым нулевым байтом (SB)
Аналог 17h, но используется 2,6-битный вариант формата Creative ADPCM.

7Dh: воспроизведение 4-битного ADPCM через DMA с автоинициализацией (SB2)
Аналог 1Fh, но используется 4-битный вариант формата Creative ADPCM.

7Fh: воспроизведение 2,6-битного ADPCM через DMAс автоинициализацией (SB2)
Аналог 1Fh, но используется 2,6-битный вариант формата Creative ADPCM.

80h, **LO, HI:** заглушить DSP (SB)
Вывести указанное число байтов тишины с текущей частотой оцифровки.

OB?h/OC?h MODE, LO, HI: обобщенный интерфейс к DSP (SB16)
Команды OB?h используются для 16-битных операций, команды OC?h - для 8-битных. Младшие четыре бита определяют режим:

бит 0: всегда О
бит 1: используется FIFO
бит 2: используется автоинициализация DMA
бит 3: направление передачи (0 - воспроизведение, 1 - оцифровка)

Аргументы этой команды - режим, младший байт длины, старший байт длины (перед указанной командой не требуется устанавливать размер DMA-буфера специально).

В байте режима определены всего два бита (остальные должны быть равны нулю):

бит 4: данные рассматриваются как числа со знаком
бит 5: режим стерео

Длина во всех случаях равна числу байтов минус один для 8-битных операций и числу слов минус один для 16-битных.

0D0h: остановить 8-битную DMA-операцию (SB)
Останавливает простую (без автоинициализации) 8-битную DMA-операцию.

0D1h: включить динамик (SB)
Разрешает работу выхода на динамик (колонки и т. д.).
После сброса DSP этот канал выключен.

OD3h: выключить динамик (SB)
Отключает выход на динамик (колонки и т. д.).

OD4h: продолжить 8-битную DMA-операцию (SB)
Продолжает DMA-операцию, остановленную командой ODOh.

OD5h: остановить 16-битную DMA-операцию (SB)
Останавливает простую (без автоинициализации) 16-битную DMA-операцию.

OD6h: продолжить 16-битную DMA-операцию (SB)
Продолжает DMA-операцию, остановленную командой OD5h.

0D8h: определить состояние динамика (SB)
Возвращает 00h, если динамик выключен; 0FFh, если включен.

OD9h: завершить 16-битную DMA-операцию с автоинициализацией (SB16)
Эта команда завершает операцию только после окончания воспроизведения текущего блока. Для немедленного прекращения воспроизведения необходимо выполнить последовательно команды OD3h, OD5h, OD9h и OD5h.

0DAh: завершить 8-битную DMA-операцию с автоинициализацией (SB2)
Аналог OD9h, но для 8-битных операций.

0E0h, BYTE: проверка наличия DSP на этом порту (SB2)
Любой байт, посланный как аргумент к этой команде, возвращается при чтении из DSP в виде своего побитового дополнения (DSP выполняет над ним операцию NOT).

0E1h: определение номера версии DSP (SB)
Возвращает последовательно старший и младший номера версии DSP:

1.?- SB
2.0-SB2
3.0 - SBPro
3.? - SBPro2
4.0? - SB16
4.11 – SB16SCSI-2
4.12 - AWE32

OE3h: чтение Copyright DSP (SBPro2)
Возвращает ASCIZ-строку с информацией Copyright данной платы.

OE4h, **BYTE:** запись в тестовый регистр (SB2)
Помещает байт в специальный неиспользуемый регистр, который сохраняется даже после переинициализации DSP.

OE8h: чтение из тестового регистра (SB2)
Возвращает байт, помещенный ранее в тестовый регистр командой OE4h.

0F0h: генерация синусоидального сигнала (SB)
Запускает DSP на воспроизведение синусоидального сигнала с частотой около 2 кГц, который можно прервать только сбросом DSP.

OF2h: запрос на прерывание в 8-битном режиме (SB)
Генерирует прерывание от звуковой карты. В качестве подтверждения от обработчика ожидается чтение из порта 22Eh.

OF3h: запрос на прерывание в 16-битном режиме (SB)
Генерирует прерывание от звуковой карты. В качестве подтверждения от обработчика ожидается чтение из порта 22Fh.

0FBh: состояние DSP (SB16)
Возвращает байт состояния текущей DSP-операции:

бит 0: 8-битное воспроизведение через DMA
бит 1: 8-битное чтение через DMA

бит 2: 16-битное воспроизведение через DMA
бит 3: 16-битное чтение через DMA
бит 4: динамик включен
биты 5-6: не определены
бит 7: TC модифицирована (может быть ноль, если предыдущая команда 40h пыталась установить неподдерживаемую частоту)

0FCh: дополнительная информация (SB16)
Возвращает дополнительный байт состояния текущей DMA-операции:
бит 1: синхронный режим (одновременная запись и воспроизведение)
бит 2: 8-битный режим с автоинициализацией
бит 4: 16-битный режим с автоинициализацией

OFDh: последняя выполненная команда (SB16)
Возвращает последнюю успешную команду DSP.

Программирование микшера

Это устройство предназначено для регулирования громкости на всех каналах, используемых звуковой платой.

Для управления микшером служат всего два порта ввода-вывода.

224h для записи: выбор регистра микшера (SBPro)
Запись в этот порт выбирает номер регистра, к которому будет осуществляться доступ при последующих обращениях к порту 225h.

225h для чтения и записи: чтение/запись регистра микшера (SBPro)
Чтение и запись в этот порт приводят к чтению и записи в соответствующий регистр микшера. Рассмотрим их назначение.

Регистр **00h** для записи: сброс и инициализация (SBPro)
Выбор этого регистра в порту 224h начинает инициализацию. Следует подождать как минимум 100 мкс, а затем записать в порт 225h число 01h (команда «завершить инициализацию»).

Регистр Olh для чтения: состояние микшера (SBPro)
Возвращает последний номер регистра.

Регистр 04h для чтения и записи: уровень ЦАП (SBPro)
биты 4-0: уровень правого ЦАП
биты 7-4: уровень левого ЦАП

Регистр **0Ah** для чтения и записи: уровень микрофона (SBPro)
биты 2-0: уровень микрофона

Регистр 22h для чтения и записи: общий уровень (SBPro)
биты 4-0: правый общий уровень
биты 7-4: левый общий уровень

Регистр **26h** для чтения и записи: уровень FM (SBPro)
биты 4-0: правый уровень FM
биты 7-4: левый уровень FM

Регистр **28h** для чтения и записи: уровень CD audio (SBPro)
биты 4-0: правый уровень CD audio
биты 7-4: левый уровень CD audio

Регистр **2Eh** для чтения и записи: уровень линейного входа (SBPro)
биты 4–0: уровень правого линейного входа
биты 7-4: уровень левого линейного входа

Регистр **30h** для чтения и записи: левый общий уровень (SB16)
биты 7–1: левый общий уровень

Регистр **31h** для чтения и записи: правый общий уровень (SB16)
биты 7–1: правый общий уровень

Регистр **32h** для чтения и записи: левый уровень ЦАП (SB16)
биты 7-1: левый уровень ЦАП

Регистр **33h** для чтения и записи: правый уровень ЦАП (SB16)
биты 7-1: правый уровень ЦАП

Регистр **34h** для чтения и записи: левый уровень FM (SB16)
биты 7-1: левый уровень FM

Регистр **35h** для чтения и записи: правый уровень FM (SB16)
биты 7-1: правый уровень FM

Регистр **36h** для чтения и записи: левый уровень CD audio (SB16)
биты 7-1: левый уровень CD audio

Регистр 37h для чтения и записи: правый уровень CD audio (SB16)
биты 7-1: правый уровень CD audio

Регистр **38h** для чтения и записи: левый уровень линейного входа (SB16)
биты 7-1: левый уровень линейного входа

Регистр **39h** для чтения и записи: правый уровень линейного входа (SB16)
биты 7-1: правый уровень линейного входа

Регистр **3Ah** для чтения и записи: уровень микрофона (SB16)
биты 7-3: уровень микрофона

Регистр **3Bh** для чтения и записи: уровень динамика PC (SB16)
биты 7-5: уровень динамика PC

Регистр **3Ch** для чтения и записи: управление выводом (SB16)
бит 0: микрофон включен
бит 1: правый канал CD audio включен
бит 2: левый канал CD audio включен

бит 3: правый канал линейного входа включен
бит 4: левый канал линейного входа включен
биты 7-5: нули

Регистр 3Dh для чтения и записи: управление левым каналом ввода (SB16)
бит 0: микрофон включен
бит 1: правый канал CD audio включен
бит 2: левый канал CD audio включен
бит 3: правый канал линейного входа включен
бит 4: левый канал линейного входа включен
бит 5: правый канал FM включен
бит 6: левый канал FM включен
бит 7: нуль

Регистр 3Eh для чтения и записи: управление правым каналом ввода (SB16)
бит 0: микрофон включен
бит 1: правый канал CD audio включен
бит 2: левый канал CD audio включен
бит 3: правый канал линейного входа включен
бит 4: левый канал линейного входа включен
бит 5: правый канал FM включен
бит 6: левый канал FM включен
бит 7: нуль

Регистр 3Fh для чтения и записи: уровень усиления в левом канале ввода (SB16)
биты 7-5: усиление в левом канале ввода

Регистр 40h для чтения и записи: уровень усиления в правом канале ввода (SB 16)
биты 7-5: усиление в правом канале ввода

Регистр 41h для чтения и записи: уровень усиления в левом канале вывода (SB16)
биты 7-5: усиление в левом канале вывода

Регистр 42h для чтения и записи: уровень усиления в правом канале вывода (SB16)
биты 7-5: усиление в правом канале вывода

Регистр 43h для чтения и записи: автоматическая подстройка усиления (SB16)
бит 0: автоматическая подстройка усиления включена

Регистр 44h для чтения и записи: уровень усиления высоких частот слева (SB16)
биты 7-4: усиление высоких частот слева

Регистр 45h для чтения и записи: уровень усиления высоких частот справа (SB16)
биты 7-4: усиление высоких частот справа

Регистр 46h для чтения и записи: уровень усиления басов слева (SB16)
биты 7-4: усиление басов слева

Регистр 47h для чтения и записи: уровень усиления басов справа (SB16)
биты 7-4: усиление басов справа

Регистр 80h для чтения и записи: выбор прерывания (SB16)
бит 0: IRQ2
бит 1: IRQ5
бит 2: IRQ7
бит 3: IRQ10
биты 7-4: всегда 1

Эта установка сохраняется при сбросе микшера или даже при «горячей» перезагрузке компьютера.

Регистр 81h для чтения и записи: выбор DMA (SB16)
бит 0: 8-битный DMAO
бит 1: 8-битный DMA1
бит 2: O
бит 3: 8-битный DMA3
бит 4: O
бит 5: 16-битный DMA5
бит 6: 16-битный DMA6
бит 7: 16-битный DMA7

Данная установка сохраняется при сбросе микшера или даже при «горячей» перезагрузке компьютера.

Регистр 82h для чтения: состояние прерывания (SB16)
бит 0: происходит обработка прерывания от 8-битной операции
бит 2: происходит обработка прерывания от 16-битной операции
бит 3: происходит обработка прерывания операции MPU-401
биты 7–4: зарезервированы

Частотный синтез (программирование AdLib)

Синтезатор, расположенный на звуковой плате и отвечающий за FM-музыку, управляется тремя портами - портом состояния, выбора регистра и портом данных. Для совместимости с различными платами они дублируются несколько раз.

228h, 388h для чтения: порт состояния FM (SB)
220h для чтения: порт состояния левого канала FM (SBPro)
222h, 38Ah для чтения: порт состояния правого канала FM (SBPro)
Определяет, закончился ли отсчет FM-таймеров и какого именно.
биты 4-0: всегда O
бит 5: таймер 2 (период - 230 мкс) сработал
бит 6: таймер 1 (период - 80 мкс) сработал
бит 7: один из таймеров сработал

228h, 388h для записи: выбор регистра FM (SB)
220h для записи: выбор регистра левого канала FM (SBPro)
222h, 38Ah для записи: выбор регистра правого канала FM (SBPro)

Запись числа в этот порт выбирает, с каким регистром синтезатора будут работать последующие команды записи в порт данных.

Процедура записи числа в регистр FM выглядит следующим образом:

1. Выполнить запись в порт выбора регистра.
2. Подождать как минимум 3,3 мкс.
3. Выполнить запись в порт данных.
4. Подождать как минимум 23 мкс перед любой другой операцией со звуковой платой.

На современных платах эти паузы можно сделать меньше. Так, если используется синтезатор OPL3, требуемое значение пауз - 0,1 и 0,28 мкс.

229h, 389h для записи: запись в регистр FM (SB)
221h для записи: запись в регистр левого канала FM (SBPro)
223h, 38Bh для записи: запись в регистр правого канала FM (SBPro)

Всего в этих синтезаторах доступно 244 регистра - от 01h до 0F5h. Рассмотрим наиболее полезные.

Регистр **Olh:** тестовый регистр
биты 4–0: O
бит 5: FM микросхема контролирует форму волны
биты 7–6: 0

Регистр **02h:** счетчик первого таймера
Значение этого счетчика увеличивается на единицу каждые 80 мкс. Когда регистр переполняется, вырабатывается прерывание таймера IRQ0 и устанавливаются биты 7 и 6 в порту состояния.

Регистр **03h:** счетчик второго таймера
Значение этого счетчика увеличивается на единицу каждые 320 мкс. Когда регистр переполняется, вырабатывается прерывание таймера IRQ0 и устанавливаются биты 7 и 5 в порту состояния.

Регистр **04h:** регистр управления таймером
бит 0: установка бита запускает первый таймер (с начальным значением счетчика, помещенным в регистр 02h)
бит 1: установка бита запускает второй таймер (с начальным значением счетчика, помещенным в регистр 03h)
биты 4-2: зарезервированы
бит 5: маска второго таймера - установка этого бита запрещает включение второго таймера при установке бита 1
бит 6: маска первого таймера - установка этого бита запрещает включение первого таймера при установке бита O
бит 7: сброс флагов для обоих таймеров

Регистр **08h:** включение режимов CSM и Keysplit
биты 5-0: зарезервированы
бит 6: включение режима keysplit
бит 7: 1 - режим CSM; 0 - режим FM

Режим CSM (синусоидально-волновой синтез речи) применялся на платах AdLib без DSP для воспроизведения оцифрованного звука (с очень плохим качеством).

Следующие регистры описываются группами - по одному на голос.

Регистры **20h** - 35h: различные настройки режимов
- биты 3-0: какая гармоника будет создавать звук (или модуляцию) относительно точно установленной частоты голоса:
  - O - одной октавой ниже
  - 1 – сточно установленной частотой голоса
  - 2 — одной октавой выше
  - 3 - октавой и квинтой выше
  - 4 - двумя октавами выше
  - 5 - двумя октавами и большой терцией выше
  - 6 - двумя октавами и квинтой выше
  - 7 - двумя октавами и малой септимой выше
  - 8 - тремя октавами выше
  - 9 - тремя октавами и большой секундой выше
  - A, B - тремя октавами и большой терцией выше
  - C, D - тремя октавами и квинтой выше
  - E, F - тремя октавами и большой септимой выше
- бит 4: режим keyboard scaling - если он выбран, длина звука сокращается и он повышается
- бит 5: продление стадии поддержки звука (то есть он не затухает сразу после нарастания)
- бит 6: включает режим вибрато (его глубина контролируется флагом в регистре 0BDh)
- бит 7: применение амплитудной модуляции (ее глубина контролируется флагом в регистре OBDh)

Регистры 40h - 55h: управление уровнями выхода
- биты 5-0: общий выходной уровень канала (00000 - максимум, 11111 - минимум)
- биты 7-6: понижение выходного уровня с повышением частоты:
  - 00 - не понижать
  - 10 - 1,5 дБ на октаву
  - 01 - 3 дБ на октаву
  - 11 - 7 дБ на октаву

Регистры 60h - 75h: темп нарастания/спада
- биты 3-0: темп спада (0 - минимум, F - максимум)
- биты 7-4: темп нарастания (0 - минимум, F - максимум)

Регистры 80h - 95h: уровень поддержки/темп отпускания
- биты 3-0: темп отпускания (0 - минимум, F - максимум)
- биты 7-4: уровень поддержки (0 - минимум, F - максимум)

Регистры **0A0h** - OA8h: нота (младший байт)
биты 7-0: биты 7-0 номера ноты

Регистры **0B0h** - OB8h: октава и нота (старшие биты)
биты 1-0: биты 9-8 номера ноты
биты 4-2: номер октавы
бит 5: включить звук в этом канале

Значение 10-битного номера ноты соответствует следующим реальным нотам (для четвертой октавы):
016Bh - C# (277,2 Гц)
0181h – D (293,7 Гц)
0198h – D# (311,1 Гц)
01B0h - E (329,6 Гц)
01CAh – F (349,2 Гц)
01E5h - F# (370,0 Гц)
0202h - G (392,0 Гц)
0220h - G# (415,3 Гц)
0241h – A (440,0 Г ц)
0263h - A# (466,2 Гц)
0287h - B (493,9 Гц)
02AEh - C (523,3 Гц)

Регистры **0C0h** - OC8h: обратная связь/алгоритм
бит 0: 0 - первый оператор модулирует второй, 1 – оба оператора непосредственно производят звук
биты 3–1: сила обратной связи. Если это поле ненулевое, первый оператор будет посылать долю выходного сигнала обратно в себя

Регистр **0BDh**: глубины модуляций/ритм
бит 0: включен HiHat
бит 1: включен Cymbal
бит 2: включен Tam-tam
бит 3: включен Snare drum
бит 4: включен Bass drum
бит 5: 1 - включен ритм (6 голосов на мелодию), 0 - ритм выключен (9 голосов на мелодию)
бит 6: глубина вибрато
бит 7: глубина амплитудной модуляции (1-4,8 дБ; 0-1 дБ)

Регистры **0E0h** - OF5h: выбор формы волны
биты 1–0: форма волны, использующаяся, если бит 5 регистра 01h установлен в 1.
00: синусоида
01: синусоида без отрицательной половины
02: абсолютное значение синусоиды (нижние полуволны отражены вверх)
11: пилообразные импульсы

Чтобы извлечь из FM простой звук, выполним такую последовательность действий:

1. Обнулим все регистры (грубый способ инициализации AdLib).

Канал 0 будет использоваться для голоса:

2. Запишем в регистр 20h число 01h - кратность модуляции 1.
3. Запишем в регистр 40h число 10h - уровень модуляции 40 дБ.
4. Запишем в регистр 60h число 0F0h - нарастание быстрое, спад - долгий.
5. Запишем в регистр 80h число 77h - поддержка и отпускание средние.
6. Запишем в регистр 0A0h число 98h - нота D#.

Канал 3 будет использоваться для несущей:

7. Запишем в регистр 023h число O1h - кратность несущей 1.
8. Запишем в регистр 043h число 00h - максимальная громкость для несущей (47 дБ).
9. Запишем в регистр 063h число OF0h - нарастание среднее, спад - долгий.
10. Запишем в регистр 083h число 77h - поддержка и отпускание - средние.
11. Запишем в. регистр OB0h число 031h - установить октаву, старшие биты ноты и включить голос.

С этого момента синтезатор звучит.

12. Запишем в регистр OB0h число 11h (или любое с нулевым битом 5) для выключения звука.

Пример программы

Программирование современных звуковых плат - весьма сложное занятие, поэтому в качестве примера рассмотрим одну часто применяемую операцию - *воспроизведение оцифрованного звука. С* этой целью потребуется программировать только DSP. Для вывода звука через звуковую плату может использоваться один из трех режимов: прямой вывод (команда 10h), когда программа должна сама с нужной частотой посылать отдельные байты из оцифрованного звука в DSP; простой DMA-режим, когда выводится блок данных, после чего вызывается прерывание; и DMA с автоинициализацией, когда данные выводятся непрерывно и после вывода каждого блока вызывается прерывание. Именно в этом порядке увеличивается качество воспроизводимого звука. Так как мы пока не умеем работать с DMA, рассмотрим первый способ.

Чтобы вывести оцифрованные данные с нужной частотой в DSP, придется перепрограммировать канал 0 системного таймера на требуемую частоту и установить собственный обработчик прерывания 08h. При этом будет нарушена работа системных часов, хотя можно не выключать совсем старый обработчик, а передавать ему управление примерно 18,2 раза в секунду, то есть, в частности, при каждом 604-м вызове нашего обработчика на частоте 11 025 Гц. Покажем, как это сделать на примере простой программы, которая именно таким способом воспроизведет файл c:\windows\media\tada.wav (или c:\windows\tada.wav, если вы измените соответствующую директиву EQU в начале программы).

```
; wavdir.asm
; Воспроизводит файл c:\windows\media\tada.wav, не используя DMA.
; Нормально работает только под DOS в реальном режиме
```

```
; (то есть не в окне DOS (Windows) и не под EMM386, QEMM или другими
; подобными программами).

FILESPEC equ "c:\windows\media\tada.wav" ; Имя файла tada.wav с
                                ; полным путем (замените на c:\windows\tada.wav
                                ; для старых версий Windows).
SBPORT  equ 220h                ; Базовый порт звуковой платы (замените, .
                                ; если у вас он отличается).
        .model  tiny
        .code
        .186                    ; Для pusha/popa.
org     100h                    ; COM-программа.
start:
        call    dsp_reset       ; Сброс и инициализация DSP.
        jc      no_blaster
        mov     bl,0D1h         ; Команда DSP D1h.
        call    dsp_write       ; Включить звук.
        call    open_file       ; Открыть и прочитать tada.wav.
        call    hook_int8       ; Перехватить прерывание таймера.
        mov     bx,5            ; Делитель таймера для частоты 22 050 Hz
                                ; (на самом деле соответствует 23 867 Hz).
        call    reprogram_pit   ; Перепрограммировать таймер.

main_loop:                      ; Основной цикл.
        cmp     byte ptr finished_flag,0
        je      main_loop       ; Выполняется, пока finished_flag равен нулю.

        mov     bx,0FFFFh       ; Делитель таймера для частоты 18,2 Hz.
        call    reprogram_pit   ; Перепрограммировать таймер.
        call    restore_int8    ; Восстановить IRQ0.
no_blaster:
        ret

buffer_addr     d.w     offset buffer   Адрес текущего играемого байта.
old_int08h      dd      ?               Старый обработчик INT 08h (IRQ0).
finished_flag   db      0               Флаг завершения.
filename        db      FILESPEC, 0     Имя файла tada.wav с полным путем.

; Обработчик INT 08h (IRQ0).
; Посылает байты из буфера в звуковую плату.
int08h_handler  proc    far
        pusha                           ; Сохранить регистры.
        cmp     byte ptr cs:finished_flag,1 ; Если флаг уже 1,
        je      exit_handler            ; ничего не делать.
        mov     di,word ptr cs:buffer_addr  ; Иначе: DI = адрес текущего байта.
        mov     bl,10h                  ; Команда DSP 10h.
        call    dsp_write               ; Непосредственный 8-битный вывод.
        mov     bl,byte ptr cs:[di]     ; BL = байт данных для вывода.
        call    dsp_write
        inc     di                      ; DI = адрес следующего байта.
        cmp     di,offset buffer+27459  ; 27 459 - длина звука в tada.wav.
```

```
        jb      not_finished                    ; Если весь буфер пройден,
        mov     byte ptr cs:finished_flag,1     ; установить флаг в 1.
not_finished:                                   ; Иначе:
        mov     word ptr cs:buffer_addr,di      ; сохранить текущий адрес.
exit_handler:
        mov     al,20h  ; Завершить обработчик аппаратного прерывания,
        out     20h,al  ; послав неспецифичный EOI (см. раздел 5.10.10).
        popa            ; Восстановить регистры.
        iret
int08h_handler  endp

; Процедура dsp_reset.
; Сброс и инициализация DSP.
dsp_reset       proc near
        mov     dx,SBPORT+6     ; Порт 226h - регистр сброса DSP.
        mov     al,1            ; Запись единицы в него начинает инициализацию.
        out     dx,al
        mov     cx,40           ; Небольшая пауза.
dsploop:
        in      al,dx
        loop    dsploop
        mov     al,0            ; Запись нуля завершает инициализацию.
        out     dx,al           ; Теперь DSP готов к работе.
; Проверить, есть ли DSP вообще.
        add     dx,8            ; Порт 22Eh - состояние буфера чтения DSP.
        mov     cx,100
check_port:
        in      al,dx           ; Прочитать состояние буфера.
        and     al,80h          ; Проверить бит 7.
        jz      port_not_ready  ; Если ноль - порт еще не готов.
        sub     dx,4            ; Иначе: порт 22Ah - чтение данных из DSP.
        in      al,dx
        add     dx,4            ; И снова порт 22Eh.
        cmp     al,0AAh         ; Если прочиталось число AAh - DSP присутствует
                                ; и действительно готов к работе.
        je      good_reset
port_not_ready:
        loop    check_port      ; Если нет - повторить проверку 100 раз
bad_reset:
        stc                     ; и сдаться.
        ret                     ; Выход с CF = 1.
good_reset:
        clc                     ; Если инициализация прошла успешно,
        ret                     ; выход с CF = 0.
dsp_reset       endp

; Процедура dsp_write.
; Посылает байт из BL в DSP.
dsp_write       proc near
        mov     dx,SBPORT+0Ch   ; Порт 22Ch - ввод данных/команд DSP.
```

```
write_loop:                         ; Подождать готовности буфера записи DSP.
        in      al,dx               ; Прочитать порт 22Ch
        and     al,80h              ; и проверить бит 7.
        jnz     write_loop          ; Если он не ноль - подождать еще.
        mov     al,bl               ; Иначе:
        out     dx,al               ; послать данные.
        ret
dsp_write       endp

; Процедура  reprogram_pit.
; Перепрограммирует канал 0 системного таймера на новую частоту.
; Вход: BX = делитель частоты.
reprogram_pit   proc    near
        cli                         ; Запретить прерывания.
        mov     al,00110110b        ; Канал 0,  запись младшего и старшего байтов,
                                    ; режим работы 3, формат счетчика - двоичный.
        out     43h,al              ; Послать это в регистр команд первого таймера.
        mov     al,bl               ; Младший байт делителя -
        out     40h,al              ; в регистр данных канала 0.
        mov     al,bh               ; И старший байт -
        out     40h,al              ; туда же.
        sti                         ; Теперь IRQ0 вызывается с частотой 1 193 180/BX Hz.
        ret
reprogram_pit   endp

; Процедура hook_int8.
; Перехватывает прерывание INT 08h (IRQ0).
hook_int8       proc    near
        mov     ax,3508h                ; AH = 35h, AL = номер прерывания.
        int     21h                     ; Получить адрес старого обработчика.
        mov     word ptr old_int08h,bx  ; Сохранить его в old_int08h.
        mov     word ptr old_int08h+2,es
        mov     ax,2508h                ; AH = 25h, AL = номер прерывания.
        mov     dx,offset int08h_handler ; DS:DX - адрес обработчика.
        int     21h                     ; Установить обработчик.
        ret
hook_int8       endp

; Процедура restore_int8.
; Восстанавливает прерывание INT 08h (IRQ0).
restore_int8    proc near
        mov     ax,2508h                ; AH = 25h, AL = номер прерывания.
        lds     dx,dword ptr old_int08h ; DS:DX - адрес обработчика.
        int     21h                     ; Установить старый обработчик.
        ret
restore_int8    endp

; Процедура open_file.
; Открывает файл filename и копирует звуковые данные из него, считая его файлом
; tada.wav, в буфер buffer.
open_file       proc near
        mov     ax,3D00h                ; AH = 3Dh, AL = 00.
```

```
        mov     dx,offset filename      ; DS:DX - ASCIZ-имя файла с путем.
        int     21h                     ; Открыть файл для чтения.
        jc      error_exit              ; Если не удалось открыть файл - выйти.
        mov     bx,ax                   ; Идентификатор файла в BX.
        mov     ax,4200h                ; AH = 42h, AL = 0.
        mov     cx,0                    ; CX:DX - новое значение указателя.
        mov     dx,38h          ; По этому адресу начинаются данные в tada.wav.
        int     21h                     ; Переместить файловый указатель.
        mov     ah,3Fh                  ; AH = 3Fh.
        mov     cx,27459        ; Это - длина звуковых данных в файле tada.wav.
        mov     dx,offset buffer        ; DS:DX - адрес буфера.
        int     21h                     ; Чтение файла.
        ret
error_exit:                             ; Если не удалось открыть файл.
        mov     ah,9                    ; AH = 09h.
        mov     dx,offset notopenmsg    ; DS:DX = сообщение об ошибке.
        int     21h                     ; Открыть файл для чтения.
        int     20h                     ; Конец программы.
notopenmsg      db      "Ошибка при открытии файла", 0Dh, 0Ah, '$'
open_file       endp

buffer:                                 ; Здесь начинается буфер длиной 27 459 байт.
        end     start
```

Если вы скомпилировали программу latency.asm из раздела 5.10.5 и попробовали запустить ее в разных условиях, то могли заметить, что под Windows 95, а также под EMM386 и в некоторых других ситуациях пауза между реальным срабатыванием прерывания таймера и запуском обработчика может оказаться весьма значительной и варьироваться с течением времени, так что качество звука, выводимого нашей программой wavdir.asm, окажется под EMM386 очень плохим, а в DOS-задаче под Windows 95 вообще получится протяжный хрип. Чтобы этого избежать, а также чтобы указывать точную скорость оцифровки звука и выводить 16-битный звук, нужно обратиться к программированию контроллера DMA (пример программы, выводящей звук при помощи ,DMA, см. в конце следующего раздела).

5.10.9. Контроллер DMA

Контроллер DMA используется для обмена данными между внешними устройствами и памятью. Он нужен в работе с жесткими дисками и дисководами, звуковыми платами и другими устройствами, работающими со значительными объемами данных. Начиная с PC AT, в компьютерах присутствуют два DMA-контроллера - 8-битный (с каналами О, 1, 2 и 3) и 16-битный (с каналами 4, 5, 6 и 7). Канал 2 используется для обмена данными с дисководами, канал 3 - для жестких дисков, канал 4 теряется при каскадировании контроллеров, а назначение остальных каналов может варьироваться.

DMA позволяет выполнить чтение или запись блока данных, начинающегося с линейного адреса, который описывается как 20-битное число для первого DMA-контроллера и как 24-битное для второго, то есть данные для 8-битного DMA должны располагаться в пределах первого мегабайта памяти, а для второго -

в пределах первых 16 Мб. Старшие четыре бита для 20-битных адресов и старшие 8 бит для 24-битных адресов хранятся в регистрах страниц DMA, адресуемых через порты 80h - 8Fh:

порт 81h: страничный адрес для канала 2 (биты 3-0 = биты 19-16 адреса)
порт 82h: страничный адрес для канала *3* (биты 3-0 = биты 19-16 адреса)
порт 83h: страничный адрес для канала 1 (биты 3-0 = биты 19-16 адреса)
порт 87h: страничный адрес для канала 0 (биты 3-0 = биты 19-16 адреса)
порт 89h: страничный адрес для канала 6 (биты 7-0 = биты 23-17 адреса)
порт 8Ah: страничный адрес для канала 7 (биты 7-0 = биты 23-17 адреса)
порт 8Bh: страничный адрес для канала 5 (биты 7-0 = биты 23-17 адреса)

Страничный адрес определяет начало 64/128-килобайтного участка памяти, с которым будет работать данный канал, поэтому при передаче данных через DMA обязательно надо следить за тем, чтобы не было выхода за границы этого участка, то есть чтобы не было попытки пересечения адреса 1000h:0, 2000h:0, 3000h:0 для первого DMA или 2000h:0, 4000h:0, 6000h:0 для второго.

Младшие 16 бит адреса записывают в следующие порты:

OOh: биты 15-0 адреса блока данных для канала О
01h: счетчик переданных байт канала О
02h - 03h: аналогично для канала 1
04h - 05h: аналогично для канала 2
06h - 07h: аналогично для канала 3
(для этих портов используются две операции чтения/записи - сначала передаются биты 7-0, затем биты 15-8)

OCOh: биты 8–1 адреса блока данных для канала 4 (бит 0 адреса всегда равен нулю)
0C1h: биты 16-9 адреса блока данных для канала 4
OC2h: младший байт счетчика переданных слов канала 4
0C3h: старший байт счетчика переданных слов канала 4
OC4h - 0C7h: аналогично для канала 5
OC8h - OCBh: аналогично для канала 5
OCCh - 0CFh: аналогично для канала 5
(эти порты рассчитаны на чтение/запись целыми словами)

Каждый из указанных двух DMA-контроллеров также имеет собственный набор управляющих регистров - регистры первого контроллера адресуются через порты 08h - OFh, а второго - через ODO - 0DFh:

порт 08h/ODOh для чтения: регистр состояния DMA
бит 7, 6, 5, 4: установлен запрос на DMA на канале 3/7, 2/6, 1/5, 0/4
бит 3, 2, 1, 0: закончился DMA на канале 3/7, 2/6, 1/5, 0/4
порт O8h/D0h для записи: регистр команд DMA (устанавливается BIOS)
бит 7: сигнал DACK использует высокий уровень
бит 6: сигнал DREQ использует высокий уровень

бит 5: 1/0: расширенный/задержанный цикл записи
бит 4: 1/0: приоритеты сменяются циклически/фиксированно
бит 3: сжатие во времени
бит 2: DMA-контроллер отключен
бит 1: разрешен захват канала 0 (для режима память-память)
бит 0: включен режим память-память (канал 0 - канал 1)

порт 09h/0D2h для записи: регистр запроса DMA
бит 2: 1/0: установка/сброс запроса на DMA
биты 1-0: номер канала (00, 01, 10, 11 = 0/4, 1/5, 2/6, 3/7)

порт OAh/OD4h для записи: регистр маски канала DMA
бит 2: 1/0: установка/сброс маскирующего бита
биты 1-0: номер канала (00, 01, 10, 11 = 0/4, 1/5, 2/6, 3/7)

порт OBh/OD6h для записи: регистр режима DMA
биты 7-6: 00 - передача по запросу
01 - одиночная передача (используется для звука)
10 - блочная передача (используется для дисков)
11 - канал занят для каскадирования
бит 5: 1/0: адреса уменьшаются/увеличиваются
бит 4: режим автоинициализации
биты 3-2:
00 - проверка
01 — запись
10 - чтение
биты 1-0: номер канала (00, 01, 10, 11 = 0/4, 1/5, 2/6, 3/7)

порт OCh/OD8h для записи: сброс переключателя младший/старший байт
Для чтения/записи 16-битных значений из/в 8-битные порты OOh - 08h. Очередной байт, переданный в эти порты, будет считаться младшим, а следующий за ним - старшим.

порт 0Dh/0DAh для записи: сброс контроллера DMA
Любая запись сюда приводит к полному сбросу DMA-контроллера, так что его надо инициализировать заново.

порт ODh/ODAh для чтения: последний переданный байт/слово.

порт OEh/ODCh для записи: любая запись снимает маскирующие биты со всех каналов

порт OFh/ODEh для записи: регистр маски всех каналов:
биты 3-0: маскирующие биты каналов 3/7, 2/6, 1/5, 0/4

Чаще всего внешнее устройство само инициализирует передачу данных, и все, что необходимо сделать программе, - это записать адрес начала буфера в порты, соответствующие используемому каналу, длину передаваемого блока данных минус один в регистр счетчика нужного канала, установить режим работы канала и снять маскирующий бит.

В качестве примера вернемся к программированию звуковых плат и изменим программу wavdir.asm так, чтобы она использовала DMA.

```
; wavdma.asm
; Пример программы, проигрывающей файл C:\WINDOWS\MEDIA\TADA.WAV
; на звуковой карте при помощи DMA.
FILESPEC equ "c:\windows\media\tada.wav"   ; Заменить на c:\windows\tada.wav.
                                           ; для старых версий Windows.
SBPORT  equ 220h  . .
; SBDMA equ 1                              ; Процедура program_dma рассчитана
                                           ; лишь на канал 1.
SBIRQ   equ 5                              ; Только IRQ0 - IRQ7.
        .model  tiny
        .code
        .186
        org     100h                       ; COM-программа.
start:
        call    dsp_reset                  ; Инициализация OSP.
        jc      no_blaster
        mov     bl,0D1h                    ; Команда 0D1h.
        call    dsp_write                  ; Включить звук.
        call    open_file                  ; Прочитать файл в буфер.
        call    hook_sbirq                 ; Перехватить прерывание.
        mov     bl,40h                     ; Команда 40h.
        call    dsp_write                  ; Установка скорости передачи.
        mov     bl,0B2h                    ; Константа для 11025Hz/Stereo.
        call    dsp_write
        call    program_dma                ; Начать DMA-передачу данных.

main_loop:                                 ; Основной цикл.
        cmp     byte ptr finished_flag,0
        je      main_loop                  ; Выход, когда байт finished_flag = 1.

        call    restore_sbirq              ; Восстановить прерывание.
no_blaster:
        ret

old_sbirq       dd      ?                  ; Адрес старого обработчика.
finished_flag   db      0                  ; Флаг окончания работы.
filename        db      FILESPEC, 0        ; Имя файла.

; Обработчик прерывания звуковой карты.
; Устанавливает флаг finished_flag в 1.
sbirq_handler   proc    far
        push    ax
        mov     byte ptr cs:finished_flag,1        ; Установить флаг.
        mov     al,20h                             ; Послать команду EOI
        out     20h,al                             ; в контроллер прерываний.
        pop     ax
        iret
sbirq_handler   endp

; Процедура dsp_reset.
; Сброс и инициализация DSP.
```

```
dsp_reset       proc near
        mov     dx,SBPORT+6     ; Порт 226h - регистр сброса DSP.
        mov     al,1            ; Запись в него единицы запускает инициализацию.
        out     dx,al
        mov     cx,40           ; Небольшая пауза.
dsploop:
        in      al,dx
        loop    dsploop
        mov     al,0            ; Запись нуля завершает инициализацию.
        out     dx,al           ; Теперь DSP готов к работе.

        add     dx,8            ; Порт 22Eh - бит 7 при чтении указывает на занятость
        mov     cx,100          ; буфера записи DSP.
check_port:
        in      al,dx           ; Прочитать состояние буфера записи.
        and     al,80h          ; Если бит 7 ноль,
        jz      port_not_ready  ; порт еще не готов.
        sub     dx,4            ; Иначе: порт 22Ah - чтение данных из DSP.
        in      al,dx
        add     dx,4            ; Порт снова 22Eh.
        cmp     al,0AAh         ; Проверить, что DSP возвращает 0AAп при чтении -
                                ; это сигнал о готовности к работе.
        je      good_reset
port_not_ready:
        loop    check_port      ; Повторить проверку на 0AAп 100 раз.
bad_reset:
        stc                     ; Если Sound Blaster не откликается,
        ret                     ; вернуться с CF = 1.
good_reset:
        clc                     ; Если инициализация прошла успешно,
        ret                     ; вернуться с CF = 0.
dsp_reset       endp

; Процедура dsp_write
; Посылает байт из BL в DSP
dsp_write       proc near
        mov     dx,SBPORT+0Ch   ; Порт 22Ch - ввод данных/команд DSP.
write_loop:                     ; Подождать до готовности буфера записи DSP,
        in      al,dx           ; прочитать порт 22Ch
        and     al,80h          ; и проверить бит 7.
        jnz     write_loop      ; Если он не ноль - подождать еще.
        mov     al,bl           ; Иначе:
        out     dx,al           ; послать данные.
        ret
dsp_write       endp

; Процедура hook_sbirq.
; Перехватывает прерывание звуковой карты и разрешает его.
hook_sbirq      proc    near
        mov     ax,3508h+SBIRQ  ; AH = 35h, AL = номер прерывания.
        int     21h             ; Получить адрес старого обработчика
```

```
        mov     word ptr old_sbirq,bx     ; и сохранить его.
        mov     word ptr old_sbirq+2,es
        mov     ax,2508h+SBIRQ            ; AH = 25h, AL = номер прерывания.
        mov     dx,offset sbirq_handler   ; Установить новый обработчик.
        int     21h
        mov     cl,1
        shl     cl,SBIRQ
        not     cl                        ; Построить битовую маску.
        in      al,21h                    ; Прочитать OCW1.
        and     al,cl                     ; Разрешить прерывание.
        out     21h,al                    ; Записать OCW1.
        ret
hook_sbirq      endp

; Процедура  restore_sbirq.
; Восстанавливает обработчик и запрещает прерывание.
restore_sbirq   proc near
        mov     ax,3508h+SBIRQ          ; AH = 25h, AL = номер прерывания.
        lds     dx,dword ptr old_sbirq
        int     21h                      ; Восстановить обработчик.
        mov     cl,1
        shl     cl,SBIRQ                 ; Построить битовую маску.
        in      al,21h                   ; Прочитать OCW1.
        or      al,cl                    ; Запретить прерывание
        out     21h,al                   ; Записать OCW1.
        ret
restore_sbirq   endp

; Процедура open_file.
; Открывает файл filename и копирует звуковые данные из него, считая, что
; это tada.wav, в буфер buffer.
open_file       proc near
        mov     ax,3D00h                 ; AH = 3Dh, AL = 00.
        mov     dx,offset filename       ; DS:DX - ASCIZ-строка с именем файла.
        int     21h                      ; Открыть файл для чтения.
        jc      error_exit               ; Если не удалось открыть файл - выйти.
        mov     bx,ax                    ; Идентификатор файла в BX.
        mov     ax,4200h                 ; AH = 42h, AL = 0.
        mov     cx,0                     ; CX:DX - новое значение указателя.
        mov     dx,38h                   ; По этому адресу начинаются данные
                                         ; в tada.wav.
        int     21h                      ; Переместить файловый указатель.
        mov     ah,3Fh                   ; AH = 3Fh.
        mov     cx,27459                 ; Это - длина данных в файле tada.wav.
        push    ds
        mov     dx,ds
        and     dx,0F000h        ; Выравнять буфер на границу 4-килобэйтной страницы
        add     dx,1000h         ; для DMA.
        mov     ds,dx
        mov     dx,0             ; DS:DX - адрес буфера.
        int     21h              ; Чтение файла.
```

```
        pop     ds
        ret
error_exit:                             ; Если не удалось открыть файл.
        mov     ah,9                    ; AH = 09h.
        mov     dx,offset notopenmsg      DS:DX = адрес сообщения об ошибке.
        int     21h                     ; Вывод строки на экран.
        int     20h                     ; Конец программы.
; сообщение об ошибке
notopenmsg      db      "Ошибка при открытии файла",0Dh,0Ah,'$'

open_file       endp

; Процедура program_dma.
; Настраивает канал 1 DMA.
program_dma     proc near
        mov     al,5            ; Замаскировать канал 1:
        out     0Ah,al
        xor     al,al           ; Обнулить счетчик.
        out     0Ch,al
        mov     al,49h          ; Установить режим передачи
                                ; (используйте 59h для автоинициализации).
        out     0Bh,al

        push    cs
        pop     dx
        and     dh,0F0h
        add     dh,10h          ; Вычислить адрес буфера.

        xor     ax,ax
        out     02h,al          ; Записать младшие 8 бит.
        out     02h,al          ; Записать следующие 8 бит.
        mov     al,dh
        shr     al,4
        out     83h,al          ; Записать старшие 4 бита.

        mov     ax,27459        ; Длина данных в tada.wav.
        dec     ax              ; DMA требует длину минус один.
        out     03h,al          ; Записать младшие 8 бит длины.
        mov     al,ah
        out     03h,al          ; Записать старшие 8 бит длины.
        mov     al,1
        out     0Ah,al          ; Снять маску с канала 1.

        mov     bl,14h          ; Команда 14h.
        call    dsp_write       ; 8-битное простое DMA-воспроизведение.
        mov     bx,27459        ; Размер данных в tada.wav
        dec     bx              ; минус 1.
        call    dsp_write       ; Записать в DSP младшие 8 бит длины
        mov     bl,bh
        call    dsp_write       ; и старшие.
        ret
program_dma     endp
        end     start
```

В этом примере задействован обычный DMA-режим работы, в котором звуковая плата проигрывает участок данных, вызывает прерывание и, пока обработчик прерывания подготавливает новый буфер данных, программирует DMA и звуковую плату для продолжения воспроизведения, проходит некоторое время, что может звучать как щелчок. Этого можно избежать, если воспользоваться режимом автоинициализации, позволяющим обойтись без остановок во время воспроизведения.

При использовании режима DMA с автоинициализацией нужно сделать следующее: загрузить начало воспроизводимого звука в буфер длиной, например, 8 Кб и запрограммировать DMA на его передачу с автоинициализацией. Затем сообщить DSP, что проигрывается звук с автоинициализацией и размер буфера равен 4 Кб. Далее, когда придет прерывание от звуковой платы, она не остановится и продолжит воспроизведение из вторых 4 Кб буфера, поскольку находится в режиме автоинициализации. Теперь запишем в первые 4 Кб следующий блок данных. Когда кончится 8-килобайтный буфер, DMA начнет посылать его сначала, потому что мы его тоже запрограммировали для автоинициализации (бит 4 порта 0Bh/0D6h), DSP вызовет прерывание и тоже не остановится, продолжая воспроизводить данные, которые посылает ему DMA-контроллер, а мы тем временем запишем во вторые 4 Кб буфера следующий участок проигрываемого файла и т. д.

5.10.10. Контроллер прерываний

Контроллер прерываний - устройство, которое получает запросы на аппаратные прерывания от всех внешних устройств. Он определяет, какие запросы следует обслужить, какие должны ждать своей очереди, а какие не будут обслуживаться вообще. Существует два контроллера прерываний, так же как и DMA. Первый контроллер, обслуживающий запросы на прерывания от IRQ0 до IRQ7, управляется через порты 20h и 21h, а второй (IRQ8 - IRQ15) - через порты 0A0h и 0A1h.

Команды контроллеру делят на команды управления (OCW) и инициализации (ICW):

порт 20h/OAOh для записи: OCW2, OCW3, ICW1
порт 20h/OAOh для чтения: см. команду OCW3
порт 21h/0A1h для чтения и записи: OCW1 - маскирование прерываний
порт 21h/OA1h для записи: ICW2, ICW3, ICW4 сразу после ICW1

Команды управления

OCW1:

биты 7-0: прерывание 7-0/15-8 запрещено

При помощи этой команды можно временно запретить или разрешить то или иное аппаратное прерывание. Например, команды

```
in      al,21h
or      al,00000010b
out     21h,al
```

приводят к отключению IRQ1, то есть клавиатуры.

Мы пользовались OCW1 в программе term2.asm, чтобы разрешить IRQ3 - прерывание от последовательного порта COM2.

OCW2: команды конца прерывания и сдвига приоритетов
биты 7-5: команда
000b: запрещение сдвига приоритетов в режиме без EOI
001b: неспецифичный EOI (конец прерывания в режиме с приоритетами)
010b: нет операции
011b: специфичный EOI (конец прерывания в режиме без приоритетов)
100b: разрешение сдвига приоритетов в режиме без EOI
101b: сдвиг приоритетов с неспецифичным EOI
110b: сдвиг приоритетов
111b: сдвиг приоритетов со специфичным EOI
биты 4-3: 00b (указывают, что это OCW2)
биты 2-0: номер IRQ для команд 011b, 110b и 111b

Как упоминалось в разделе 5.8.2, если несколько прерываний происходят одновременно, обслуживается в первую очередь то, у которого высший приоритет. При инициализации контроллера высший приоритет имеет IRQ0 (прерывание от системного таймера), а низший - IRQ7. Все прерывания второго контроллера (IRQ8 - IRQ15) оказываются в этой последовательности между IRQ1 и IRQ3, так как именно IRQ2 используется для каскадирования этих двух контроллеров. Команды сдвига приоритетов позволяют изменить ситуацию, присвоив завершающемуся (команды 101 или 111) или обрабатывающемуся (110) прерыванию низший приоритет, причем следующее прерывание получит наивысший, и далее по кругу.

Более того, в тот момент, когда выполняется обработчик аппаратного прерывания, других прерываний с низшими приоритетами нет, даже если обработчик выполнил команду sti. Чтобы разрешить другие прерывания, каждый обработчик обязательно должен послать команду EOI - конец прерывания - в соответствующий контроллер. Именно поэтому обработчики аппаратных прерываний в программах term2.asm и wavdma.asm заканчивались командами

```
mov     al,20h  ; команда "неспецифичный конец прерывания"
out     20h,al  ; посылается в первый контроллер прерываний
```

Если бы контроллер был Инициализирован в режиме без приоритетов, вместо неспецифичного EOI пришлось бы посылать специфичный, содержащий в младших трех битах номер прерывания, но BIOS инициализирует контроллер именно в режиме с приоритетами. Кроме того, контроллер мог быть инициализирован в режиме без EOI, но тогда в ходе работы обработчика прерывания могли бы происходить все остальные прерывания, включая обрабатываемое. О способах инициализации контроллера говорится далее, а здесь рассмотрим последнюю команду управления.

OCW3: чтение состояния контроллера и режим специального маскирования
бит 7: 0

биты 6-5: режим специального маскирования
00 - не изменять
10 - выключить
11 - включить
биты 4-3: 01 - указывает, что это OCW3
бит 2: режим опроса
биты 1-0: чтение состояния контроллера
00 - не читать
10 - читать регистр запросов на прерывания
11 - читать регистр обслуживаемых прерываний

В режиме специального маскирования при выполнении обработчика прерывания разрешены все прерывания, кроме осуществляющегося в настоящий момент и маскируемых командой OCW1, что имеет смысл сделать, если обработчику прерывания с достаточно высоким приоритетом потребуется много времени.

Чаще всего OCW3 используют для чтения состояния контроллера - младшие два бита выбирают, какой из регистров контроллера будет возвращаться при последующем чтении из порта 21h/0A1h. Оба возвращаемых регистра имеют структуру, аналогичную OCW1, - каждый бит отвечает соответствующему IRQ.

Из регистра запросов на прерывания можно узнать, какие прерывания произошли, но пока не были обработаны, а из регистра обслуживаемых прерываний - какие прерывания обрабатываются в данный момент. Итак, еще одна мера безопасности, которую применяют резидентные программы, - нельзя работать с дисководом (IRQ6), если в данный момент обслуживается прерывание от последовательного порта (IRQ3), и нельзя работать с диском (IRQ14/15), если обслуживается прерывание от системного таймера (IRQ0).

Команды инициализации

Чтобы инициализировать контроллер, BIOS посылает последовательность команд: ICW1 в порт 20h/OAOh (она отличается от OCW своим битом 4) и ICW2, ICW3, ICW4 в порт 21h/0A1h сразу после этого.

ICW1:
биты 7-4: 0001b
бит 3: 1/0: срабатывание по уровню/фронту сигнала IRQ (принято 0)
бит 2: 1/0: размер вектора прерывания 4 байта/8 байт (1 для 80x86)
бит 1: каскадирования нет, ICW3 не будет послано
бит 0: ICW4 будет послано
ICW2: номер обработчика прерывания для IRQO/IRQ8 (кратный восьми)
(08h - для первого контроллера, 70h - для второго. Некоторые операционные системы изменяют первый обработчик на 50h)
ICW3 для ведущего контроллера:
биты 7-0: к выходу 7-0 присоединен подчиненный контроллер (0100b в PC)
ICW3 для подчиненного контроллера:
биты 3-0: номер выхода ведущего контроллера, к которому подсоединен ведомый

ICW4:

биты 7-5: О
бит 4: контроллер в режиме фиксированных приоритетов
биты 3-2: режим:
00, 01 - небуферированный
10 - буферированный/подчиненный
11 - буферированный/ведущий
бит 1: режим с автоматическим EOI
(то есть обработчикам не надо посылать EOI в контроллер)
бит 0: 0 - режим совместимости с 8085; 1 - обычный

Повторив процедуру инициализации, программа может, например, изменить соответствие между обработчиками прерываний и реальными аппаратными прерываниями. Переместив базовый адрес первого контроллера на неиспользуемую область (например, 50h) и установив собственные обработчики на каждое из прерываний INT 50h - 58h, вызывающие INT 08h - 0Fh, вы будете абсолютно уверены в том, что никакая программа не определит обработчик аппаратного прерывания, который получил бы управление раньше вашего.

```
; picinit.asm
; Выполняет полную инициализацию обоих контроллеров прерываний
; с отображением прерываний IRQ0 - IRQ7 на векторы INT 50h - 57h.
; Программа остается резидентной и издает короткий звук после каждого IRQ1.
; Восстановление старых обработчиков прерываний и переинициализация
; контроллера в прежнее состояние опущены.

        .model  tiny
        .code
org     100h                    ; COM-программа.

PIC1_BASE       equ     50h     ; На этот адрес процедура pic_init перенесет
                                ; IRQ0 - IRQ7.
PIC2_BASE       equ     70h     ; На этот адрес процедура pic_init перенесет
                                ; IRQ8 - IRQ15.

start:
        jmp     end_of_resident ; Переход на начало инсталляционной части.

irq0_handler:                   ; Обработчик IRQ0
                                ; (прерывания от системного таймера).
        push    ax
        in      al,61h
        and     al,11111100b    ; Выключение динамика.
        out     61h,al
        pop     ax
        int     08h             ; Старый обработчик IRQ0.
        iret                    ; Он послал EOI, так что завершить простым iret
irq1_handler:                   ; обработчик IRQ1 (прерывание от клавиатуры).
        push    ax
        in      al,61h
```

```
        or      al,00000011b    ; Включение динамика.
        out     61h,al
        pop     ax
        int     09h             ; Старый обработчик IRQ1.
        iret
irq2_handler:                   ; И так далее.
        int     0Ah
        iret
irq3_handler:
        int     0Bh
        iret
irq4_handler:
        int     0Ch
        iret
irq5_handler:
        int     0Dh
        iret
irq6_handler:
        int     0Eh
        iret
irq7_handler:
        int     0Fh
        iret

end_of_resident:                ; Конец резидентной части.
        call    hook_pic1_ints  ; Установка наших обработчиков
                                ; INT 50h - 57h.
        call    init_pic        ; Переинициализация контроллера прерываний.
        mov     dx,offset end_of_resident
        int     27h             ; Оставить наши новые обработчики резидентными.

; Процедура init_pic.
; Выполняет инициализацию обоих контроллеров прерываний,
; отображая IRQ0 - IRQ7 на PIC1_BASE - PIC1_BASE+7,
; а IR08 - IRQ15 на PIC2_BASE - PIC2_BASE+7.
; Для возврата в стандартное состояние вызвать
; PIC1_BASE = 08h,
; PIC2_BASE = 70h.
init_pic        proc    near
        cli
        mov     al,00010101b    ; ICW1.
        out     20h,al
        out     0A0h,al
        mov     al,PIC1_BASE    ; ICW2 для первого контроллера.
        out     21h,al
        mov     al,PIC2_BASE    ; ICW2 для второго контроллера.
        out     0A1h,al
        mov     al,04h          ; ICW3 для первого контроллера.
        out     21h,al
        mov     al,02h          ; ICW3 для второго контроллера.
```

```
        out     0A1h,al
        mov     al,00001101b    ; ICW4 для первого контроллера.
        out     21h,al
        mov     al,00001001b    ; ICW4 для второго контроллера.
        out     0A1h,al
        sti
        ret
init_pic        endp

; Перехват прерываний от PIC1_BASE до PIC1_BASE+7.
hook_pic1_ints  proc near
        mov     ax,2500h+PIC1_BASE
        mov     dx,offset irq0_handler
        int     21h
        mov     ax,2501h+PIC1_BASE
        mov     dx,offset irq1_handler
        int     21h
        mov     ax,2502h+PIC1_BASE
        mov     dx,offset irq2_handler
        int     21h
        mov     ax,2503h+PIC1_BASE
        mov     dx,offset irq3_handler
        int     21h
        mov     ax,2504h+PIC1_BASE
        mov     dx,offset irq4_handler
        int     21h
        mov     ax,2505h+PIC1_BASE
        mov     dx,offset irq5_handler
        int     21h
        mov     ax,2506h+PIC1_BASE
        mov     dx,offset irq6_handler
        int     21h
        mov     ax,2507h+PIC1_BASE
        mov     dx,offset irq7_handler
        int     21h
        ret
hook_pic1_ints  endp

        end     start
```

5.10.11. Джойстик

И напоследок - о программировании джойстика. Он подключается к еще одному, помимо последовательного и параллельного, внешнему порту компьютера - к игровому. Для игрового порта зарезервировано пространство портов ввода-вывода от 200h до 20Fh, но при общении с джойстиком используется всего один порт - 201h, чтение из которого возвращает состояние джойстика:

порт 201h для чтения:

биты 7, 6: состояние кнопок 2, 1 джойстика B
биты 5, 4: состояние кнопок 2, 1 джойстика A

биты 3, 2: y- и x-координаты джойстика B
биты 1, 0: y- и x-координаты джойстика A

С состоянием кнопок все просто - достаточно прочитать байт из 201h и определить значение нужных битов. Но для определения координаты джойстика придется выполнить весьма неудобную и медленную операцию: надо записать в порт 201h любое число и засечь время, постоянно считывая состояние джойстика. Сразу после записи в порт биты координат будут равны нулю, и время, за которое они обращаются в 1, пропорционально соответствующей координате (X-координаты растут слева направо, а Y-координаты - сверху вниз).

Если джойстик отсутствует, биты координат или будут единицами с самого начала, или останутся нулями неопределенно долгое время. Кроме того, после записи в порт 201h нельзя производить это же действие еще раз, пока хотя бы один из четырех координатных битов не обратится в 1.

В BIOS для работы с джойстиком есть функция 84h прерывания 15h, но общение напрямую с портами оказывается гораздо быстрее и ненамного сложнее. Например, чтобы определить координаты джойстика, BIOS выполняет целых четыре цикла измерения координат, по одному на каждую.

Чтобы получить значение координаты в разумных единицах, мы определим, на сколько изменилось показание счетчика канала 0 системного таймера, и разделим это число на 16 - результатом окажется то самое число, которое возвращает BIOS. Для стандартного джойстика (150кОм) оно должно быть в пределах 0–416, хотя обычно максимальное значение оказывается около 150. Так как аналоговые джойстики - не точные устройства, координаты для одной и той же позиции могут изменяться на 1-2, и это надо учитывать особенно при определении состояния покоя.

Покажем, как все это можно реализовать на примере чтения координат джойстика A:

```
; Процедура read_joystick.
; Определяет текущие координаты джойстика A.
; Выход: BP - Y-координата, BX - X-координата (-1, если джойстик
; не отвечает), регистры не сохраняются.

read_joystick    proc    near
        pushf                   ; Сохранить флаги
        cli                     ; и отключить прерывания, так как
                                ; вычисляется время выполнения кода и не
                                ; нужно измерять еще и время выполнения
                                ; обработчиков прерываний.
        mov     bx, -1          ; Значение X, если джойстик не ответит.
        mov     bp, bx          ; Значение Y, если джойстик не ответит.
        mov     dx, 201h        ; Порт.

        mov     al, 0
        out     43h, al         ; Зафиксировать счетчик канала 0 таймера.
        in      al, 40h
```

```
        mov     ah,al
        in      al,40h
        xchg    ah,al           ; AX - значение счетчика.
        mov     di,ax           ; Записать его в DI.

        out     dx,al           ; Запустить измерение координат джойстика.
        in      al,dx           ; Прочитать начальное состояние координат.
        and     al,011b
        mov     cl,al           ; Записать его в CL.
read_joystick_loop:
        mov     al,0
        out     43h,al          ; Зафиксировать счетчик канала 0 таймера.
        in      al,40h
        mov     ah,al
        in      al,40h
        xchg    ah,al           ; AX - значение счетчика.
        mov     si,.di          ; SI - начальное значение счетчика.
        sub     si,ax           ; SI - разница во времени,
        cmp     si,1FF0h        ; Если наступил тайм-аут
                                ; (значение взято из процедуры BIOS),
        ja      exit_read]      ; выйти из процедуры.

        in      al,dx           ; Иначе: прочитать состояние джойстика
        and     al,0011b
        cmp     al,cl           ; сравнить его с предыдущим.
        je      read_joystick_loop
        xchg    al,cl           ; Поместить новое значение в CL
        xor     al,cl           ; и определить изменившийся бит.
        test    al,01b          ; Если это X-координата,
        jz      x_same
        mov     bx,si           ; записать X-координату в BX.
x_same:
        test    al,10b          ; Если это Y-координата,
        jz      read_joystick_loop
        mov     bp,si           ; записать Y-координату в BP.

exit_readj:
        test    bx,bx           ; Проверить, равен ли BX -1.
        js      bx_bad
        shr     bx,4            ; Если нет - разделить на 16.
bx_bad: test    bp,bp           ; Проверить, равен ли BP -1.
        js      bp_bad
        shr     bp,4            ; Если нет - разделить на 16.
bp_bad:
        popf
        ret
read_joystick   endp
```

Если вы когда-нибудь играли с помощью джойстика, то наверняка вам известна процедура калибровки, когда игра предлагает провести джойстик по двум или

четырем углам. Это необходимо выполнять, чтобы определить, какие координаты возвращает конкретный джойстик для крайних положений, так как даже у одного и того же джойстика данные величины могут со временем изменяться.

5.11. Драйверы устройств в DOS

Итак, в предыдущих разделах говорилось о том, как происходит работа с некоторыми устройствами на самом низком уровне - уровне портов ввода-вывода. Однако прикладные программы обычно никогда не используют это уровень, а обращаются ко всем устройствам через средства операционной системы. DOS, в свою очередь, обращается к средствам BIOS, которые осуществляют взаимодействие на уровне портов со всеми стандартными устройствами. Фактически процедуры BIOS и выполняют функции драйверов устройств - программ, осуществляющих интерфейс между операционной системой и аппаратной частью компьютера. BIOS обычно лучше всего известно, как управлять устройствами, которые поставляются вместе с компьютером, но, если требуется подключить новое устройство, о котором BIOS ничего не знает, появляется необходимость в специально написанном загружаемом драйвере.

Драйверы устройств в DOS - исполняемые файлы со специальной структурой, которые загружаются на этапе запуска (при выполнении команд DEVICE или DEVICEHIGH файла config.sys) и становятся фактически частью системы. Драйвер всегда начинается с 18-байтного заголовка:

+00: 4 байта - дальний адрес следующего загружаемого драйвера DOS - так как в момент загрузки драйвер будет последним в цепочке, адрес должен быть равен OFFFFh:OFFFFh
+04: 2 байта - атрибуты драйвера
+06: 2 байта - адрес процедуры стратегии
+08: 2 байта - адрес процедуры прерывания
+0Ah: 8 байт - имя драйвера для символьных устройств (дополненное пробелами) для блочных устройств - байт по смещению OAh включает число устройств, поддерживаемых этим драйвером, а остальные байты могут содержать имя драйвера

Здесь следует заметить, что DOS поддерживает два типа драйверов - символьного и блочного устройств. Первый тип используется для любых устройств - клавиатуры, принтера, сети, а второй - только для устройств, на которых могут существовать файловые системы, то есть для дисководов, RAM-дисков, нестандартных жестких дисков, для доступа к разделам диска, занятым другими операционными системами, и т. д. Для работы с символьным устройством программа должна открыть его при помощи функции DOS «открыть файл или устройство», а для работы с блочным устройством - обратиться к соответствующему логическому диску.

Итак, код драйвера устройства представляет собой обычный код программы, как и в случае с COM-файлом, но в начале не надо размещать директиву org 100h для пропуска PSP. Можно также объединить драйвер и исполняемую программу,

разместив в EXE-файле код драйвера с нулевым смещением от начала сегмента, а точку входа самой программы ниже.

При обращении к драйверу DOS сначала вызывает процедуру стратегии (адрес по смещению 06 в заголовке), передавая ей адрес буфера запроса, содержащий все параметры, передаваемые драйверу, а затем процедуру прерывания (адрес по смещению 08) без каких-либо параметров. Процедура стратегии должна сохранить адрес буфера запроса, а процедура прерывания - собственно выполнить все необходимые действия. Структура буфера запроса меняется в зависимости от типа команды, передаваемой драйверу, но структура его заголовка остается постоянной:

+00h: байт - длина буфера запроса (включая заголовок)
+01h: байт - номер устройства (для блочных устройств)
+02h: байт - код команды (00h - 19h)
+03h: 2 байта - слово состояния драйвера - должно быть заполнено драйвером
  бит 15: произошла ошибка
  биты 10-14: 00000b
  бит 9: устройство занято
  бит 8: команда обслужена
  биты 7-0: код ошибки
    OOh: устройство защищено от записи
    01h: неизвестное*устройство
    02h: устройство не готово
    03h: неизвестная команда
    04h: ошибка CRC
    05h: ошибка в буфере запроса
    06h: ошибка поиска
    07h: неизвестный носитель
    08h: сектор не найден
    09h: нет бумаги
    0Ah: общая ошибка записи
    OBh: общая ошибка чтения
    0Ch: общая ошибка
    OFh: неожиданная смена диска
+05h: 8 байт — зарезервировано
+0Dh: отсюда начинается область данных, отличающаяся для разных команд

Даже если драйвер не поддерживает запрошенную от него функцию, он обязательно должен установить бит 8 слова состояния в 1.

Рассмотрим символьные и блочные драйверы на конкретных примерах.

5.11.1. Символьные устройства

Драйвер символьного устройства должен содержать в поле атрибутов драйвера (смещение 04 в заголовке) единицу в самом старшем бите. Тогда остальные биты трактуются следующим образом:

бит 15: 1
бит 14: драйвер поддерживает функции чтения/записи IOCTL
бит 13: драйвер поддерживает функцию вывода до занятости
бит 12: 0
бит 11: драйвер поддерживает функции открыть/закрыть устройство
биты 10-8: 000
бит 7: драйвер поддерживает функцию запроса поддержки IOCTL
бит 6: драйвер поддерживает обобщенный IOCTL
бит 5: 0
бит 4: драйвер поддерживает быстрый вывод (через INT 29h)
бит 3: драйвер устройства «часы»
бит 2: драйвер устройства NUL
бит 1: драйвер устройства STDOUT
бит 0: драйвер устройства STDIN

IOCTL - это большой набор функций (свыше пятидесяти), доступных как различные подфункции INT 21h AH = 44h и предназначенных для прямого взаимодействия с драйверами. Но о IOCTL - чуть позже, а сейчас познакомимся с тем, как устроен, возможно, самый простой из реально полезных драйверов.

В качестве первого примера рассмотрим драйвер, который вообще не обслуживает никакое устройство, реальное или виртуальное, а просто увеличивает размер буфера клавиатуры BIOS до 256 (или больше) символов. Этого можно было бы добиться обычной резидентной программой, но BIOS хранит в своей области данных только ближние адреса для этого буфера, то есть смещения относительно сегментного адреса 0040h. Так как драйверы загружаются в память первыми, еще до командного интерпретатора, они обычно попадают в область линейных адресов 00400h - 10400h, в то время как с резидентными программами этого может не получиться.

Наш драйвер будет обрабатывать только одну команду, команду инициализации драйвера OOh. Для нее буфер запроса выглядит следующим образом:

+00h: байт - 19h (длина буфера запроса)
+01h: байт — не используется
+02h: байт - 00 (код команды)
+03h: байт — слово состояния драйвера (заполняется драйвером)
+05h: 8 байт - не используется
+0Dh: байт - число обслуживаемых устройств (заполняется блочным драйвером)
+0Eh: 4 байта - на входе - конец доступной для драйвера памяти; на выходе - адрес первого байта из той части драйвера, которая не будет резидентной (чтобы выйти без инсталляции - здесь надо записать адрес первого байта)
+12h: 4 байта - на входе - адрес строки в CONFIG.SYS, загрузившей драйвер; на выходе - адрес массива BPB (для блочных драйверов)
+16h: байт — номер первого диска
+17h: 2 байта – сообщение об ошибке (0000h, если ошибки не было) - заполняется драйвером

Процедура инициализации может использовать функции DOS 01h - 0Ch, 25h, 30h и 35h.

```
; kbdext.asm
; Драйвер символьного устройства, увеличивающий буфер клавиатуры до BUF_SIZE
; (256 по умолчанию) символов.
;
BUF_SIZE equ     256                    ; Новый размер буфера.

        .model  tiny
        .186                            ; Для сдвигов и push 0040h.
        .code
        org     0                       ; Драйвер начинается с CS:0000.
start:
; Заголовок драйвера.
                dd      -1              ; Адрес следующего драйвера - 0FFFFh:0FFFFh
                                        ; для последнего.
                dw      8000h                   ; Атрибуты: символьное устройство,
                                        ; ничего не поддерживает.
                dw      offset strategy ; Адрес процедуры стратегии.
                dw      offset interrupt ; Адрес процедуры прерывания.
                db      "$$KBDEXT"      ; Имя устройства (не должно совпадать
                                        ; с каким-нибудь именем файла).

request         dd      ?               ; Здесь процедура стратегии сохраняет
                                        ; адрес буфера запроса.
buffer          db      BUF_SIZE*2 dup (?); А это - наш новый буфер
                                        ; клавиатуры размером BUF_SIZE символов
                                        ; (два байта на символ).
; Процедура стратегии.
; На входе ES:BX = адрес буфера запроса.
strategy proc   far
        mov     cs:word ptr request,bx  ; Сохранить этот адрес для
        mov     cs:word ptr request+2,es ; процедурыпрерывания.
        ret
strategy        endp

; Процедура прерывания.
interrupt       proc    far
        push    ds                      ; Сохранить регистры.
        push    bx
        push    ax
        lds     bx,dword ptr cs:request ; DS:BX - адрес запроса.
        mov     ah,byte ptr [bx+2]      ; Прочитать номер команды.
        or      ah,ah                   ; Если команда 00h (инициализация),
        jnz     exit
        call    init                    ; обслужить ее.
                                        ; Иначе:
exit:   mov     ax,100h                 ; установить бит 8 (команда обслужена)
        mov     word ptr [bx+3],ax      ; в слове состояния драйвера
        pop     ax                      ; и восстановить регистры.
```

```
        pop     bx
        pop     ds
        ret
interrupt       endp

; Процедура инициализации.
;  Вызывается только раз при загрузке драйвера.
init    proc    near
        push    cx
        push    dx

        mov     ax,offset buffer
        mov     cx,cs                   ; CX:AX - адрес нашего буфера клавиатуры.
        cmp     cx,1000h                ; Если CX слишком велик,
        jnc     too_big                 ; не надо загружаться.
        shl     cx,4                    ; Иначе: умножить сегментный адрес на 16,
        add     cx,ax                   ; добавить смещение - получился
                                        ; линейный адрес.
        sub     cx,400h                 ; Вычесть линейный адрес начала данных BIOS.

        push    0€40h
        pop     ds
        mov     bx,1Ah                  ; DS:BX = 0040h:001Ah - адрес головы.
        mov     word ptr [bx],cx        ; Записать новый адрес головы буфера.
        mov     word ptr [bx+2],cx      ; Он же новый адрес хвоста.
        mov     bl,80h                  ; DS:BX = 0040h:0080h -
                                        ; адрес начала буфера.
        mov     word ptr [bx],cx        ; Записать новый адрес начала.
        add     cx,BUF_SIZE*2           ; Добавить размер
        mov     word ptr [bx+2],cx      ; и записать новый адрес конца.

        mov     ah,9                    ; Функция DOS 09h.
        mov     dx,offset succ_msg      ; DS:DX - адрес строки
        push    cs                      ; с сообщением об успешной установке.
        pop     ds
        int     21h                     ; Вывод строки на экран.

        lds     bx,dword ptr cs:request ; DS:BX - адрес запроса.
        mov     ax,offset init
        mov     word ptr [bx+0Eh],ax    ; CS:AX - следующий байт после
        mov     word ptr [bx+10h],cs    ; конца резидентной части.
        jmp     short done              ; Конец процедуры инициализации.

; Сюда передается управление, если мы загружены слишком низко в памяти.
too_big:
        mov     ah,9                    ; Функция DOS 09h.
        mov     dx,offset fail_msg      ; DS:DX - адрес строки
        push    cs                      ; с сообщением о неуспешной
        pop     ds                      ; установке.
        int     21h                     ; Вывод строки на экран.

        lds     bx,dword ptr cs:request ; DS:BX - адрес запроса.
```

```
        mov     word ptr [bx+0Eh],0      ; Записать адрес начала драйвера
        mov     word ptr [bx+10h],cs     ; в поле "адрес первого
                                         ; освобождаемого байта".
done:   pop     dx
        pop     cx
        ret
init            endp
; Сообщение об успешной установке (на английском, потому что в этот момент
; русские шрифты еще не загружены).
succ_msg        db      "Keyboard extender loaded",0Dh,0Ah,'$'
; Сообщение о неуспешной установке.
fail_msg        db      "Too many drivers in memory - "
                db      "put kbdext.sys first "
                db      "in config.sys",0Dh,0Ah,'$'
        end     start
```

Теперь более подробно рассмотрим функции, которые должен поддерживать ROT13 — драйвер символьного устройства. ROT13 - это метод простой модификации английского текста, применяющийся в электронной почте, чтобы текст нельзя было прочитать сразу. Методика заключается в сдвиге каждой буквы латинского алфавита на 13 позиций (в любую сторону, так как всего 26 букв). Раскодирование, очевидно, выполняется такой же операцией. Когда наш драйвер загружен, команда DOS

```
copy encrypt.txt rot13
```

приведет к тому, что текст из encrypt.txt будет выведен на экран, зашифрованный или расшифрованный ROT13, в зависимости от того, был ли он зашифрован до этого.

Рассмотрим все команды, которые может поддерживать символьное устройство, и буфера запросов, которые им передаются.

00h: инициализация (уже рассмотрена)
03h: IOCTL-чтение (если установлен бит 14 атрибута)
+OEh: 4 байта - адрес буфера
+12h: 2 байта — на входе — запрашиваемое число байтов
на выходе - реально записанное в буфер число байтов
04h: чтение из устройства
структура буфера для символьных устройств совпадает с 03h
05h: чтение без удаления символа из буфера
+ODh: на выходе - прочитанный символ,
если символа нет - установить бит 9 слова состояния
06h: определить состояние буфера чтения,
если в буфере нет символов для чтения — установить бит 9 слова состояния
07h: сбросить буфер ввода
08h: запись в устройство
+OEh: 4 байта - адрес буфера
+12h: 2 байта - на входе - число байтов для записи
на выходе - число байтов, которые были записаны

09h: запись в устройство с проверкой
аналогично 08h
0Ah: определить состояние буфера записи,
если в устройство нельзя писать - установить бит 9 слова состояния
0Bh: сбросить буфер записи
0Ch: IOCTL-запись (если установлен бит 14 атрибута),
аналогично 08h
ODh: открыть устройство (если установлен бит 11 атрибута)
0Eh: закрыть устройство (если установлен бит 11 атрибута)
11h: вывод, пока не занято (если установлен бит 13 атрибута),
аналогично 08h
в отличие от функций записи здесь не считается ошибкой записать не все байты
13h: обобщенный IOCTL (если установлен бит 6 атрибута)
+ODh: байт - категория устройства (01, 03, 05 = COM, CON, LPT)
OOh - неизвестная категория
+OEh: байт – код подфункции:
45h: установить число повторных попыток
65h: определить число повторных попыток
4Ah: выбрать кодовую страницу
6Ah: определить активную кодовую страницу
4Ch: начало подготовки кодовой страницы
4Dh: конец подготовки кодовой страницы
6Bh: получить список готовых кодовых страниц
5Fh: установить информацию о дисплее
7Fh: получить информацию о дисплее
+OFh: 4 байта - не используются
+13h: 4 байта - адрес структуры данных IOCTL - соответствует структуре, передающейся в DS:DX для INT 21h, AX = 440Ch
19h: поддержка функций IOCTL (если установлены биты 6 и 7 атрибута)
+ODh: байт - категория устройства
+OEh: байт - код подфункции
Если эта комбинация подфункции и категории устройства не поддерживается драйвером — надо вернуть ошибку 03h в слове состояния.

Итак, теперь мы можем создать полноценный драйвер символьного устройства. Упрощая задачу, реализуем только функции чтения из устройства и будем возвращать соответствующие ошибки для других функций.

Еще одно отличие этого примера - в нем показано, как совместить в одной программе исполняемый файл типа EXE и драйвер устройства. Если такую программу запустить обычным образом, она будет выполняться, начиная со своей точки входа (метка start в нашем примере), а если ее загрузить из CONFIG.SYS, DOS будет считать драйвером участок программы, начинающийся со смещения 0.

```
; rot13.asm
; Драйвер символьного устройства, выводящий посылаемые ему символы на экран
; после выполнения над ними преобразования ROT13
; (каждая буква английского алфавита смещается на 13 позиций).
; Реализованы только функции записи в устройство.
;
; Пример использования:
; copy encrypted.txt $rot13
; Загрузка - из CONFIG.SYS
; DEVICE=c:\rot13.exe,
; если rot13.exe находится в директории C:\.
;
        .model small                    ; Модель для EXE-файла.
        .code
        .186                            ; Для pusha/popa.
        org     0                       ; Код драйвера начинается с CS:0000.
                dd      -1              ; Адрес следующего драйвера.
                dw      0A800h.         ; Атрибуты нашего устройства.
                dw      offset strategy ; Адрес процедуры стратегии.
                dw      offset interrupt ; Адрес процедуры прерывания.
                db      "$ROT13",20h,20h ; Имя устройства, дополненное
                                        ; пробелами до восьми символов.
request         dd      ?               ; Сюда процедура стратегии будет писать
                                        ; адрес буфера запроса.

; Таблица адресов обработчиков для всех команд.
command_table   dw      offset init             ; 00h
                dw      3 dup(offset unsupported); 01, 02, 03
                dw      2 dup(offset read)      ; 04, 05
                dw      2 dup(offset unsupported); 06, 07
                dw      2 dup(offset write)     ; 08h, 09h
                dw      6 dup(offset unsupported); 0Ah, 0Bh, 0Ch, 0Dh, 0Eh, 0Fh
                dw      offset write            ; 10h
                dw      2 dup(offset invalid)   ; 11h, 12h
                dw      offset unsupported      ; 13h
                dw      3 dup(offset invalid)   ; 14h, 15h, 16h
                dw      3 dup(offset unsupported); 17h, 18h, 19h

; Процедура стратегии - одна и та же для всех драйверов.
strategy        proc far
        mov     word ptr cs:request,bx
        mov     word ptr cs:request+2,es
        ret
strategy        endp

; Процедура прерывания.
interrupt       proc    far
        pushf
        pusha                           ; Сохранить регистры
```

```
        push    ds                      ; и, на всякий случай, флаги.
        push    es

        push    cs
        pop     ds                      ; DS = наш сегментный адрес.
        les     si,dword ptr request    ; ES:SI = адрес буфера запроса.
        xor     bx,bx
        mov     bl,byte ptr es:[si+2]   ; BX = номер функции.
        cmp     bl,19h                  ; Проверить, что команда
                                        ; в пределах 00 - 19h.
        jbe     command_ok
        call    invalid                 ; Если нет - выйти с ошибкой.
        jmp     short interrupt_end
command_ok:                             ; Если команда находится в пределах 00 - 19h,
        shl     bx,1                    ; умножить ее на 2, чтобы получить
                                        ; смещение в таблице слов command_table,
        call    word ptr command_table[bx]; и вызвать обработчик.
interrupt_end:
        cmp     al,0                    ; AL = 0, если не было ошибок.
        je      no_error
        or      ah,80h                  ; Если была ошибка - установить бит 15 в AX.
no_error:
        or      ah,01h                  ; В любом случае установить бит 8
        mov     word ptr es:[si+3],ax   ; и записать слово состояния.
        pop     es
        pop     ds
        popa
        popf
        ret
interrupt       endp

; Обработчик команд, предназначенных для блочных устройств.
unsupported     proc    near
        xor     ax,ax                   ; Не возвращать никаких ошибок.
        ret
unsupported     endp

; Обработчик команд чтения.
read            proc    near
        mov     al,0Bh                  ; Общая ошибка чтения.
        ret
read            endp

; Обработчик несуществующих команд.
invalid         proc    near
        mov     ax,03h                  ; Ошибка "неизвестная команда".
        ret
invalid         endp

; Обработчик функций записи.
write           proc    near
        push    si
```

```
        mov     cx,word ptr es:[si+12h]  ; Длина буфера в CX.
        jcxz    write_finished           ; Если это 0 - нам делать нечего.
        lds     si,dword ptr es:[si+0Eh] ; Адрес буфера в DS:SI.

; Выполнить ROT13-преобразование над буфером.
        cld
rot13_loop:                              ; Цикл по всем символам буфера.
        lodsb                            ; AL = следующий символ из буфера в ES:SI.

        cmp     al,'A'                   ; Если он меньше "A",
        jl      rot13_done               ; это не буква.
        cmp     al,'Z'                   ; Если он больше "Z",
        jg      rot13_low                ; может быть, это маленькая буква.
        cmp     al,("A"+13)              ; Иначе: если он больше "A" + 13,
        jge     rot13_dec                ; вычесть из него 13,
        jmp     short rot13_inc          ; а в противном случае - добавить.
rot13_low:
        cmp     al,'a'                   ; Если символ меньше "a",
        jl      rot13_done               ; это не буква.
        cmp     al,'z'                   ; Если символ больше "z" -
        jg      rot13_done               ; то же самое.
        cmp     al,("a"+13)              ; Иначе: если он больше "a" + 13,
        jge     rot13_dec                ; вычесть из него 13, иначе:
rot13_inc:
        add     al,13                    ; добавить 13 к коду символа,
        jmp     short rot13_done
rot13_dec:
        sub     al,13                    ; вычесть 13 из кода символа,
rot13_done:
        int     29h                      ; вывести символ на экран
        loop    rot13_loop               ; и повторить для всех символов.

write_finished:
        xor     ax,ax                    ; Сообщить, что ошибок не было.
        pop     si
        ret
write           endp

; Процедура инициализации драйвера.
init            proc    near
        mov     ah,9                     ; Функция DOS 09h.
        mov     dx,offset load_msg       ; DS:DX - сообщение об установке.
        int     21h                      ; Вывод строки на экран.
        mov     word ptr es:[si+0Eh],offset init ; Записать адрес
        mov     word ptr es:[si+10h],cs  ; конца резидентной части.
        xor     ax,ax                    ; Ошибок не произошло.
        ret
init            endp

; Сообщение об установке драйвера.
loadjnsg        db      "ROT13 device driver loaded",0Dh,0Ah,'$'
```

```
start:                                  ; Точка входа EXE-программы.
        push    cs
        pop     ds
        mov     dx,offset exe_msg       ; DS:DX - адрес строки.
        mov     ah,9                    ; Функция DOS.
        int     21h                     ; Вывод строки на экран.
        mov     ah,4Ch                  ; Функция DOS 4Ch.
        int     21h                     ; Завершение EXE-программы. .

; Строка, которая выводится при запуске не из CONFIG.SYS:
exejnsg         db      "Эту программу надо загружать, как драйвер устройства из"
                db      "CONFIG.SYS",0Dh,0Ah,'$'

        .stack

        end     start
```

5.11.2. Блочные устройства

Блочными устройствами называются такие устройства, на которых DOS может организовать файловую систему. DOS не работает напрямую с дисками через BIOS, а только с драйверами блочных устройств, каждое из которых представляется системе как линейный массив секторов определенной длины (обычно 512 байт) с произвольным доступом (для BIOS, к примеру, диск - это четырехмерный массив секторов, дорожек, цилиндров и головок). Каждому загруженному устройству DOS присваивает один или несколько номеров логических дисков, которые соответствуют буквам, используемым для обращения к ним. Так, стандартный драйвер дисков получает буквы A, B, C и так далее, по числу видимых разделов на диске.

Рассмотрим атрибуты и команды, передающиеся блочным устройствам.

Атрибуты:

- бит 15: 0 (признак блочного устройства)
- бит 14: поддерживаются IOCTL-чтение и запись
- бит 13: не требует копию первого сектора FAT, чтобы построить BPB
- бит 12: сетевой диск
- бит 11: поддерживает команды открыть/закрыть устройство и проверить, является ли устройство сменным
- биты 10-8: 000
- бит 7: поддерживается проверка поддержки IOCTL
- бит 6: поддерживается обобщенный IOCTL и команды установить и определить номер логического диска
- биты 5-2: 0000
- бит 1: поддерживаются 32-битные номера секторов
- бит 0: O

Команды и структура переменной части буфера запроса для них (только то, что отличается от аналогичных структур для символьных устройств):

00h: инициализация
+ODh: - байт количество устройств, которые поддерживает драйвер
+12h: 4 - байта дальний адрес массива BPB-структур (по одной для каждого устройства)

BPB - это 25-байтная структура (53 для FAT32), которая описывает блочное устройство. Ее можно найти по смещению 0Bh от начала нулевого сектора на любом диске:

+0: 2 байта - число байтов в секторе (обычно 512)
+2: байт - число секторов в кластере (DOS выделяет пространство на диске для файлов не секторами, а обычно более крупными единицами - кластерами. Даже самый маленький файл занимает один кластер)
+3: 2 байта - число секторов до начала FAT (обычно один - загрузочный)
+5: байт - число копий FAT (обычно 2) (FAT - это список кластеров, в которых расположен каждый файл. DOS делает вторую копию, чтобы можно было восстановить диск, если произошел сбой во время модификации FAT)
+6: 2 байта - максимальное число файлов в корневой директории
+8: 2 байта - число секторов на устройстве (если их больше 65 536 - здесь записан 0)
+OAh: байт - описатель носителя (OF8h - для жестких дисков, 0F0h - для дискет на 1,2 и 1,44 Мб, а также других устройств)
+OBh: 2 байта - число секторов в одной копии FAT (0, если больше 65 535)
+0Dh:2 байта - число секторов на дорожке (для доступа средствами BIOS)
+OFh: 2 байта - число головок (для доступа средствами BIOS)
+11h: 4 байта - число скрытых секторов
+15h: 4 байта - 32-битное число секторов на диске

(следующие поля действительны только для дисков, использующих FAT32)
+16h: 4 байта – 32-битное число секторов в FAT
+1Dh:байт - флаги
бит 7: не обновлять резервные копии FAT
биты 3-0: номер активной FAT, если бит 7 = 1
+1Fh: 2 байта - версия файловой системы (0000h для Windows 95 OSR2)
+21h: 4 байта - номер кластера корневой директории
+25h: 2 байта - номер сектора с информацией о файловой системе (0FFFFh, если он отсутствует)
+27h: 2 байта - номер сектора запасной копии загрузочного сектора (OFFFFh, если отсутствует)
+29h: 12 байт – зарезервировано

Для всех остальных команд в поле буфера запроса со смещением +1 размещается номер логического устройства из числа обслуживаемых драйвером, к которому относится команда:

01h: проверка носителя
+0Dh: байт - на входе - описатель носителя
на выходе - 0FFh, если диск был сменен
01h, если диск не был сменен
00h, если это нельзя определить
+0Fh: 4 байта - адрес ASCIZ-строки с меткой диска (если установлен бит 11 в атрибуте)
02h: построить BPB
+0Dh: описатель носителя
+0Eh: 4 байта - на входе - дальний адрес копии первого сектора FAT
на выходе - дальний адрес BPB
03h: IOCTL-чтение (если установлен бит 14 атрибута)
04h: чтение из устройства
+0Dh: байт - описатель носителя
+12h: 2 байта - на входе - число секторов, которые надо прочитать
на выходе - число прочитанных секторов
+16h: 2 байта - первый сектор (если больше 65 535 - здесь 0FFFFh)
+18h: 4 байта — на выходе - адрес метки диска, если произошла ошибка 0Fh
+1Ch: 4 байта – первый сектор
08h: запись в устройство
структура буфера аналогична 04h с точностью до замены чтения на запись
09h: запись в устройство с проверкой
аналогично 08h
0Ch: IOCTL-запись (если установлен бит 14 атрибута)
0Dh: открыть устройство (если установлен бит 11 атрибута)
0Eh: закрыть устройство (если установлен бит 11 атрибута)
0Fh: проверка наличия сменного диска (если установлен бит 11 атрибута):
драйвер должен установить бит 9 слова состояния, если диск сменный, и сбросить, если нет
13h: обобщенный IOCTL (если установлен бит 6 атрибута)
+0Dh: байт - категория устройства:
08h: дисковое устройство
48h: дисковое устройство с FAT32
+0Eh: код - подфункции:
40h: установить параметры
60h: прочитать параметры
41h: записать дорожку
42h: отформатировать и проверить дорожку
62h: проверить дорожку
46h: установить номер тома
66h: считать номер тома
47h: установить флаг доступа
67h: прочитать флаг доступа
68h: определить тип носителя (DOS 5.0+)

4Ah: заблокировать логический диск (Windows 95)
6Ah: разблокировать логический диск (Windows 95)
4Bh: заблокировать физический диск (Windows 95)
6Bh: разблокировать физический диск (Windows 95)
6Ch: определить флаг блокировки (Windows 95)
6Dh: перечислить открытые файлы (Windows 95)
6Eh: найти файл подкачки (Windows 95)
6Fh: получить соотношение логических и физических дисков (Windows 95)
70h: получить текущее состояние блокировки (Windows 95)
71h: получить адрес первого кластера (Windows 95)
+ 13h: адрес структуры (аналогично INT 21h AX = 440Dh)

17h: определить логический диск (если установлен бит 6 атрибута)
+01h: байт - на входе - номер устройства
на выходе - его номер диска (1-A, 2-B)

18h: установить логический диск (если установлен бит 6 атрибута)
+01h: байт - номер устройства
(команды 17h и 18h позволяют DOS обращаться к одному и тому же дисководу как к устройству A: и как к устройству B)

19h: поддержка функций IOCTL (если установлены биты 6 и 7 атрибута)

Для написания своего драйвера блочного устройства можно пользоваться схемой, аналогичной символьному драйверу из предыдущей главы. Единственное важное отличие - процедура инициализации должна будет подготовить и заполнить BPB, а также сообщить DOS число устройств, для которых действует этот драйвер.

Глава 6. Программирование в защищенном режиме

Все, о чем рассказано до этой главы, рассчитано на работу под управлением DOS в реальном режиме процессора (или в режиме V86), унаследованном еще с семидесятых годов. В этом режиме процессор неспособен адресоваться к памяти выше границы первого мегабайта. Более того, так как для адресации используются 16-битные смещения (PM), невозможно работать с массивами больше 65 536 байт. Защищенный режим лишен этих недостатков, в нем можно адресоваться к участку памяти размером 4 Гб как к одному непрерывному массиву и вообще забыть о сегментах и смещениях. Этот режим намного сложнее реального, поэтому, чтобы переключить в него процессор и поддерживать работу, надо написать небольшую операционную систему. Кроме того, если процессор уже находится под управлением какой-то операционной системы, которая перевела его в защищенный режим, например Windows 95, она, скорее всего, не разрешит программе устранить себя от управления компьютером. С этой целью были разработаны специальные интерфейсы, позволяющие программам, запущенным в режиме V86 в DOS, переключаться в защищенный режим простым вызовом соответствующего прерывания - VCPI и DPMI.

6.1. Адресация в защищенном режиме

Прежде чем познакомиться с программированием в защищенном режиме, рассмотрим механизм адресации, применяющийся в нем. Так же как и в реальном режиме, адрес складывается из адреса начала сегмента и относительного смещения, но если в реальном режиме адрес начала сегмента просто лежал в соответствующем сегментном регистре, деленный на 16, то в защищенном режиме не все так просто. В сегментных регистрах находятся специальные 16-битные структуры, называемые *селекторами* и имеющие следующий вид:

биты 15-3: номер дескриптора в таблице
бит 2: индикатор таблицы 0/1 - использовать GDT/LDT
биты 1-0: уровень привилегий запроса (RPL)

Уровень привилегий запроса — это число от 0 до 3, указывающее степень защиты сегмента, для доступа к которому применяется данный селектор. Если программа имеет более высокий уровень привилегий, при использовании этого сегмента привилегии понизятся до RPL. Уровни привилегий и весь механизм защиты в защищенном режиме нам пока не потребуются.

GDT и LDT - таблицы глобальных и локальных дескрипторов соответственно. Это таблицы 8-байтных структур, называемых *дескрипторами* сегментов, где и находится начальный адрес сегмента вместе с другой важной информацией:

слово 3 (старшее):
  биты 15-8: биты 31–24 базы
  бит 7: бит гранулярности (0 - лимит в байтах, 1 - лимит в 4-килобайтных единицах)
  бит 6: бит разрядности (0/1 - 16-битный/32-битный сегмент)
  бит 5: 0
  бит 4: зарезервировано для операционной системы
  биты 3-0: биты 19-16 лимита
слово 2:
  бит 15: бит присутствия сегмента
  биты 14-13: уровень привилегий дескриптора (DPL)
  бит 12: тип дескриптора (0 - системный, 1 - обычный)
  биты 11–8: тип сегмента
  биты 7-0: биты 23-16 базы
слово 1: биты 15-0 базы
слово 0 (младшее): биты 15-0 лимита

Два основных поля структуры, которые нам интересны, - это база и лимит сегмента. База представляет линейный 32-битный адрес начала сегмента, а лимит - 20-битное число, которое равно размеру сегмента в байтах (от 1 байта до 1 мегабайта), если бит гранулярности сброшен в ноль, или в единицах по 4096 байт (от 4 Кб до 4 Гб), если он установлен в 1. Числа отсчитываются от нуля, так что лимит 0 соответствует сегменту длиной 1 байт, точно так же, как база 0 соответствует первому байту памяти.

Остальные элементы дескриптора выполняют следующие функции:

1. Бит разрядности: для сегмента кода этот бит указывает на разрядность операндов и адресов по умолчанию. То есть в сегменте с этим битом, установленным в 1, все команды будут интерпретироваться как 32-битные, а префиксы изменения разрядности адреса или операнда будут превращать их в 16-битные, и наоборот. Для сегментов данных бит разрядности управляет тем, какой регистр (SP или ESP) используют команды, работающие с этим сегментом данных как со стеком.
2. Поле DPL определяет уровень привилегий сегмента.
3. Бит присутствия указывает, что сегмент реально есть в памяти. Операционная система может выгрузить содержимое сегмента из памяти на диск и сбросить бит присутствия, а когда программа попытается к нему обратиться, произойдет исключение, обработчик которого снова загрузит содержимое данного сегмента в память.
4. Бит типа дескриптора - если он равен 1, сегмент является обычным сегментом кода или данных. Если этот бит - 0, дескриптор является одним из 16 возможных видов, определяемых полем типа сегмента.

5. Тип сегмента: для системных регистров в этом поле находится число от 0 до 15, определяющее тип сегментов (LDT, TSS, различные шлюзы), которые рассмотрены в главе 9. Для обычных сегментов кода и данных эти четыре бита выполняют следующие функции:

бит 11: 0 - сегмент данных, 1 - сегмент кода
бит 10: для данных - бит направления роста сегмента
для кода - бит подчинения
бит 9: для данных - бит разрешения записи
для кода - бит разрешения чтения
бит 8: бит обращения

6. Бит обращения устанавливается в 1 при загрузке селектора этого сегмента в регистр.
7. Бит разрешения чтения/записи выбирает разрешаемые операции с Сегментом - для сегмента кода это могут быть выполнение или выполнение/чтение, а для сегмента данных — чтение или чтение/запись.
8. Бит подчинения указывает, что данный сегмент кода является подчиненным. Это значит, что программа с низким уровнем привилегий может передать управление в сегмент кода и текущий уровень привилегий не изменится.
9. Бит направления роста сегмента обращает смысл лимита сегмента. В сегментах с этим битом, сброшенным в ноль, разрешены абсолютно все смещения от 0 до лимита, а если этот бит 1, то допустимы все смещения, кроме от О до лимита. Про такой сегмент говорят, что он растет сверху вниз, так как если лимит, например, равен –100, допустимы смещения от -100 до 0, а если лимит увеличить, станут допустимыми еще меньшие смещения.

Для обычных задач программирования нам не потребуется все многообразие возможностей адресации. Единственное, что нам нужно, - это удобный неограниченный доступ к памяти. Поэтому мы будем рассматривать простую модель памяти - так называемую модель flat, где базы всех регистров установлены в ноль, а лимиты - в 4 Гб. Именно в такой ситуации окажется, что можно забыть о сегментации и пользоваться только 32-битными смещениями.

Чтобы создать flat-память, нам потребуются два дескриптора с нулевой базой и максимальным лимитом - один для кода и один для данных.

Дескриптор кода:
лимит OFFFFFh
база 00000000h
тип сегмента OFAh
бит присутствия = 1
уровень привилегий = 3 (минимальный)
бит типа дескриптора = 1
тип сегмента: 1010b (сегмент кода, для выполнения/чтения)

бит гранулярности = 1
бит разрядности = 1
```
db      OFFh, OFFh, Oh, Oh, Oh, 0FAh, OCFh, Oh
```

Дескриптор данных:
лимит 0FFFFFh
база 00000000h
бит присутствия = 1
уровень привилегий = 3 (минимальный)
бит типа дескриптора = 1
тип сегмента: 0010b (сегмент данных, растет вверх, для чтения/записи)
бит гранулярности = 1
бит разрядности = 1
```
db      OFFh, OFFh, Oh, Oh, Oh, OF2h, OCFh, Oh
```

Для того чтобы процессор знал, где искать дескрипторы, операционная система собирает их в таблицы, которые называются GDT (таблица глобальных дескрипторов - может быть только одна) и LDT (таблица локальных дескрипторов - по одной на каждую задачу), и загружает их при помощи привилегированных команд процессора. Так как мы пока не собираемся создавать операционные системы, нам нужно только подготовить дескриптор и вызвать соответствующую функцию VCPI или DPMI.

Заметим также, что адрес, который получается при суммировании базы сегмента и смещения, называется *линейным адресом* и может не совпадать с физическим, если дело происходит в операционной системе, реализующей виртуальную память с помощью специально предусмотренного в процессорах Intel страничного механизма виртуализации памяти.

6.2. Интерфейс VCPI

Спецификация интерфейса VCPI была создана в 1989 году (вскоре после появления процессора 80386) компаниями Phar Lap Software и Quaterdeck Office Systems. Программа, применяющая VCPI для переключения в защищенный режим, должна поддерживать полный набор системных таблиц - GDT, LDT, IDT, таблицы страниц и т. д. То есть фактически VCPI-сервер обеспечивает следующее: процессор находится в защищенном режиме, и различные программы, пользующиеся им, не конфликтуют между собой.

VCPI является своего рода расширением интерфейса EMS, и все обращения к нему выполняются при помощи прерывания EMS, INT 67h с AH = 0DEh и кодом подфункции VCPI в AL.

INT 67h AX = 0DE00h: Проверка наличия VCPI
Вход: AX = ODEOOh
Выход: AH = 00, если VCPI есть
BH, BL - версия (старшая, младшая цифры)

INT 67h AX = 0DE01h: Получить точку входа VCPI
Вход: AX = ODEOIh
ES:DI = адрес 4-килобайтного буфера для таблицы страниц
DS:SI = адрес таблицы дескрипторов
Выход: AH = 0, если нет ошибок
DS:SI: первые три дескриптора заполняются дескрипторами VCPI-сервера
ES:DI: адрес первой не используемой сервером записи в таблице страниц
EBX: адрес точки входа VCPI относительно сегмента, дескриптор которого лежит первым в таблице DS:SI. Можно делать far call из 32-битного защищенного режима на этот адрес с AX = ODEOOh - 0DE05h, чтобы пользоваться функциями VCPI из защищенного режима

INT 67h AX = ODEOCh: VCPI: переключиться в защищенный режим (для вызова из V86)
Вход: AX - ODEOCh
ESI = линейный адрес таблицы со значениями для системных регистров (в первом мегабайте)
+00h: 4 байта — новое значение CR3
+04h: 4 байта - адрес 6-байтного значения для GDTR
+08h: 4 байта - адрес 6-байтного значения для IDTR
+OCh: 2 байта - LDTR
+0Eh: 2 байта - TR
+10h: 6 байт - CS:EIP - адрес точки входа
прерывания должны быть запрещены
Выход: Загружаются регистры GDTR, IDTR, LDTR, TR. Теряются регистры EAX, ESI, DS, ES, FS, GS. SS:ESP указывает на стек размером 16 байт, и его надо изменить, прежде чем снова разрешать прерывания

Точка входа VCPI, AX = ODEOCh: Переключиться в режим V86 (для вызова из PM)
Вход: Перед передачей управления командой call в стек надо поместить регистры в следующем порядке (все значения - двойные слова): GS, FS, DS, ES, SS, ESP, O, CS, EIP. Прерывания должны быть запрещены
Выход: Сегментные регистры загружаются, значение EAX не определено, прерывания запрещены

Остальные функции VCPI:

INT67hAX = 0DE02h: Определить максимальный физический адрес
Вход: AX = ODE02h
Выход: AH = 0, если нет ошибок
EDX = физический адрес самой старшей 4-килобайтной страницы, которую можно выделить

INT 67hAX = 0DE03h: Определить число свободных страниц
Вход: AX = ODE03h
Выход: AH = 0, если нет ошибок
EDX = число свободных 4-килобайтных страниц для всех задач

INT 67hAX = 0DE04h: Выделить 4-килобайтную страницу (обязательно надо вызвать DE05h)
Вход: AX = ODE04h
Выход: AH = 0, если нет ошибок
EDX = физический адрес выделенной страницы

INT 67h AX = 0DE05h: Освободить 4-килобайтную страницу
Вход: AX = 0DE05h
EDX = физический адрес страницы
Выход: AH = 6, если нет ошибок

INT 67h AX = 0DE06h: Определить физический адрес 4-килобайтной страницы в первом мегабайте
Вход: AX = ODE06h
CX = Линейный адрес страницы, сдвинутый вправо на 12 бит
Выход: AH = 0, если нет ошибок
EDX = физический адрес страницы

INT 67hAX = 0DE07h: Прочитать регистр CRO
Вход: AX = ODE07h
Выход: AH = 0, если нет ошибок
EBX = содержимое регистра CRO

INT 67hAX = 0DE08h: Прочитать регистры DRO - DR7
Вход: AX = 0DE08h
ES:DI = буфер на 8 двойных слов
Выход: AH = 0, если нет ошибок, в буфер не записываются DR4 и DR5

INT 67hAX = 0DE09h: Записать регистры DRO - DR7
Вход: AX = ODE09h
ES:DI = буфер на 8 двойных слов с новыми значениями для регистров
Выход: AH = 0, если нет ошибок (DR4 и DR5 не записываются)

INT 67h AX = ODEOAh: Определить отображение аппаратных прерываний
Вход: AX = ODEOAh
Выход: AH = 0, если нет ошибок
BX = номер обработчика для IRQO
CX = номер обработчика для IRQ8

INT 67hAX = ODEOBh: Сообщить VCPI-серверу новое отображение аппаратных прерываний (вызывается после перепрограммирования контроллера прерываний)
Вход: AX = ODEOBh
BX = номер обработчика для IRQO
CX = номер обработчика для IRQ8
Выход: AH = 0, если нет ошибок

Итак, чтобы использовать защищенный режим с VCPI, фактически надо уметь программировать его самостоятельно. Например, чтобы вызвать прерывание DOS

или BIOS, нам пришлось бы переключаться в режим V86, вызывать прерывание и затем возвращаться обратно. Естественно, этот интерфейс не получил широкого развития и был практически повсеместно вытеснен более удобным DPMI.

6.3. Интерфейс DPMI

Спецификация DPMI создана в 1990-1991 годах и представляет собой развитую систему сервисов, позволяющих программам переключаться в защищенный режим, вызывать обработчики прерываний BIOS и DOS, передавать управление другим процедурам, работающим в реальном режиме, устанавливать обработчики аппаратных и программных прерываний и исключений, работать с памятью с разделяемым доступом, устанавливать точки останова и т. д. Операционные системы, такие как Windows 95 или Linux, предоставляют DPMI-интерфейс для программ, запускаемых в DOS-задачах. Многочисленные расширители DOS, о которых говорится в следующем разделе, поддерживают DPMI для DOS-программ, запускаемых в любых условиях, так что сейчас можно считать DPMI основным интерфейсом, на который следует ориентироваться при программировании 32-битных приложений для DOS.

6.3.1. Переключение в защищенный режим

Все основные сервисы DPMI доступны только в защищенном режиме через прерывание INT 31h, следовательно, переключение режимов - первое, что должна сделать программа.

INT 2Fh AX = 1687h: Функция DPMI: получить точку входа *в* защищенный режим
Вход: AX = 1687h
Выход: AX = 0, если DPMI присутствует
BX: бит 0 = 1, если поддерживаются 32-битные программы; 0, если нет
CL: тип процессора (02 - 80286, 03 - 80386 и т. д.)
DH:DL - версия DPMI в двоичном виде (обычно 00:90 (00:5Ah) или 01:00)
SI = размер временной области данных, требуемой для переключения в 16-байтных параграфах
ES:DI = адрес процедуры переключения в защищенный режим

Вызвав эту функцию, программа должна выделить область памяти размером SI×16 байт и выполнить дальний CALL на указанный адрес. Единственные входные параметры - регистр ES, который содержит сегментный адрес области данных для DPMI и бит 0 регистра AX. Если этот бит 1 - программа собирается стать 32-битным приложением, а если 0 - 16-битным. Если по возвращении из процедуры установлен флаг CF, переключения не произошло и программа все еще в реальном режиме. Если CF = 0, программа переключилась в защищенный режим и в сегментные регистры загружены следующие селекторы:
CS: 16-битный селектор с базой, совпадающей со старым CS, и лимитом 64 Кб
DS: селектор с базой, совпадающей со старым DS, и лимитом 64 Кб

SS: селектор с базой, совпадающей со старым SS, и лимитом 64 Кб
ES: селектор с базой, совпадающей с началом блока PSP, и лимитом 100h
FS и GS - 0

Разрядность сегментов данных определяется заявленной разрядностью программы. Остальные регистры сохраняются, а для 32-битных программ старшее слово ESP обнуляется. По адресу ES:[002Ch] записывается селектор сегмента, содержащего переменные окружения DOS.

После того как произошло переключение, можно загрузить собственные дескрипторы для сегментов кода и данных и перейти в режим с моделью памяти flat.

Перечислим теперь основные функции, предоставляемые DPMI для работы с дескрипторами и селекторами.

6.3.2. Функции DPMI управления дескрипторами

INT 31h, AX = 0: Выделить локальные дескрипторы
Вход: AX = 0
CX = количество необходимых дескрипторов
Выход: если CF = 0, AX = селектор для первого из заказанных дескрипторов

Эта функция только выделяет место в таблице LDT, создавая в ней дескриптор сегмента данных с нулевыми базой и лимитом, так что пользоваться им пока нельзя.

INT 31h, AX = 1: Удалить локальный дескриптор
Вход: AX = 1
BX = селектор
Выход: CF = 0, если не было ошибки

Эта функция влияет на дескрипторы, созданные функцией 0, и при переключении в защищенный режим, но не на дескрипторы, созданные функцией 2.

INT 31h, AX = 2: Преобразовать сегмент в дескриптор
Вход: AX = 2
BX = сегментный адрес (0A000h - для видеопамяти, 0040h - для данных BIOS)
Выход: если CF = 0, AX = готовый селектор на сегмент, начинающийся с указанного адреса, и с лимитом 64 Кб

Так программы в защищенном режиме могут обращаться к различным областям памяти ниже границы 1 Мб, например для прямого вывода на экран.

INT 31h, AX = 6: Определить базу сегмента
Вход: AX = 6
BX = селектор
Выход: если CF = 0, CX:DX = 32-битный линейный адрес начала сегмента

INT 31h, AX = 7: Сменить базу сегмента
Вход: AX = 7
BX = селектор
CX:DX = 32-битная база
Выход: CF = 0, если не было ошибок

INT 31h, AX = 8: Сменить лимит сегмента
Вход: AX = 8
BX = селектор
CX:DX = 32-битный лимит (длина сегмента минус 1)
Выход: CF = 0, если не было ошибок
(чтобы определить лимит сегмента, можно пользоваться командой LSL)

6*INT 31h,AX = 9:* Сменить права доступа сегмента
Вход: AX = 9
BX = селектор
CL = права доступа/тип (биты 15-8 слова 2 дескриптора)
CH = дополнительные права (биты 7-4 соответствуют битам 7-4 слова 3 дескриптора, биты 3-0 игнорируются)
Выход: CF = 0, если не было ошибок
(чтобы определить права доступа сегмента, можно пользоваться командой LAR)

INT 31h, AX = 0Ah: Создать копию дескриптора
Вход: AX = OOOAh
BX = селектор (сегмента кода или данных)
Выход: если CF = O, AX = селектор на сегмент данных с теми же базой и лимитом

INT 31h, AX = 0Bh: Прочитать дескриптор
Вход: AX = OOOBh
BX = селектор
ES:EDI = селекторхмещение 8-байтного буфера
Выход: если CF = 0, в буфер помещен дескриптор

INT 31h, AX = 0Ch: Загрузить дескриптор
Вход: AX = OOOCh
BX = селектор
ES:EDI = адрес 8-байтного дескриптора
Выход: CF = 0, если не было ошибок

INT 31 h, AX = 0Dh: Выделить определенный дескриптор
Вход: AX = 000Dh
BX = селектор на один из первых 16 дескрипторов
(значения селектора 04h - 7Ch)
Выход: CF = 0, если нет ошибок (CF = 1 и AX = 8011h, если этот дескриптор занят)

Данного набора функций, а точнее пары функций 00 и 0Ch, достаточно для того, чтобы полностью настроить режим flat (или любой другой) после переключения в защищенный режим. Но прежде чем это осуществить, нам надо познакомиться с тем, как в DMPI сделан вызов обработчиков прерываний реального режима, иначе наша программа просто не сможет себя проявить.

6.3.3. Передача управления между режимами в DPMI

Вызов любого программного прерывания, кроме INT 31h и INT 21h/AH = 4Ch (функция DPMI: завершение программы), приводит к тому, что DPMI-сервер

переключается в режим V86 и передает управление обработчику этого же прерывания, скопировав все регистры, кроме сегментных регистров и стека (состояние сегментных регистров не определено, а стек предоставляет сам DPMI-сервер). После того как обработчик прерывания возобновит управление, DPMI-сервер возвратиться в защищенный режим и вернет управление программе. Таким способом можно вызывать все прерывания, передающие параметры только в регистрах, например проверку нажатия клавиши или вывод символа на экран. Чтобы вызвать прерывание, использующее сегментные регистры, например вывод строки на экран, а также в других ситуациях, когда требуется обращение к процедуре, работающей в другом режиме, применяются следующие функции.

INT 31h, AX = 0300h: Вызов прерывания в реальном режиме

Вход: AX = 0300h
BH = O, BL = номер прерывания
CX = число слов из стека защищенного режима, которое будет скопировано в стек реального режима и обратно
ES:EDI = селекторхмещение структуры регистров v86_regs (см. ниже)

Выход: если CF = 0, структура в ES:EDI модифицируется

Значения регистров CS и IP в структуре v86_regs игнорируются. Вызываемый обработчик прерывания должен восстанавливать стек в исходное состояние (например, INT 25h и INT 26h этого не делают).

INT 31h, AX = 0301h: Вызов дальней процедуры в реальном режиме

Вход: AX = 0301h
BH = 0
CX = число слов из стека защищенного режима, которое будет скопировано в стек реального режима и обратно
ES:EDI = селектор:смещение структуры регистров v86_regs (см. ниже)

Выход: если CF = 0, структура в ES:EDI модифицируется

Вызываемая процедура должна заканчиваться командой RETF.

INT 31h, AX = 0302h: Вызов обработчика прерывания в реальном режиме

Вход: AX = 0302h
BH = 0
CX = число слов из стека защищенного режима, которое будет скопировано в стек реального режима и обратно
ES:EDI = селекторхмещение структуры регистров v86_regs (см. ниже)

Выход: если CF = 0, структура в ES:EDI модифицируется

Вызываемая процедура должна заканчиваться командой IRET.

Эти три функции используют следующую структуру данных для передачи значений регистров в реальный режим и обратно:

+00h: 4 байта - EDI
+04h: 4 байта - ESI
+08h: 4 байта - EBP
+0Ch: 4 байта - игнорируются
+10h: 4 байта – EBX

+14h: 4 байта - EDX
+18h: 4 байта - ECX
+1Ch: 4 байта – EAX
+20h: 2 байта - FLAGS
+22h: 2 байта - ES
+24h: 2 байта - DS
+26h: 2 байта - FS
+28h: 2 байта - GS
+2Ah: 2 байта - IP
+2Ch: 2 байта - CS
+2Eh: 2 байта - SP
+30h: 2 байта - SS

Значения SS и SP могут быть нулевыми, тогда DPMI-сервер сам предоставит стек для работы прерывания.

Кроме указанных функций, которые передают управление процедурам в реальном режиме из защищенного, существует механизм, позволяющий делать в точности обратное действие — передавать управление процедурам в защищенном режиме из реального.

INT 31h, AX = 0303h: Выделить точку входа для вызова из реального режима
Вход: AX = 0303h
DS:ESI = селекторхмещение процедуры в защищенном режиме (заканчивающейся IRET), которую будут вызывать из реального
ES:EDI = селекторхмещение структуры v86_regs, которая будет использоваться для передачи регистров
Выход: если CF = O, CX:DX = сегментхмещение точки входа

При передаче управления в такую процедуру DS:ESI указывает на стек реального режима, ES:EDI - на структуру v86_regs, SS:ESP - на стек, предоставленный DPMI-сервером, и остальные регистры не определены.

Количество точек входа, которыми располагает DPMI-сервер, ограничено, и неиспользуемые точки входа должны быть удалены при помощи следующей функции DPMI.

INT 31h, AX = 0304h: Освободить точку входа для вызова из реального режима
Вход: AX = 0304h
CX:DX = сегментхмещение точки входа
Выход: CF = 0, если точка входа удалена

6.3.4. Обработчики прерываний

Прежде чем мы рассмотрим первый пример программы, использующей DPMI, остановимся еще на одной группе его функций - операции с обработчиками прерываний. Когда происходит прерывание или исключение, управление передается сначала по цепочке обработчиков прерываний в защищенном режиме, последний

обработчик - стандартный DPMI - переходит в режим V86, а затем управление проходит по цепочке обработчиков прерывания в реальном режиме (обработчики прерываний и исключений в реальном режиме совладают).

INT 31h, AX = 0200h: Определить адрес реального обработчика прерывания
Вход: AX = 0200h
BL = номер прерывания
Выход: CF = 0 всегда, CX:DX - сегментхмещение обработчика прерывания в реальном режиме

INT 31h, AX = 0201h: Установить реальный обработчик прерывания
Вход: AX = 0201h
BL = номер прерывания
CX:DX = сегмент:смещение обработчика прерывания в реальном режиме
Выход: CF = 0 всегда

INT 31h, AX = 0204h: Определить адрес защищенного обработчика прерывания
Вход: AX = 0204h
BL = номер прерывания
Выход: CF = 0 всегда, CX:EDX = селекторхмещение обработчика

INT 31h, AX = 0205h: Установить защищенный обработчик прерывания
Вход: AX = 0205h
BL = номер прерывания
CX:EDX = селекторхмещение обработчика
Выход: CF = O

INT 31h, AX = 0202h: Определить адрес обработчика исключения
Вход: AX = 0202h
BL = номер исключения (00h - 1Fh)
Выход: если CF = O, CX:EDX = селекторхмещение обработчика исключения

INT 31h, AX = 0203h: Установить обработчик исключения
Вход: AX = 0203h
BL = номер исключения (OOh - 1Fh)
CX:EDX = селекторхмещение обработчика исключения
Выход: CF = 0, если не было ошибок

Если обработчик исключения передает управление дальше по цепочке на стандартный обработчик DPMI-сервера, следует помнить, что только исключения O, 1, 2, 3, 4, 5 и 7 передаются обработчикам из реального режима, а остальные исключения приводят к прекращению работы программы.

6.3.5. Пример программы

Теперь воспользуемся интерфейсом DPMI для переключения в защищенный режим и вывода строки на экран. Этот пример будет работать только в системах, предоставляющих DPMI для запускаемых в них DOS-программ, например в Windows 95.

```
; dpmiex.asm
; Выполняет переключение в защищенный режим средствами DPMI.
; При компиляции с помощью WASM необходима опция /D_WASM_.
        .386                    ; 32-битный защищенный режим появился в 80386.

; 16-битный сегмент - в нем выполняется подготовка и переход в защищенный режим.
RM_seg  segment byte    public  use16
        assume  cs:RM_seg, ds:PM_seg, ss:RM_stack

; Точка входа программы.
RM_entry:
; Проверить наличие DPMI.
        mov     ax,1687h        ; Номер 1678h.
        int     2Fh             ; Прерывание мультиплексора.
        test    ax,ax           ; Если AX не ноль,
        jnz     DPMI_error      ; произошла ошибка (DPMI отсутствует).
        test    bl,1            ; Если не поддерживается 32-битный режим,
        jz      DPMI_error      ; нам тоже нечего делать.

; Подготовить базовые адреса для будущих дескрипторов.
        mov     eax,PM_seg
        mov     ds,ax           ; DS - сегментный адрес PM_seg.
        shl     eax,4           ; EAX - линейный адрес начала сегмента PM_seg.
        mov     dword ptr PM_seg_addr,eax
        or      dword ptr GDT_flatCS+2,eax      ; Дескриптор для CS.
        or      dword ptr GDT_flatDS+2,eax      ; Дескриптор для DS.

; Сохранить адрес процедуры переключения DPMI.
        mov     word ptr DPMI_ModeSwitch,di
        mov     word ptr DPMI_ModeSwitch+2,es

; ES должен указывать на область данных для переключения режимов.
; У нас она будет совпадать с началом будущего 32-битного стека.
        add     eax,offset DPMI_data            ; Добавить к EAX смещение
        shr     eax,4           ; и превратить в сегментный адрес.
        inc     ax
        mov     es,ax

; Перейти в защищенный режим.
        mov     ax, 1           ; AX = 1 - мы будем 32-м приложением.
ifdef _WASM_
        db      67h             ; Поправка для wasm.
endif
        call    dword ptr DPMI_ModeSwitch
        jc      DPMI_error      ; Если переключения не произошло - выйти.

; Теперь мы находимся в защищенном режиме, но лимиты всех сегментов
; установлены на 64 Кб, а разрядности сегментов - на 16 бит.
; Нам надо подготовить два 32-битных селектора с лимитом 4 Гб -
; один для кода и один для данных.

        push    ds
        pop     es              ; ES вообще был сегментом PSP с лимитом 100h.
```

```
        mov     edi,offset GDT          ; EDI - адрес таблицы GDT.
; Цикл по всем дескрипторам в нашей GDT,
        mov     edx,1                   ; которых всего два (0, 1),
sel_loop:
        xor     ax,ax                   ; Функция DPMI 00.
        mov     cx,1
        int     31h                     ; Создать локальный дескриптор.
        mov     word ptr selectors[edx*2],ax    ; Сохранить селектор
        mov     bx,ax                           ; в таблице selectors.
        mov     ax,000Ch                ; Функция DPMI 0Ch.
        int     31h                     ; Установить селектор.
        add     di,8                    ; EDI - следующий дескриптор.
        dec     dx
        jns     sel_loop

; Загрузить селектор сегмента кода в CS при помощи команды RETF.
        push    dword ptr Sel_flatCS    ; Селектор для CS.
ifdef _WASM_
        db      066h
endif
        push    offset PM_entry         ; EIP
        db      066h                    ; Префикс размера операнда.
        retf                            ; Выполнить переход в 32-битный сегмент.

; Сюда передается управление, если произошла ошибка при инициализации DPMI
; (обычно, если DPMI просто нет).
DPMI_error:
        push    cs
        pop     ds
        mov     dx,offset nodpmijnsg
        mov.    ah,9h                   ; Вывод строки на экран.
        int     21h
        mov     ah,4Ch                  ; Конец EXE-программы.
        int     21h
nodpmi_msg      db      "Ошибка DPMI$"
RM_seg  ends

; Сегмент PM_seg содержит код, данные и стек для защищенного режима.
PM_seg  segment byte    public  use32
        assume  cs:PM_seg,ds:PM_seg,ss:PM_seg

; Таблица дескрипторов.
GDT     label   byte
; Дескриптор для CS.
GDT_flatCS      db      0FFh,0FFh,0h,0h,0h,0FAh,0CFh,0h
; Дескриптор для DS.
GDT_flatDS      db      0FFh,0FFh,0h,0h,0h,0F2h,0CFh,0h

; Точка входа в 32-битный режим - загружен только CS.
PM_entry:
        mov     ax,word ptr Sel_flatDS  ; Селектор для данных
        mov     ds,ax                   ; в DS,
```

```
        mov     es,ax                   ; в ES
        mov     ss,ax                   ; и в SS.
        mov     esp,offset PM_stack_bottom ; И установить стек.

;=============================================================
; Отсюда начинается текст собственно программы.
; Программа работает в модели памяти flat с ненулевой базой,
; база CS, DS, ES и SS совпадает и равна линейному адресу начала PM_seg,
; все лимиты - 4 Гб.

        mov     ax,0300h                ; Функция DPMI 0300h.
        mov     bx,0021h                ; Прерывание DOS 21h.
        xor     ecx,ecx                 ; Стек не копировать.
        mov     edi,offset v86_regs     ; ES:EDI - адрес v86_regs.
        int     31h                     ; Вызвать прерывание.

        mov     ah,4Ch                  ; Это единственный способ
        int     21h                     ; правильно завершить DPMI-программу.

hello_msg       db      "Hello world из 32-битного защищенного режима!$"

v86_regs:               ; Значения регистров для функции DPMI 0300h.
                dd      0,0,0,0,0       ; EDI, ESI, EBP, 0, EBX
v_86_edx        dd      offset hello_msg ; EDX
                dd      0               ; ECX
v86_eax         dd      0900h           ; EAX (AH = 09h, вывод строки на экран)
                dw      0,0             ; FLAGS, ES
v86_ds          dw      PM_seg          ; DS
                dw      0,0,0,0,0,0     ; FS, GS, 0, 0, SP, SS

; Различные временные переменные, нужные для переключения режимов.
DPMI_ModeSwitch dd      ?               ; Точка входа DPMI.
PM_seg_addr     dd      ?               ; Линейный адрес сегмента PM_seg.

; Значения селекторов.
selectors:
Sel_flatDS      dw      ?
Sel_flatCS      dw      ?

; Стек для нашей 32-битной программы
DPMI_data:              ; и временная область данных DPMI одновременно.
                db      16384 dup (?)
PM_stack_bottom:
PM_seg  ends

; Стек 16-битной программы, который использует DPMI-сервер при переключении режимов.
; Windows 95 требует 16 байт,
; CWSDPMI требует 32 байта,
; QDPMI требует 96 байт.
; Мы выберем по максимуму.
RM_stack segment byte stack "stack" use16
                db      96 dup (?)
RM_stack ends
        end     RM_entry ; точка входа для DOS - RM_entry
```

Несмотря на то что DPMI разрешает пользоваться многими прерываниями напрямую и всеми через функцию 0300h, он все равно требует некоторой подготовки для переключения режимов. Кроме того, программа, работающая с DPMI для переключения режимов, должна сочетать в себе 16- и 32-битный сегменты, что неудобно с точки зрения практического программирования. На самом деле для написания приложений, идущих в защищенном режиме под DOS, никто не применяет переключение режимов вообще - это делают специальные программы, называющиеся расширителями DOS.

6.4. Расширители DOS

Расширитель DOS (DOS Extender) - это средство разработки (программа, набор программ, часть компоновщика или просто объектный файл, с которым нужно компоновать свою программу), позволяющее создавать 32-битные приложения, которые запускаются в защищенном режиме с моделью памяти flat и с работающими функциями DPMI. Расширитель сам выполняет переключение в защищенный режим, причем, если в системе уже присутствует DPMI, VCPI или другие средства переключения в защищенный режим из V86, он пользуется ими, а если программа запускается в настоящем реальном режиме, DOS-расширитель переводит процессор в защищенный режим и осуществляет все необходимые действия для поддержания его работы. Кроме полного или частичного набора функций DPMI, расширители DOS обычно поддерживают некоторые функции прерывания 21h, за что и получили свое название. В частности, практически всегда поддерживается функция DOS 09h вывода строки на экран: в DS:EDX помещают селектор:смещение начала строки, и расширитель это правильно интерпретирует (многие DPMI-серверы, включая встроенный сервер Windows 95, тоже эмулируют данную функцию DOS).

Так как при старте программы расширитель должен первым получить управление, для выполнения переключения режимов его код нужно объединить с нашей программой на стадии компиляции или компоновки.

6.4.1. Способы объединения программы с расширителем

Первые популярные DOS-расширители, такие как Start32, Raw32, System64, 386Power, PMODE и другие, распространялись в виде исходных текстов (Start32 и PMODE оказали решающее влияние на развитие DOS-расширителей в целом). Чтобы использовать такой расширитель, надо скомпилировать его любым ассемблером в объектный файл, который необходимо скомпоновать вместе со своей программой. В большинстве случаев нужно назвать точку входа своей программы main или _main и закончить модуль директивой end без параметра, тогда DOS-расширитель получит управление первым и передаст его на метку main после того, как будут настроены все сегменты для модели памяти flat.

Самым популярным из профессиональных компиляторов, поддерживающих расширители DOS, стал Watcom C/C++. Он использует модификацию коммерческого DOS-расширителя DOS4G, названную DOS/4GW. Дело в том, что компоновщик wlink.exe поддерживает, среди большого числа различных форматов

вывода, формат линейных исполняемых файлов LE, применяющийся в операционной системе OS/2 (а также, с небольшими модификациями, для драйверов в Windows). Достаточно дописать файл в формате OS/2 LE в конец загрузчика DOS-расширителя, написанного соответствующим образом, чтобы потом его запускать. Загрузчик расширителя можно указать прямо в командной строке wlink (командой op stub) или скопировать позже. В комплект поставки расширителей часто входит специальная утилита, которая заменяет загрузчик, находящийся в начале такой программы, своим.

Чтобы скомпилировать, например, программу Ifbfire.asm, которую мы рассмотрим далее, нужно воспользоваться следующими командами:
Компиляция:

```
wasm Ifbfire.asm
```

Компоновка с DOS/4GW (стандартный расширитель, распространяемый с Watcom C):

```
wlink file Ifbfire.obj form os2 le op stub=wstub.exe
```

Компоновка с PMODE/W (самый популярный из бесплатных расширителей):

```
wlink file Ifbfire.obj form os2 le op stub=pmodew.exe
```

Компоновка с ZRDX (более строгий с точки зрения реализации):

```
wlink file Ifbfire.obj form os2 ls op stub=zrdx.exe
```

Компоновка с WDOSX (самый универсальный расширитель):

```
wlink file Ifbfire.obj form os2 le op stub=wdosxle.exe
```

И так далее.

К сожалению, формат исполняемых файлов DOS (так называемый формат MZ), в котором по умолчанию создают программы другие компиляторы, очень неудобен для объединения с расширителями, хотя универсальный расширитель WDOSX способен обработать и такой файл, и даже просто файл с 32-битным кодом без всяких заголовков (какой можно получить, создав COM-файл с директивой org 0), и файл в формате PE (см. главу 7), правда, не во всех случаях такие программы будут работать успешно.

И наконец, третий подход к объединению расширителя и программы можно видеть на примере DOS32, куда входит программа dlink.exe, являющаяся компоновщиком, который вполне подойдет вместо link, tlink или wlink, чтобы получить исполняемый файл, работающий с этим расширителем.

Тем не менее популярность подхода, используемого в Watcom, настолько высока, что подавляющее большинство программ, применяющих идею расширителей DOS, написано именно на Watcom C или на ассемблере для WASM.

Прежде чем мы сможем написать обещанный в разделе 4.5.2 пример программы, работающей с линейным кадровым буфером SVGA, познакомимся еще с двумя группами функций DPMI, которые нам потребуются.

6.4.2. Управление памятью в DPMI

INT 31h, AX = 0100h: Выделить память ниже границы 1 Мб
Вход: AX = 0100h
BX = требуемый размер в 16-байтных параграфах
Выход: если CF = O,
AX - сегментный адрес выделенного блока для использования в реальном режиме;
DX - селектор выделенного блока для применения в защищенном режиме

Обработчик этой функции выходит в V86 и вызывает функцию DOS 48h для выделения области памяти, которую потом можно использовать для передачи данных между нашей программой и обработчиками прерываний, возвращающими структуры данных в памяти.

INT 31h, AX = 0101 h: Освободить память ниже границы 1 Мб
Вход: AX = 0102h
DX = селектор освобождаемого блока
Выход: CF = 0, если не было ошибок

INT 31h, AX = 0102h: Изменить размер блока, выделенного функцией 0100h
Вход: AX = 0102h
BX = новый размер блока в 16-байтных параграфах
DX = селектор модифицируемого блока
Выход: CF = 0, если не было ошибок

INT 31h, AX = 0500h: Получить информацию о свободной памяти
Вход: AX = 0500h
ES:EDI = адрес 48-байтного буфера
Выход: CF = 0 всегда, буфер заполняется следующей структурой данных:
+00h: 4 байта - максимальный доступный блок в байтах
+04h: 4 байта - число доступных нефиксированных страниц
+08h: 4 байта - число доступных фиксированных страниц
+0Ch: 4 байта - линейный размер адресного пространства в страницах
+10h: 4 байта - общее число нефиксированных страниц
+14h: 4 байта - общее число свободных страниц
+18h: 4 байта - общее число физических страниц
+1Ch: 4 байта - свободное место в линейном адресном пространстве
+20h: 4 байта - размер swap-файла или раздела в страницах
+24h: 0Ch байт - все байты равны FFh

INT 31 h, AX = 0501 h: Выделить блок памяти
Вход: AX = 0501h
BX:CX = размер блока в байтах, больше нуля
Выход: если CF = 0,
BX:CX = линейный адрес блока;
SI:DI = идентификатор блока для функций 0502 и 0503

INT 31h, AX = 0502h: Освободить блок памяти
Вход: AX = 0502h
SI:DI = идентификатор блока
Выход: CF = 0, если не было ошибки

INT 31h, AX = 0503h: Изменить размер блока памяти
Вход: AX = 0503h
BX:CX = новый размер в байтах
SLDI = идентификатор блока
Выход: если CF = O,
BX:CX = новый линейный адрес блока;
SLDI = новый идентификатор

Нам потребуются еще две функции DPMI для работы с устройством, которое отображает свою память в физическое пространство адресов.

INT31h, AX = 0800h: Отобразить физический адрес выше границы 1 Мб на линейный
Вход: BX:CX = физический адрес
SI:DI = размер области в байтах
Выход: если CF = O, BX:CX = линейный адрес, который можно использовать для доступа к этой памяти

INT 31h,AX = 0801k. Отменить отображение, выполненное функцией 0800h
Вход: AX = 0801h
BX:CX = линейный адрес, возвращенный функцией 0800h
Выход: CF = 0, если не было ошибок

6.4.3. *Вывод на экран через линейный кадровый буфер*

```
; lfbfire.asm
; Программа, работающая с SVGA при помощи линейного кадрового буфера (LFB).
; Демонстрирует стандартный алгоритм генерации пламени.
; Требуется поддержка LFB видеоплатой (или загруженный эмулятор univbe),
; для компиляции необходим DOS-расширитель.
;
        .486p                           ; Для команды xadd.
        .model  flat                    ; Основная модель памяти
                                        ; в защищенном режиме.
        .code
        assume  fs:nothing              ; Нужно только для MASM.
_start:
        jmp     short   _main
                db      "WATCOM"        ; Нужно только для DOS/4GW.

; Начало программы.
; На входе обычно CS, DS и SS указывают на один и тот же сегмент с лимитом 4 Гб,
; ES указывает на сегмент с PSP на 100h байт, остальные регистры не определены.
```

```
_main:
        sti                             ; Даже флаг прерываний не определен,
        cld                             ; не говоря уже о флаге направления.

        mov     ax,0100h                ; Функция DPMI 0100h.
        mov     bx,100h                 ; Размер в 16-байтных параграфах.
        int     31h                     ; Выделить блок памяти ниже 1 Мб.
        jc      DPMI_error
        mov     fs,dx                   ; FS - селектор для выделенного блока.

; Получить блок общей информации о VBE 2.0 (см. раздел 4.5.2).
        mov     dword ptr fs:[0],'2EBV' ; Сигнатура VBE2 в начало блока.
        mov     word ptr v86_eax,4F00h  ; Функция VBE 00h.
        mov     word ptr v86_es,ax      ; Сегмент, выделенный DPMI.
        mov     ax,0300h                ; Функция DPMI 300h.
        mov     bx,0010h                ; Эмуляция прерывания 10h.
        xor     ecx,ecx
        mov     edi,offset v86_regs
        push    ds
        pop     es                      ; ES:EDI - структура v86_regs.
        int     31h                     ; Получить общую информацию VBE2.
        jc      DPMI_error
        cmp     byte ptr v86_eax,4Fh    ; Проверка поддержки VBE.
        jne     VBE_error

        movzx   ebp,word ptr fs:[18]    ; Объем SVGA-памяти в EBP
        shl     ebp,6                   ; в килобайтах.

; Получить информацию о видеорежиме 101h.
        mov     word ptr v86_eax,4F01h  ; Номер функции INT 10h.
        mov     word ptr v86_ecx,101h   ; Режим 101h - 640x480x256.
                                        ; ES:EDI - те же самые.
        mov     ax,0300h                ; Функция DPMI 300h - эмуляция
        mov     bx,0010h                ; прерывания INT 10h.
        xor     ecx,ecx
        int     31h                     ; Получить данные о режиме.
        jc      DPMI_error
        cmp     byte ptr v86_eax,4Fh
        jne     VBE_error
        test    byte ptr fs:[0],80h     ; Бит 7 байта атрибутов = 1 - LFB есть.
        jz      LFB_error

; Построение дескриптора сегмента, описывающего LFB.
; Лимит.
        mov     eax,ebp                 ; Видеопамять в килобайтах.
        shl     eax,10                  ; Теперь в байтах.
        dec     eax                     ; Лимит = размер - 1.
        shr     eax,12                  ; Лимит в 4-килобайтных единицах.
        mov     word ptr videodsc+0,ax  ; Записать биты 15-0 лимита
        shr     eax,8
        and     ah,0Fh
```

```
        or      byte ptr videodsc+6,ah    ; и биты 19-16 лимита.
        mov     eax,ebp                   ; Видеопамять в килобайтах.
        shl     eax,10                    ; В байтах.
        mov     edx,dword ptr fs:[40]     ; Физический адрес LFB.
        mov     cx,dx
        shld    ebx,edx,16                ; Поместить его в CX:DX,
        mov     di,ax
        shld    esi,eax,16                ; а размер - в SI:DI
        mov     ax,800h                   ; и вызвать функцию DPMI 0800h:.
        int     31h                       ; Отобразить физический адрес в линейный.
        jc      DPMI_error
        shrd    edx,ebx,16                ; Перенести полученный линейный адрес
        mov     dx,cx                     ; из BS:CX в EDX
        mov     word ptr videodsc+2,dx    ; и записать биты 15-0 базы,
        shr     edx,16
        mov     byte ptr videodsc+4,dl    ; биты  23-16
        mov     byte ptr videodsc+7,dh    ; и биты 31-24.
; Права.
        mov     bx,cs
        lar     cx,bx                     ; Прочитать права нашего сегмента
        and     cx,6000h                  ; и перенести биты DPL
        or      byte ptr videodsc+5,ch    ; в строящийся дескриптор.

; Поместить наш дескриптор в LDT и получить селектор.
        mov     cx,1                      ; Получить один новый дескриптор
        mov     ax,0                      ; у DPMI.
        int     31h
        jc      DPMI_error
        mov     word ptr videosel,ax      ; Записать его селектор.

        push    ds
        pop     es
        mov     edi,offset videodsc       ; ES:EDI - адрес нашего дескриптора.
        mov     bx,ax                     ; BX - выданный нам селектор.
        mov     ax,0Ch                    ; Функция DPMI 0Ch:.
        int     31h                       ; Загрузить дескриптор в LDT.
        jc      DPMI_error                ; Теперь в videosel лежит селектор на LFB.

; Переключение в режим 4101h (101h + LFB).
        mov     word ptr v86_eax,4F02h    ; 4F02h - установить SVGA-режим.
        mov     word ptr v86_ebx,4101h    ; Режим 4101h = 101h + LFB.
        mov     edi,offset v86_regs       ; ES:EDI - структура v86_regs.
        mov     ax,0300h                  ; Функция DPMI 300h.
        mov     bx,0010h                  ; Эмуляция прерывания 10h.
        xor     ecx,ecx
        int     31h

        mov     ax,word ptr videosel      ; AX - наш селектор.
enter_flame:
        ; Сюда придет управление с селектором в AX на A000h:0000,
        ; если произошла ошибка в любой VBE-функции.
        mov     es,ax                     ; ES - селектор видеопамяти или LFB.
```

```
; Отсюда начинается процедура генерации пламени.

; Генерация палитры для пламени.
        xor     edi,edi         ; Начать написание палитру с адреса ES:0000.
        xor     ecx,ecx
palette_gen:
        xor     eax,eax         ; Цвета начинаются с 0, 0, 0.
        mov     cl,63           ; Число значений для одного компонента.
palette_l1:
        stosb                   ; Записать байт,
        inc     eax             ; увеличить компонент,
        cmpsw                   ; пропустить два байта,
        loop    palette_l1      ; и так 64 раза.

        push    edi
        mov     cl,192
palette_l2:
        stosw                   ; Записать два байта
        inc     di              ; и пропустить один,
        loop    palette_12      ; и так 192 раза (до конца палитры).
        pop     edi             ; Восстановить EDI.
        inc     di
        jns     palette_gen

; Палитра сгенерирована, записать ее в регистры VGA DAC (см. раздел 5.10.4).
        mov     al,0            ; Начать с регистра 0.
        mov     dx,03C8h        ; Индексный регистр на запись.
        out     dx,al
        push    es
        push    ds              ; Поменять местами ES и DS.
        pop     es
        pop     ds
        xor     esi,esi
        mov     ecx,256*3       ; Писать все 256 регистров
        mov     edx,03C9h       ; в порт данных VGA DAC.
        rep     outsb
        push    es
        push    ds              ; Поменять местами ES и DS.
        pop     es
        pop     ds

; Основной цикл - анимация пламени, пока не будет нажата любая клавиша.
        xor     edx,edx                 ; Должен быть равен нулю.
        mov     ebp,7777h               ; Любое число.
        mov     ecx,dword ptr scr_width ; Ширина экрана.
main_loop:
        push    es                      ; Сохранить ES.
        push    ds
        pop     es              ; ES = DS - мы работаем только с буфером.
; Анимация пламени (классический алгоритм).
        inc     ecx
```

```
        mov     edi,offset buffer
        mov     ebx,dword ptr scr_height
        shr     ebx,1
        dec     ebx
        mov     esi,scr_width
animate_flame:
        mov     ax,[edi+esi*2-1]        ; Вычислить среднее значение цвета
        add     al,ah                   ; в данной точке (EDI) из значений
        setc    ah                      ; цвета в точке слева и на две строки
        mov     dl,[edi+esi*2+1]        ; вниз, справа и на две строки вниз
        add     ax,dx
        mov     dl,[edi+esi*4]          ; и на четыре строки вниз,
        add     ax,dx                   ; причем первое значение
        shr     ax,2                    ; модифицировать.
        jz      already_zero            ; Уменьшить яркость цвета.
        dec     ax
already_zero:
        stosb                           ; Записать новый цвет в буфер.
        add     eax,edx
        shr     eax,1
        mov     byte ptr [edi+esi-1],al
        loop    animate_flame
        mov     ecx,esi
        add     edi,ecx
        dec     ebx
        jnz     animate_flame

; Псевдослучайная полоска в низу экрана, которая служит генератором пламени.
generator_bar:
        xadd    bp,ax
        stosw
        stosw
        loop    generator_bar

        pop     es                      ; Восстановить ES для вывода на экран.
                                        ; Вывод пламени на экран.
        xor     edi,edi                 ; ES:EDI - LFB.
        push    esi
        add     esi,offset buffer       ; DS:ESI - буфер.
        mov     ecx,dword ptr scr_size  ; Размер буфера в двойных словах.
        rep     movsd                   ; Скопировать буфер на экран.
        pop     esi

        mov     ah,1                    ; Если не была нажата
        int     16h                     ; никакая клавиша,
        jz      main_loop               ; продолжить основной цикл.
        mov     ah,0                    ; Иначе -
        int     16h                     ; считать эту клавишу.

exit_all:
        mov     ax,03h                  ; Восстановить текстовый режим.
```

```
        int     10h
        mov     ax,4C00h                ; AH = 4Ch
        int     21h                     ; Выход из программы под расширителем DOS.

; Различные обработчики ошибок.
DPMI_error:                     ; Ошибка при выполнении одной из функций DPMI.
        mov     edx,offset DPMI_error_msg
        mov     ah,9
        int     21h                     ; Вывести сообщение об ошибке
        jmp     short  exit_all         ; и выйти.
VBE_error:                              ; Не поддерживается VBE.
        mov     edx,offset VBE_error_msg
        mov     ah,9
        int     21h                     ; Вывести сообщение об ошибке
        jmp     short start_with_vga    ; и использовать VGA.
LFB_error:                              ; Не поддерживается LFB.
        mov     edx,offset LFB_error_msg
        mov     ah,9                    ; Вывести сообщение об ошибке.
        int     21h
start_with_vga:
        mov     ah,0                    ; Подождать нажатия любой клавиши.
        int     16h
        mov     ax,13h                  ; Переключиться в видеорежим 13h,
        int     10h                     ; 320x200x256.
        mov     ax,2                    ; Функция DPMI 0002h:
        mov     bx,0A000h               ; построить дескриптор для реального
        int     31h                     ; сегмента.
        mov     dword ptr scr_width,320 ; Установить параметры режима
        mov     dword ptr scr_height,200
        mov     dword ptr scr_size,320*200/4
        jmp     enter_flame             ; и перейти к пламени.

        .data
; Различные сообщения об ошибках.
VBE_error_msg   db      "Ошибка VBE 2.0",0Dh,0Ah
                db      "Будет использоваться режим VGA 320x200",0Dh,0Ah,'$'
DPMI_error_msg  db      "Ошибка DPMI$"
LFB_error_msg   db      "LFB недоступен",0Dh,0Ah
                db      "Будет использоваться режим VGA 320x200",0Dh,0Ah,'$'
; Параметры видеорежима.
scr_width       dd      640
scr_height      dd      480
scr_size        dd      640*480/4
; Структура, используемая функцией DPMI 0300h.
v86_regs label byte
v86_edi         dd      0
v86_esi         dd      0
v86_ebp         dd      0
v86_res         dd      0
v86_ebx         dd      0
```

```
v86_edx         dd      0
v86_ecx         dd      0
v86_eax         dd      0
v86_flags       dw      0
v86_es          dw      0
v86_ds          dw      0
v86_fs          dw      0
v86_gs          dw      0
v86_ip          dw      0
v86_cs          dw      0
v86_sp          dw      0
v86_SS          dw      0
; Дескриптор сегмента, соответствующего LFB.
videodsc dw     0                       ; Биты 15-0 лимита.
                dw      0               ; Биты 15-0 базы.
                db      0               ; Биты 16-23 базы.
                db      10010010b       ; Доступ.
                db      10000000b       ; Биты 16-19 лимита и другие биты.
                db      0               ; Биты 24-31 базы.
; Селектор сегмента, описывающего LFB.
videosel dw     0
        .data?
; Буфер для экрана.
buffer          db      640*483 dup(?)
; стек
        .stack  1000h
        end     _start
```

Программирование с DOS-расширителями - один из лучших выходов для приложений, которые должны работать в любых условиях, включая старые версии DOS, и в то же время требуют 32-битный режим. Еще совсем недавно большинство компьютерных игр, в частности знаменитые Doom и Quake, выпускались именно как программы, использующие расширители DOS. На сегодняшний день в связи с повсеместным распространением операционных систем для PC, работающих в 32-битном защищенном режиме, требование работы в любых версиях DOS становится менее актуальным и все больше программ выходят только в версиях для Windows 95 или NT, чему и посвящена следующая глава.

Глава 7. Программирование для Windows 95/NT

Несмотря на то что Windows 95/NT кажутся более сложными операционными системам по сравнению с DOS, программировать для них на ассемблере намного проще. С одной стороны, Windows-приложение запускается в 32-битном режиме (мы не рассматриваем Windows 3.11 и более старые версии, которые работали в 16-битном режиме) с моделью памяти flat, так что программист получает все те преимущества, о которых говорилось в предыдущей главе, а с другой стороны - нам больше не нужно изучать в деталях, как программировать различные устройства компьютера на низком уровне. В настоящих операционных средах приложения пользуются только системными вызовами, число которых здесь превышает 2000 (около 2200 для Windows 95 и 2434 для Windows NT). Все Windows-приложения используют специальный формат исполняемых файлов - формат PE (Portable Executable). Такие файлы начинаются как обычные EXE-файлы старого образца (их также называют MZ по первым двум символам заголовка). Если такой файл запустить из DOS, он выполнится и выдаст сообщение об ошибке (текст сообщения зависит от используемого компилятора), в то время как Windows заметит, что после обычного MZ-заголовка файла идет PE-заголовок, и запустит приложение. Это будет означать лишь то, что для компиляции программ потребуются другие параметры в командной строке.

7.1. Первая программа

В качестве нашего первого примера посмотрим, насколько проще написать под Windows программу, которая загружает другую программу. В DOS (см. раздел 4.10) нам приходилось изменять распределение памяти, заполнять специальный блок данных EPB и только затем вызывать DOS. Здесь же не только вся процедура сокращается до одного вызова функции, а еще оказывается, что можно точно так же загружать программы, документы, графические и текстовые файлы и даже почтовые и Internet-адреса - все, для чего в реестре Windows записано действие, выполняющееся при попытке открытия.

```
; winurl.asm
; Пример программы для Win32.
; Запускает установленный по умолчанию браузер на адрес, указанный в строке URL.
; Аналогично можно запускать любую программу, документ и какой угодно файл,
; для которого определена операция open.
;
include shell32.inc
include kernel32.inc
```

```
        .386
        .model  flat
        .const
URL      db     "http://www.lionking.org/~cubbi/",0
        .code
_start:         ; Метка точки входа должна начинаться с подчеркивания.
        xor     ebx,ebx
        push    ebx             ; Для исполняемых файлов - способ показа.
        push    ebx             ; Рабочая директория.
        push    ebx             ; Командная строка.
        push    offset URL      ; Имя файла с путем
        push    ebx             ; Операция open или print (если NULL - open).
        push    ebx             ; Идентификатор окна, которое получит сообщения.
        call    ShellExecute    ; ShellExecute (NULL,NULL,url,NULL,NULL,NULL)
        push    ebx             ; Код выхода.
        call    ExitProcess     ; ExitProcess(0)
end     _start
```

Итак, *в* этой программе выполняется вызов двух системных функций Win32 - ShellExecute (открыть файл) и ExitProcess (завершить процесс). Чтобы активизировать системную функцию Windows, программа должна поместить в стек все параметры от последнего к первому и передать управление дальней CALL. Данные функции сами освобождают стек (завершаясь RET N) и возвращают результат работы в регистре EAX. Такая договоренность о передаче параметров называется STDCALL. С одной стороны, это позволяет вызывать функции с нефиксированным числом параметров, а с другой - вызывающая сторона не должна заботиться об освобождении стека. Кроме того, функции Windows сохраняют значение регистров EBP, ESI, EDI и EBX, этим мы пользовались в нашем примере - хранили 0 в регистре EBX и применяли 1-байтную команду PUSH EBX вместо 2-байтной PUSH 0.

Прежде чем мы сможем скомпилировать winurl.asm, нужно создать файлы kernel32.inc и shell32.inc, куда поместим директивы, описывающие вызываемые системные функции.

```
; kernel32.inc
; Включаемый файл с определениями функций из kernel32.dll.
ifdef   _TASM_
        includelib      import32.lib
                ; Имена используемых функций.
                extrn   ExitProcess:near
else
        includelib      kernel32.lib
                ; Истинные имена используемых функций.
                extrn   __imp__ExitProcess@4:dword
                ; Присваивания для облегчения читаемости кода.
        ExitProcess     equ     __imp__ExitProcess@4
endif

; shell32.inc
; Включаемый файл с определениями функций из shell32.dll.
```

```
ifdef   _TASM_
        includelib      import32.lib
                ; Имена используемых функций.
                extrn   ShellExecuteA:near
                ; Присваивания для облегчения читаемости кода.
        ShellExecute    equ     ShellExecuteA
else
        includelib      shell32.lib
                ; Истинные имена используемых функций.
                extrn   __imp__ShellExecuteA@24:dword
                ; Присваивания для облегчения читаемости кода.
        ShellExecute    equ     __imp__ShellExecuteA@24
endif
```

Имена всех системных функций Win32 модифицируются так, что перед именем функции ставится подчеркивание, а после - знак @ и число байтов, которое занимают параметры, передаваемые ей в стеке: так ExitProcess превращается в _ExitProcess@4. Компиляторы с языков высокого уровня часто останавливаются на этом и вызывают функции по имени _ExitProcess@4, но реально появляется небольшая процедура-заглушка, которая ничего не делает, а лишь передает управление на такую же метку, но с добавленным __imp_ - __imp__ExitProcess@4. Во всех наших примерах мы будем обращаться напрямую к __imp__ExitProcess@4. К сожалению, TASM (а точнее TLINK32) использует собственный способ вызова системных функций, который нельзя обойти подобным образом, и программы, скомпилированные с его помощью, оказываются немного больше и в некоторых случаях работают медленнее. Мы отделили описания функций для TASM во включаемых файлах при помощи директив условного ассемблирования, которые будут использовать их, если в командной строке ассемблера указать /D_TASM_.

Кроме того, все функции, работающие со строками (как, например, ShellExecute), существуют в двух вариантах. Если строка рассматривается в обычном смысле, как набор символов ASCII, к имени функции добавляется A (ShellExecuteA). Другой вариант функции, использующий строки в формате UNICODE (два байта на символ), заканчивается буквой U. Во всех наших примерах будут использоваться обычные ASCII-функции, но, если вам потребуется перекомпилировать программы на UNICODE, достаточно поменять A на U во включаемых файлах.

Итак, теперь, когда у нас есть все необходимые файлы, можно скомпилировать первую программу для Windows.

Компиляция MASM:

```
ml /c /coff /Cp winurl.asm
link winurl.obj /subsystem:windows
```

(здесь и далее используется 32-битная версия link.exe)

Компиляция TASM:

```
tasm /m /ml /D_TASM_ winurl.asm
tlink32 /Tpe /aa /c /x winurl.obj
```

Компиляция WASM:

```
wasm   winurl.asm
wlink file winurl.obj form windows nt op c
```

Также для компиляции потребуются файлы kernel32.lib и shell32.lib в первом и третьем случае и import32.1ib - во втором. Эти файлы входят в дистрибутивы любых средств разработки для Win32 от соответствующих компаний - Microsoft, Watcom (Sybase) и Borland (Inprise), хотя их всегда можно воссоздать из файлов kernel32.dll и shell32.dll, находящихся в директории WINDOWS\SYSTEM.

Иногда вместе с дистрибутивами различных средств разработки для Windows поставляется файл windows.inc, в котором дано макроопределение Invoke или заменена макросом команда call так, что при вызове можно передать список аргументов, первым из которых будет имя вызываемой функции, а затем через запятую - все параметры. С использованием данных макроопределений наша программа выглядела бы так:

```
_start:
        xor     ebx,ebx
        Invoke  ShellExecute, ebx, ebx, offset URL, ebx,\  .
                              ebx, ebx
        Invoke  ExitProcess,  ebx
end     _start
```

И этот текст компилируется в точно такой же код, что и у нас, но выполняется вызов промежуточной функции _ExitProcess@4, а не функции _ _imp_ _ExitProcess@4. Использование данной формы записи не позволяет применять отдельные эффективные приемы оптимизации, которые мы будем приводить в наших примерах, - помещение параметров в стек заранее и вызов функции командой JMP. И наконец, файла windows.inc у вас может просто не оказаться, так что будем писать push перед каждым параметром вручную.

7.2. Консольные приложения

Исполнимые программы для Windows делятся на два основных типа - консольные и графические приложения. При запуске консольного приложения открывается текстовое окно, с которым программа может общаться функциями WriteConsole/ReadConsole и другими (соответственно при запуске из другого консольного приложения, например файлового менеджера FAR, программе отводится текущая консоль и управление не возвращается к FAR, пока программа не закончится). Графические приложения соответственно не получают консоли и должны открывать окна, чтобы вывести что-нибудь на экран.

Для компиляции консольных приложений мы будем пользоваться следующими командами.

MASM:

```
ml /c /coff /Cp winurl.asm
link winurl.asm /subsystem:console
```

TASM:

```
tasm /m /ml /D_TASM_ winurl.asm
tlink32 /Tpe /ap /c /x winurl.obj
```

WASM:

```
wasm winurl.asm
wlink file winurl.obj form windows nt runtime console op c
```

Попробуйте скомпилировать программу winurl.asm указанным способом, чтобы увидеть, как отличается работа консольного приложения от графического.

В качестве примера полноценного консольного приложения напишем программу, которая перечислит все подключенные сетевые ресурсы (диски и принтеры), используя системные функции WNetOpenEnum, WNetEnumResource и WNetCloseEnum.

```
; netenum.asm
; Консольное приложение для Win32, перечисляющее сетевые ресурсы.
include def32.inc
include kernel32.inc
include mpr.inc

        .386
        .model flat
        .const
greet_message   db      'Example Win32 console program',0Dh,0Ah,0Dh,0Ah,0
error1_message  db      0Dh,0Ah,'Could not get current user name',0Dh,0Ah,0
error2_message  db      0Dh,0Ah,'Could not enumerate',0Dh,0Ah,0
good_exit_msg   db      0Dh,0Ah,0Dh,0Ah,'Normal termination',0Dh,0Ah,0
enum_msg1       db      0Dh,0Ah,'Local ',0
enum_msg2       db      ' remote - ',0
        .data
user_name       db      "List of connected resources for user"
user_buff       db      64 dup (?)      ; Буфер для WNetGetUser.
user_buff_l     dd      $-user_buff     ; Размер буфера для WNetGetUser.
enum_buf_l      dd      1056            ; Длина enum_buf в байтах.
enum_entries    dd      1               ; Число ресурсов, которые в нем помещаются.
        .data?
enum_buf        NTRESOURCE <?,?,?,?,?,?,?,?>    ; Буфер для WNetEnumResource.
                dd      256 dup (?)     ; 1024 байта для строк.
message_l       dd      ?               ; Переменная для WriteConsole.
enum_handle     dd      ?               ; Идентификатор для WNetEnumResource.
        .code
_start:
; Получить от системы идентификатор буфера вывода stdout.
        push    STD_OUTPUT_HANDLE
        call    GetStdHandle            ; Возвращает идентификатор STDOUT в eax,
        mov     ebx,eax                 ; а мы будем хранить его в EBX.
; Вывести строку greetjnessage на экран.
        mov     esi,offset greet:_message
        call    output_string
```

```
; Определить имя пользователя, которому принадлежит наш процесс.
        mov     esi,offset user_buff
        push    offset user_buff_l      ; Адрес переменной с длиной буфера.
        push    esi                     ; Адрес буфера.
        push    0                       ; NULL
        call    WNetGetUser
        cmp     eax,NO_ERROR            ; Если произошла ошибка,
        jne     error_exit1             ; выйти,
        mov     esi,offset user_name    ; иначе - вывести строку на экран.
        call    output_string

; Начать перечисление сетевых ресурсов.
        push    offset enum_handle      ; Идентификатор для WNetEnumResource.
        push    0
        push    RESOURCEUSAGE_CONNECTABLE ; Все присоединяемые ресурсы.
        push    RESOURCETYPE_ANY        ; Ресурсы любого типа.
        push    RESOURCE_CONNECTED      ; Только присоединенные сейчас.
        call    WNetOpenEnum            ; Начать перечисление.
        cmp     eax,NO_ERROR            ; Если произошла ошибка,
        jne     error_exit2             ; выйти.
; Цикл перечисления ресурсов.
enumeration_loop:
        push    offset enum_buf_l       ; Длина буфера в байтах.
        push    offset enum_buf         ; Адрес буфера.
        push    offset enum_entries     ; Число ресурсов.
        push    dword ptr enum_handle   ; Идентификатор от WNetOpenEnum.
        call    WNetEnumResource
        cmp     eax,ERROR_NO_MORE_ITEMS ; Если они закончились,
        je      end_enumeration         ; завершить перечисление,
        cmp     eax,NO_ERROR            ; если произошла ошибка,
        jne     error_exit2             ; выйти с сообщением об ошибке.
; Вывод информации о ресурсе на экран.
        mov     esi,offset enum_msg1    ; Первая часть строки -
        call    output_string           ; на консоль.
        mov     esi,dword ptr enum_buf.lpLocalName ; Локальное имя устройства -
        call    output_string           ; на консоль.
        mov     esi,offset enum_msg2    ; Вторая часть строки -
        call    output_string           ; на консоль.
        mov     esi,dword ptr enum_buf.lpRemoteName ; Удаленное имя устройства -
        call    output_string           ; туда же.

        jmp     short enumeration_loop  ; Продолжить перечисление.
; Конец цикла.
end_enumeration:
        push    dword ptr, enum_handle
        call    WNetCloseEnum           ; Конец перечисления.

        mov     esi,offset good_exit_msg
exit_program:
        call    output_string           ; Вывести строку.
        push    0                       ; Код выхода.
```

```
        call    ExitProcess             ; Конец программы.
; Выходы после ошибок.
error_exit1:
        mov     esi,offset error1_message
        jmp     short exit_program
error_exit2:
        mov     esi,offset error2_message
        jmp     short exit_program

; Процедрура output_string.
; Выводит на экран строку.
; Вход: esi - адрес строки.
;       ebx - идентификатор stdout или другого консольного буфера.
output_string proc near
; Определить длину строки.
        cld
        xor     eax,eax
        mov     edi,esi
        repne   scasb
        dec     edi
        sub     edi,esi
; Послать ее на консоль.
        push    0
        push    offset message_l        ; Сколько байтов выведено на консоль.
        push    edi                     ; Сколько байтов надо вывести на консоль.
        push    esi                     ; Адрес строки для вывода на консоль.
        push    ebx                     ; Идентификатор буфера вывода.
        call .  WriteConsole            ; WriteConsole (hConsoleOutput,lpvBuffer,
                                        ; cchToWrite,lpcchWritten,lpvReserved)
        ret
output_string endp

end     _start
```

В файл kernel32.inc надо добавить между ifdef _TASM_ и else строки:

```
                extrn   GetStdHandle:near
                extrn   WriteConsoleA:near
WriteConsole    equ     WriteConsoleA
```

и между- else и endif:

```
                extrn   __imp__GetStdHandle@4
                extrn   __imp__WriteConsoleA@20
GetStdHandle    equ     __imp__GetStdHandle@4
WriteConsole    equ     __imp__WriteConsoleA@20
```

Кроме того, надо создать файл mpr.inc:

```
; mpr.inc
; Включаемый файл с определениями функций из mpr.dll.
ifdef _TASM_
        includelib      import32.lib
; Имена используемых функций.
                extrn   WNetGetUserA:near
                extrn   WNetOpenEnumA:near
```

```
                extrn   WNetEnumResourceA:near
                extrn   WNetCloseEnum:near
; Присваивания для облегчения читаемости кода.
WNetGetUser     equ     WNetGetUserA
WNetOpenEnum    equ     WNetOpenEnumA
WNetEnumResource equ    WNetEnumResourceA
else
        includelib mpr.lib
; Истинные имена используемых функций.
                extrn   __imp__WNetGetUserA@12:dword
                extrn   __imp__WNetOpenEnumA@20:dword
                extrn   __imp__WNetEnumResourceA@16:dword
                extrn   __imp__WNetCloseEnum@4:dword
; Присваивания для облегчения читаемости кода.
WNetGetUser     equ     __imp__WNetGetUserA@12
WNetOpenEnum    equ     __imp__WNetOpenEnumA@20
WNetEnumResource equ    __imp__WNetEnumResourceA@16
WNetCloseEnum   equ     __imp__WNetCloseEnum@4
endif
```

Еще потребуется файл def32.inc, куда мы поместим определения констант и структур из разных включаемых файлов для языка C. Существует утилита h2inc, преобразующая эти файлы целиком, но нас интересует собственный включаемый файл, в который будем добавлять новые определения по мере надобности.

```
; def32.inc
; Файл с определениями констант и типов для примеров программ под
; Win32.

; Из winbase.h.
STD_OUTPUT_HANDLE       equ     -11

; Из winerror.h.
NO_ERROR                equ     0
ERROR_NO_MORE_ITEMS     equ     259

; Из winnetwk.h.
RESOURCEUSAGE_CONNECTABLE equ   1
RESOURCETYPE_ANY        equ     0
RESOURCE_CONNECTED      equ     1
NTRESOURCE  struc
        dwScope         dd      ?
        dwType          dd      ?
        dwDisplayType   dd      ?
        dwUsage         dd      ?
        lpLocalName     dd      ?
        lpRemoteName    dd      ?
        lpComment       dd      ?
        lpProvider      dd      ?
NTRESOURCE  ends
```

Этот пример, разумеется, можно было построить эффективнее, выделив большой буфер для WNetEnumResource, например при помощи LocalAlloc или

GlobalAlloc (в Win32 это одно и то же), а потом, прочитав информацию обо всех ресурсах, хранящихся в нем, пришлось бы следить за тем, кончились они или нет, и вызывать WNetEnumResource еще раз.

7.3. Графические приложения

7.3.1 *Окно типа MessageBox*

Для того чтобы вывести на экран любое окно, программа обычно должна сначала описать его внешний вид и все свойства, то есть то, что называется классом окна. О том, как это сделать, - немного позже, а для начала выведем одно из окон с предопределенным классом - окно типа MessageBox. MessageBox - это маленькое окно с указанным текстовым сообщением и одной или несколькими кнопками. В нашем примере сообщением будет традиционное **Hello world!**

```
; winhello.asm
; Графическое win32-приложение.
; Выводит окно типа messagebox с текстом "Hello world!".

include def32.1nc
include kernel32.inc
include user32.inc

        .386
        .model flat
        .const
; Заголовок окна.
hello_title       db "First Win32 GUI program", 0
; Сообщение.
hello_message     db "Hello world!",0
        .code
_start:
        push    MB_ICONINFORMATION        ; Силь окна.
        push    offset hello_title        ; Адрес строки с заголовком.
        push    offset  hello_message     ; Адрес строки с сообщением.
        push    0                         ; Идентификатор предка.
        call    MessageBox

        push    0                         ; Код выхода.
        call    ExitProcess               ; Завершение программы.
end     _start
```

Естественно, нам потребуются новые добавления к включаемым файлам: добавить к файлу def32.inc строку

```
; Из winuser.h.
MB_ICONINFORMATION        equ     40h
```

и создать новый файл, user32.inc, куда будут входить определения функций из user32.dll - библиотеки, где расположены все основные функции, отвечающие за оконный интерфейс:

```
; user32.inc
; Включаемый файл с определениями функций из user32.dll.
;
ifdef _TASM_
                includelib      import32.lib
; Имена используемых функций.
                extrn   MessageBoxA:near
; Присваивания.
                MessageBox      equ     MessageBoxA
else
                includelib      user32.lib
; Истинные имена используемых функций.
                        extrn   _imp__MessageBoxA@16:dword
; Присваивания для облегчения читаемости кода.
                MessageBox      equ     _imp__MessageBoxA@16
endif
```

Теперь можно скомпилировать эту программу аналогично тому, как мы компилировали winurl.asm, и запустить — на экране появится маленькое окно с нашим сообщением, которое пропадет после нажатия кнопки OK. Если скомпилировать winhello.asm как консольное приложение, ничего не изменится, но текстовое окно с именем программы будет открыто до тех пор, пока не закроется окно с нашим сообщением.

7.3.2. Окна

Теперь, когда мы знаем, как просто выводится окно с предопределенным классом, возьмемся за создание собственного окна - процедуры, на которой будут базироваться все последующие примеры, и познакомимся с понятием сообщения. В DOS основным средством передачи управления программам в различных ситуациях служат прерывания. В Windows прерывания используются для системных нужд, а для приложений существует аналогичный механизм - механизм событий. Так, нажатие клавиши на клавиатуре, если эта клавиша не используется Windows, генерирует сообщение WM_KEYDOWN или WM_KEYUP, которое можно перехватить, добавив в цепь обработчиков события собственное при помощи SetWindowHookEx. Затем события преобразуются в сообщения, которые рассылаются функциям - обработчикам сообщений - и которые можно прочитать из основной программы при помощи вызовов GetMessage и PeekMessage.

Для начала нам потребуется только обработка сообщения закрытия окна (WM_DESTROY и WM_QUIT), по которому программа будет завершаться.

```
; window.asm
; Графическое win32-приложение, демонстрирующее базовый вывод окна.
;
include def32.inc
include kernel32.inc
include user32.inc
        .386
        .model flat
        .data
```

```
class_name      db      "window class 1",0
window_name     db      "Win32 assembly example",0
; Структура, описывающая класс окна.
wc      WNDCLASSEX <4*12,CS_HREDRAW or CS_VREDRAW,offset win_proc, 0, 0, ?, ?, ?, \
                    COLOR_WINDOW+1,0,offset class_name,0>
        ; Здесь находятся следующие поля:
        ; wc.cbSize = 4*12 - размер этой структуры
        ; wc.style - стиль окна (перерисовывать при изменении размера)
        ; wc.lpfnWndProc - обработчик событий окна (win_proc)
        ; wc.cbClsExtra - число дополнительных байтов после структуры (0)
        ; wc.cbWndExtra - число дополнительных байтов после окна (0)
        ; wc.hInstance - идентификатор нашего процесса (?)
        ; wc.hIcon - идентификатор иконки (?)
        ; wc.hCursor - идентификатор курсора (?)
        ; wc.hbrBackground - идентификатор кисти или цвет фона + 1 (COLOR_WINDOW+1)
        ; wc. lpszMenuName - ресурс с основным меню (в этом примере - 0)
        ; wc.lpszClassName - имя класса (строка class_name)
        ; wc.hIconSm - идентификатор маленькой иконки (только в Windows 95,
        ; для NT должен быть 0).

        .data?
msg_    MSG     <?,?,?,?,?,?>    ; А это - структура, в которую возвращается
                                  ; сообщение после GetMessage.
        .code
_start:
        xor     ebx,ebx           ; В EBX будет 0 для команд push 0.
; Определить идентификатор нашей программы
        push    ebx
        call    GetModuleHandle
        mov     esi,eax           ; и сохранить его в ESI.
; Заполнить и зарегистрировать класс.
        mov     dword ptr wc.hInstance,eax ; Идентификатор предка.
; Выбрать иконку.
        push    IDI_APPLICATION   ; Стандартная иконка приложения.
        push    ebx               ; Идентификатор модуля с иконкой.
        call    LoadIcon
        mov     wc.hIcon,eax      ; Идентификатор иконки для нашего класса.
; Выбрать форму курсора.
        push    IDC_ARROW         ; Стандартная стрелка.
        push    ebx               ; Идентификатор модуля с курсором.
        call    LoadCursor
        mov     wc.hCursor,eax    ; Идентификатор курсора для нашего класса.
        push    offset wc
        call    RegisterClassEx   ; Зарегистрировать класс.
; Создать окно.
        mov     ecx,CW_USEDEFAULT; push ecx короче push N в пять раз.
        push    ebx               ; Адрес структуры CREATESTRUCT (здесь NULL).
        push    esi               ; Идентификатор процесса, который будет получать.
                                  ; сообщения от окна (то есть наш).
        push    ebx               ; Идентификатор меню или окна-потомка.
```

```
        push    ebx             ; Идентификатор окна-предка.
        push    ecx             ; Высота (CW_USEDEFAULT - по умолчанию).
        push    ecx             ; Ширина (по умолчанию).
        push    ecx             ; Y-координата (по умолчанию).
        push    ecx             ; X-координата (по умолчанию).
        push    WS_OVERLAPPEDWINDOW     ; Стиль окна.
        push    offset window_name      ; Заголовок окна.
        push    offset class_name; Любой зарегистрированный класс.
        push    ebx             ; Дополнительный стиль.
        call    CreateWindowEx  ; Создать окно (eax - идентификатор окна).
        push    eax             ; Идентификатор для UpdateWindow.
        push    SW_SHOWNORMAL   ; Тип показа для для ShowWindow.
        push    eax             ; Идентификатор для ShowWindow.
; Больше идентификатор окна нам не потребуется.
        call    ShowWindow      ; Показать окно
        call    UpdateWindow    ; и послать ему сообщение WM_PAINT.

; Основной цикл - проверка сообщений от окна и выход по WM_QUIT.
        mov     edi,offset msg_ ; push edi короче push N в 5 раз.
message_loop:
        push    ebx             ; Последнее сообщение.
        push    ebx             ; Первое сообщение.
        push    ebx             ; Идентификатор окна (0 - любое наше окно).
        push    edi             ; Адрес структуры MSG.
        call    GetMessage      ; Получить сообщение от окна с ожиданием -
                                ; не забывайте использовать PeekMessage,
                                ; если нужно в этом цикле что-то выполнять.
        test    eax,eax         ; Если получено WM_QUIT,
        jz      exit_msg_loop   ; выйти.
        push    edi             ; Иначе - преобразовать сообщения типа
        call    TranslateMessage ; WM_KEYUP в сообщения типа WM_CHAR
        push    edi
        call    DispatchMessage ; и послать их процедуре окна (иначе его просто
                                ; нельзя будет закрыть).
        jmp     short message_loop      ; Продолжить цикл.
exit_msg_loop:
; Выход из программы.
        push    ebx
        call    ExitProcess

; Процедура win_proc.
; Вызывается окном каждый раз, когда оно получает какое-нибудь сообщение.
; Именно здесь будет происходить вся работа программы.

; Процедура не должна изменять регистры EBP,EDI,ESI и EBX!.

win_proc        proc
; Так как мы получаем параметры в стеке, построить стековый кадр.
        push    ebp
        mov     ebp,esp
; Процедура типа WindowProc вызывается со следующими параметрами:
```

```
        wp_hWnd    equ dword ptr [ebp+08h]; идентификатор окна,
        wp_uMsg    equ dword ptr [ebp+0Ch]; номер сообщения,
        wp_wParam  equ dword ptr [ebp+10h]; первый параметр,
        wp_lParam  equ dword ptr [ebp+14h]; второй параметр.
; Если мы получили сообщение WM_DESTROY (оно означает, что окно уже удалили
; с экрана, нажав Alt-F4 или кнопку в верхнем правом углу),
; то отправим основной программе сообщение WM_QUIT.
        cmp     wp_uMsg,WM_DESTROY
        jne     not_wm_destroy
        push    0                       ; Код выхода.
        call    PostQuitMessage         ; Послать WM_QUIT
        jmp     short end wm_check      ; и выйти из процедуры.
not_wm_destroy:
; Если мы получили другое сообщение - вызвать его обработчик по умолчанию.
        leave                           ; Восстановить ebp
        jmp     DefWindowProc           ; и вызвать DefWindowProc с нашими параметрами
                                        ; и адресом возврата в стеке.
end_wm_check:
        leave                           ; Восстановить ebp
        ret     16                      ; и вернуться самим, очистив стек от параметров.
win_proc endp
        end     _start
```

Необходимые добавления в файл def32.inc:

```
; Из winuser. h.
IDI_APPLICATION         equ     32512
WM_DESTROY.             equ     2
CS_HREDRAW              equ     2
CS_VREDRAW              equ     1
CW_USEDEFAULT           equ     80000000h
WS_OVERLAPPEDWINDOW     equ     0CF0000h
IDC_ARROW               equ     32512
SW_SHOWNORMAL           equ     1
COLOR_WINDOW            equ     5
WNDCLASSEX              struc
        cbSize          dd      ?
        • style         dd      ?
        lpfnWndProc     dd      ?
        cbClsExtra      dd      ?
        cbWndExtra      dd      ?
        hInstance       dd      ?
        hIcon           dd      ?
        hCursor         dd      ?
        hbrBackground   dd      ?
        lpszMenuName    dd      ?
        lpszClassName   dd      ?
        hIconSm         dd      ?
WNDCLASSEX              ends
MSG                     struc
      . hwnd            dd      ?
```

```
        message         dd      ?
        wParam          dd      ?
        lParam          dd      ?
        time            dd      ?
        Pt              dd      ?
MSG                     ends
```

Добавления в файл user32.inc:
между ifdef _TASM_ и else:

```
                        extrn   DispatchMessageA:near
                        extrn   TranslateMessage:near
                        extrn   GetMessageA:near
                        extrn   LoadIconA:near
                        extrn   UpdateWindow:near
                        extrn   ShowWindow:near
                        extrn   CreateWindowExA:near
                        extrn   DefWindowProcA:near
                        extrn   PostQuitMessage:near
                        extrn   RegisterClassExA:near
                        extrn   LoadCursorA:near
        DispatchMessage equ     DispatchMessageA
        GetMessage      equ     GetMessageA
        LoadIcon        equ     LoadIconA
        CreateWindowEx  equ     CreateWindowExA
        DefWindowProc   equ     DefWindowProcA
        RegisterClassEx equ     RegisterClassExA
        LoadCursor      equ     LoadCursorA
```

и между else и endif:

```
                        extrn_____imp__DispatchMessageA@4:dword
                        extrn_____imp__TranslateMessage@4:dword
                        extrn_____imp__GetMessageA@16:dword
                        extrn_____imp__LoadIconA@8:dword
                        extrn_____imp__UpdateWindow@4:dword
                        extrn_____imp__ShowWindow@8:dword
                        extrn_____imp__CreateWindowExA@48:dword
                        extrn_____imp__DefWindowProcA@16:dword
                        extrn_____imp__PostQuitMessage@4:dword
                        extrn_____imp__RegisterClassExA@4:dword
                        extrn_____imp__LoadCursorA@8:dword
        DispatchMessage equ_______imp__DispatchMessageA@4
        TranslateMessage equ______imp__TranslateMessage@4
        GetMessage      equ_______imp__GetMessageA@16
        Loadlconequ     __imp__LoadIconA@8
        UpdateWindow    equ_______imp__UpdateWindow@4
        ShowWindow      equ_______imp__ShowWindow@8
        CreateWindowEx  equ_______imp__CreateWindowExA@48
        DefWindowProc   equ_______imp__DefWindowProcA@16
        PostQuitMessage equ_______imp__PostQuitMessage@4
        RegisterClassEx equ_______imp__RegisterClassExA@4
        LoadCursor      equ_______imp__LoadCursorA@8
```

и в файл kernel32.inc между ifdef _TASM_ и else:

```
                    extrn   GetModuleHandleA:near
GetModuleHandle     equ     GetModuleHandleA
```

и между else и endif:

```
                    extrn   __imp__GetModuleHandleA@4:dword
GetModuleHandle     equ     __imp__GetModuleHandleA@4
```

В начале раздела говорилось, что программировать под Windows просто, а в то же время текст обычной программы вывода пустого окна на экран уже занимает больше места, чем, например, текст программы проигрывания wav-файла из раздела 5.10.8. Где же обещанная простота? Так вот, оказывается, что, написав window.asm, мы уже создали значительную часть всех последующих программ, а когда мы дополним этот текст полноценным диалогом, обнаружится, что больше не нужно писать все эти громоздкие конструкции, достаточно лишь копировать отдельные участки текста.

7.5.5. *Меню*

Меню - один из краеугольных камней идеологии Windows. Похожие друг на друга меню разрешают пользоваться абсолютно незнакомыми программами, не читая инструкций, и узнавать об их возможностях путем просмотра содержания различных пунктов меню. Попробуем добавить меню и в нашу программу window.asm.

Первое, что нам нужно получить, - это само меню. Его, так же как и иконки, диалоги и другая информация (вплоть до версии программы), записывают в файлы ресурсов. Файл ресурсов имеет расширение *.RC для текстового файла или *.RES для бинарного файла, созданного специальным компилятором ресурсов (RC, BRCC32 или WRC). И те, и другие файлы ресурсов можно редактировать особыми программами, входящими в дистрибутивы C/C++, или с помощью других средств разработки для Windows, но мы не будем создавать слишком сложное меню и напишем RC-файл вручную, например так:

```
// winmenu.rc
// Файл ресурсов для программы winmenu.asm.

#define ZZZ_TEST 1
#define ZZZ_OPEN  2
#define ZZZ_SAVE  3
#define ZZZ_EXIT  4

ZZZ_Menu MENU {
        POPUP "&File" {
                MENUITEM "&Open",ZZZ_OPEN
                MENUITEM. "&Save", ZZZ_SAVE
                MENUITEM  SEPARATOR
                MENUITEM  "E&xit",ZZZ_EXIT
        }
        MENUITEM "&Edit",ZZZ_TEST
}
```

Чтобы добавить этот файл в программу, его надо скомпилировать и указать имя *.RES-файла для компоновщика:

MASM:

```
ml /c /coff /Cp winmenu.asm
rc /r winmenu.rc
link winmenu.obj winmenu.res /subsystem:windows
```

TASM:

```
tasm /x /m /ml /D_TASM_ winmenu.asm
brcc32 winmenu.res
tlink32 /Tpe /aa /c winmenu.obj,,,, ,winmenu.res
```

WASM:

```
wasm winmenu.asm
wrc /r /bt=nt winmenu.rc
wlink file winmenu.obj res winmenu.res form windows nt op c
```

А теперь сам текст программы. Чтобы показать, как мало требуется внести изменений в программу window.asm, комментарии для всех строк, перенесенных оттуда без поправок, заменены символом *.

```
; winmenu.asm
; Графическое win32-приложение, демонстрирующее работу с меню.
; Звездочками отмечены строки, скопированные из файла window.asm.
;
ZZZ_TEST        equ     0       ; Сообщения от нашего меню
ZZZ_OPEN        equ     1       ; должны совпадать с определениями из winmenu.rc.
ZZZ_SAVE        equ     2       ; Кроме того, в нашем примере их номера важны,
ZZZ_EXIT        equ     3       ; потому что они используются как индексы для
                                ; таблицы переходов к обработчикам.

include def32.inc               ;*
include kernel32.inc            ;*
include user32.inc              ;*
        .386                    ;*
        .model  flat            ;*
        .data                   ;*
class_name      db      "window class 1",0      ;*
window_name     db      "Win32 assembly example",0 ;*
menu_name       db      "ZZZ_Menu",0            ; Имя меню в файле ресурсов.
test_msg        db      "You selected menu item TEST",0  ; Строки для
open_msg        db      "You selected menu item OPEN",0  ; демонстрации работы
save_msg        db      "You selected menu item SAVE",0  ; меню.
wc      WNDCLASSEX <4*12,CS_HREDRAW or CS_VREDRAW,offset win_proc,0,0,?,?,?,\
                COLOR_WINDOW+1,0,offset class_name,0> ;*
        .data?                          ;*
msg_    MSG     <?,?,?,?,?,?>          ;*
        .code                           ;*
_start:                                 ;*
        xor     ebx,ebx                 ;*
        push    ebx                     ;*
        call    GetModuleHandle         ;*
        mov     esi,eax                 ;*
        mov     dword ptr wc.hInstance,eax ;*
```

```
        push    IDI_APPLICATION         ;*
        push    ebx                     ;*
        call    LoadIcon                ;*
        mov     wc.hIcon,eax            ;*
        push    IDC_ARROW               ;*
        push    ebx                     ;*
        call    LoadCursor              ;*
        mov     wc.hCursor,eax          ;*
        push    offset wc               ;*
        call    RegisterClassEx         ;*

        push    offset menu_name        ; Имя меню.
        push    esi                     ; Наш идентификатор.
        call    LoadMenu                ; Загрузить меню из ресурсов.
        mov     ecx,CW_USEDEFAULT       ;*
        push    ebx                     ;*
        push    esi                     ;*
        push    eax                     ; Идентификатор меню или окна-потомка.
        push    ebx                     ;*
        push    ecx                     ;*
        push    ecx                     ;*
        push    ecx                     ;*
        push    ecx                     ;*
        push    WS_OVERLAPPEDWINDOW     ;*
        push    offset window_name      ;*
        push    offset class_name       ;*
        push    ebx                     ;*
        call    CreateWindowEx          ;*
        push    eax                     ;*
        push    SW_SHOWNORMAL           ;*
        push    eax                     ;*
        call    ShowWindow              ;*
        call    UpdateWindow            ;*
        mov     edi,offset msg_         ;*
message_loop:                           ;*
        push    ebx                     ;*
        push    ebx                     ;*
        push    ebx                     ;*
        push    edi                     ;*
        call    GetMessage              ;*
        test    eax,eax                 ;*
        jz      exit_msg_loop           ;*
        push    edi                     ;*
        call    TranslateMessage        ;*
        push    edi                     ;*
        call    DispatchMessage         ;*
        jmp     short message_loop      ;*
exit_msg_loop:                          ;*
        push    ebx                     ;*
        call    ExitProcess             ;*
```

```
; Процедура win_proc.
; Вызывается окном каждый раз, когда оно получает какое-нибудь сообщение.
; Именно здесь будет происходить вся работа программы.
;
; Процедура не должна изменять регистры EBP,EDI,ESI и EBX!
win_proc        proc                                    ;*
        push    ebp                                     ;*
        mov     ebp,esp                                 ;*
                wp_hWnd equ dword ptr [ebp+08h]         ;*
                wp_uMsg equ dword ptr [ebp+0Ch]         ;*
                wp_wParam equ dword ptr [ebp+10h]       ;*
                wp_lParam equ dword ptr [ebp+14h]       ;*
        cmp     wp_uMsg,WM_DESTROY                      ;*
        jne     not_wm_destroy                          ;*
        push    0                                       ;*
        call    PostQuitMessage                         ;*
        jmp     short end_wm_check                      ;*
not_wm_destroy:                                         ;*
        cmp     wp_uMsg,WM_COMMAND      ; Если мы получили WM_COMMAND -
        jne     not_wm_command          ; это от нашего меню
        mov     eax,wp_wParam           ; и в wParam лежит наше подсообщение.
        jmp     dword ptr menu_handlers[eax*4] ; Косвенный переход
        ; (в 32-битном режиме можно осуществлять переход по любому регистру).

menu_handlers   dd      offset menu_test,offset menu_open
                dd      offset menu_save,offset menu_exit
; Обработчики событий test, open и save выводят MessageBox.
; Обработчик exit выходит из программы.
menu_test:
        mov     eax,offset test_msg     ; Сообщение для MessageBox.
        jmp     short  showjnsg
menu_open:
        mov     eax,offset open_msg     ; Сообщение для MessageBox.
        jmp     short  show_msg
menu_save:
        mov     eax,offset save_msg     ; Сообщение для MessageBox.
show_msg:
        push    MB_OK                   ; Стиль для MessageBox.
        push    offset menu_name        ; Заголовок.
        push    eax                     ; Сообщение.
        push    wp_hWnd                 ; Идентификатор окна-предка.
        call    MessageBox              ; Вызов функции.
        jmp     short end_wm_check      ; Выход из win_proc.
menu_exit:                              ; Если выбрали пункт EXIT,
        push    wp_hWnd
        call    DestroyWindow           ; уничтожить наше окно.

end_wm_check:
        leave                           ;*
        xor     eax,eax         ; Вернуть 0 как результат работы процедуры.
        ret     16              ;*
```

```
not_wm_command:                    ; not_wm_command, чтобы избавиться от лишнего jmp.
        leave                      ;*
        jmp     DefWindowProc      ;*
win_proc        endp               ;*
         end    _start             ;*
```

Итак, из 120 строк программы новыми оказались всего 36, а сама программа, с точки зрения пользователя, стала намного сложнее. Таким образом выглядит программирование под Windows на ассемблере - берется одна написанная раз и навсегда шаблонная программа, модифицируются ресурсы и пишутся обработчики для различных событий меню и диалогов. Фактически все программирование оказывается сосредоточенным именно в этих процедурах-обработчиках.

Добавления к включаемым файлам в этом примере тоже оказываются незначительными по сравнению с window.asm.

В user32.inc между ifdef _TASM_ и else:

```
                extrn   LoadMenuA:near
                extrn   DestroyWindow:near
LoadMenu        equ     LoadMenuA
```

и между else и endif:

```
                extrn   __imp__LoadMenuA@8:dword
                extrn   __imp__DestroyWindow@4:dword
LoadMenu        equ     __imp__LoadMenuA@8
DestroyWindow   equ     __imp__DestroyWindow@8
```

и в def32.inc:

```
; Из winuser.h.
WM_COMMAND      equ     111h
MB_OK           equ     0
```

7.3.4. *Диалоги*

Графические программы для Windows почти никогда не ограничиваются одним меню, потому что оно не позволяет ввести реальную информацию - только выбрать какой-либо пункт из предложенных. Конечно, в цикле после GetMessage или PeekMessage можно обрабатывать события передвижения мыши и нажатия клавиш, и так делают в интерактивных программах, например в играх, но если требуется ввести текст и в дальнейшем его редактировать, выбрать файл на диске и т. п., то основным средством ввода информации в программах для Windows оказываются диалоги.

Диалог описывается, так же как и меню, в файле ресурсов, но если меню легко написать вручную, то ради диалогов, скорее всего, придется пользоваться каким-нибудь редактором, идущим в комплекте с вашим любимым компилятором, при условии, конечно, что вы не знаете в точности, по каким координатам располагается каждый контрол (активный элемент диалога).

```
// windlg.rc
// Файл ресурсов, описывающий диалог, который используется в программе windlg.asm.
// Все следующие определения можно заменить на #include <winuser.h>, если он есть.
```

```
// Стили для диалогов.
#define DS_CENTER          0x0800L
#define DS_MODALFRAME      0x80L
#define DS_3DLOOK          0x0004L
// Стили для окон.
#define WS_MINIMIZEBOX     0x00020000L
#define WS_SYSMENU         0x00080000L
#define WS_VISIBLE         0x10000000L
#define WS_OVERLAPPED      0x00000000L
#define WS_CAPTION         0xC00000L
// Стили для редактора.
#define ES_AUTOHSCROLL     0x80L
#define ES_LEFT            0

#define ZDLG_MENU7
// Идентификаторы контролов диалога.
#define IDC_EDIT           0
#define IDC_BUTTON         1
#define IDC_EXIT           2
// Идентификаторы пунктов меню.
#define IDM_GETTEXT        10
#define IDM_CLEAR11
#define IDM_EXIT           12

ZZZ_Dialog DIALOG 10,10,205,30                    // x, y, ширина, высота.
STYLE DS_CENTER | DS_MODALFRAME | DS_3DLOOK | WS_CAPTION I WS_MINIMIZEBOX I
WS_SYSMENU | WS_VISIBLE | WS_OVERLAPPED
CAPTION "Win32 assembly dialog example"           // Заголовок.
MENU ZDLG_MENU                                    // Меню.
BEGIN                                             // Начало списка контролов.
        EDITTEXT          IDC_EDIT,15,7,111,13,ES_AUTOHSCROLL | ES_LEFT
        PUSHBUTTON        "E&xit",IDC_EXIT,141,8,52,13
END

ZDLG_MENU MENU                                    // Меню.
BEGIN
        POPUP "Test"
        BEGIN
                MENUITEM "Get Text",IDM_GETTEXT
                MENUITEM "Clear Text",IDM_CLEAR
                MENUITEM SEPARATOR
                MENUITEM "E&xit",IDM_EXIT
        END
END
```

На простом примере покажем, как можно применять диалог, даже не регистрируя нового класса. Для этого надо создать диалог командой CreateDialog или одним из ее вариантов, не конфигурируя никакого окна-предка. Все сообщения от диалога и окон, которые он создает, будут посылаться в процедуру-обработчик типа DialogProc, аналогичную WindowProc.

```
; windlg.asm
; Графическое win32-приложение, демонстрирующее работу с диалогом.
; Идентификаторы контролов (элементов диалога).
IDC_EDIT        equ  0
IDC_BUTTON      equ  1
IDC_EXIT        equ  2
; Идентификаторы элементов меню.
IDM_GETTEXT     equ 10
IDM_CLEAR       equ 11
IDM_EXIT        equ 12

include def32.inc
include kernel32.inc
include user32.inc

        .386
        .model  flat
        .data
dialog_name     db      "ZZZ_Dialog",0   ; Имя диалога в ресурсах.
        .data?
buffer          db      512 dup(?)       ; Буфер для введенного текста.
        .code
_start:
        xor     ebx,ebx                  ; В EBX будет 0 для команд push 0
                                         ; (короче в 2 раза).
; Определить идентификатор нашей программы.
        push    ebx
        call    GetModuleHandle
; Запустить диалог.
        push    ebx     ; Значение, которое перейдет как параметр WM_INITDIALOG.
        push    offset dialog_proc ; Адрес процедуры типа DialogProc.
        push    ebx     ; Идентификатор окна-предка (0 - ничей диалог).
        push    offset dialog_name ; Адрес имени диалога в ресурсах.
        push    eax     ; Идентификатор программы, в ресурсах которой
                        ; находится диалог (наш идентификатор в EAX).
        call    DialogBoxParam
; Выход из программы.
        push    ebx
        call    ExitProcess
;
; Процедура dialog_proc.
; Вызывается диалогом каждый раз, когда в нем что-нибудь происходит.
; Именно здесь будет происходить вся работа программы.
;
; Процедура не должна изменять регистры EBP,EDI,ESI и EBX!
;
dialog_proc     proc    near
; Так как мы получаем параметры в стеке, построим стековый кадр.
        push    ebp
        mov     ebp,esp
```

```
; Процедура типа DialogProc вызывается со следующими параметрами:
        dp_hWnd equ dword ptr [ebp+08h]  ; идентификатор диалога,
        dp_uMsg equ dword ptr [ebp+0Ch]  ; номер сообщения,
        dp_wParam equ dword ptr [ebp+10h]; первый параметр,
        dp_lParam equ dword ptr [ebp+14h] ; второй параметр.
        mov     ecx,dp_hWnd              ; ECX будет хранить идентификатор диалога,
        mov     eax,dp_uMsg              ; а EAX - номер сообщения.
        cmp     eax,WM_INITDIALOG        ; Если мы получили WM_INITDIALOG,
        jne     not_initdialog
        push    IOC_EOIT
        push    dp_hWnd
        call    GetDlgItem               ; определить идентификатор
        push    eax                      ; окна редактирования текста
        call    SetFocus                 ; и передать ему фокус.
not_initdialog:
        cmp     eax,WM_CLOSE             ; Если мы получили WM_CLOSE,
        jne     not_close
        push    0
        push    ecx
        call    EndDialog                ; закрыть диалог.
not_close:
        cmp     eax,WM_COMMAND           ; Если мы получили WM_COMMAND,
        jne     not_command
        mov     eax,dp_wParam            ; EAX = wParam (номер сообщения).
        cmp     dp_lParam,0              ; Если lparam ноль - сообщение от меню.
        jne     lParam_not_0
        cmp     ax,IDM_GETTEXT           ; Если это пункт меню Get Text.
        jne     not_gettext
        push    512                      ; Размер буфера.
        push    offset buffer            ; Адрес буфера.
        push    IOC_EDIT                 ; Номер контрола редактирования.
        push    ecx
        call    GetDlgItemText           ; Считать текст в буфер
        push    MB_OK
        push    offset dialog_name
        push    offset buffer
        push    dp_hWnd
        call    MessageBox               ; и показать его в MessageBox.
not_gettext:
        cmp     eax,IDM_CLEAR            ; Если это пункт меню Clear:
        jne     not_clear
        push    0                        ; NULL.
        push    IDC_EDIT                 ; Номер контрола.
        push    ecx
        call    SetDlgItemText           ; Установить новый текст.
not_clear:
        cmp     eax,IDM_EXIT             ; Если это пункт меню Exit:
        jne     not_exit
        push    0                        ; код возврата.
```

```
        push    ecx             ; Идентификатор диалога.
        call    EndDialog       ; Закрыть диалог.
lParam_not_0:                   ; lParam не ноль - сообщение от контрола.
        cmp     eax,IDC_EXIT    ; Если сообщение от кнопки Exit.
        jne     not_exit
        shr     eax,16
        cmp     eax,BN_CLICKED  ; Если ее нажали:
        push    0               ; код возврата.
        push    ecx             ; Идентификатор диалога.
        call    EndDialog       ; Закрыть диалог.
not_exit:
        xor     eax,eax         ; После обработки команды
        inc     eax             ; OialogProc должен возвращать TRUE (eax = 1).
        leave
        ret     16              ; Конец процедуры.
not_command:                    ; Сюда передается управление, если мы получили
                                ; какое-то незнакомое сообщение.
        xor     eax,eax         ; Код возврата FALSE (eax = 0).
        leave
        ret     16              ; Конец процедуры.
dialog_proc     endp
        end     _start
```

Добавления в наш user32.inc.

Между ifdef _TASM_ и else:

```
                extrn   DialogBoxParamA:near
                extrn   GetDlgItem:near
                extrn   SetFocus:near
                extrn   GetDlgItemTextA:near
                extrn   SetDlgItemTextA:near
                extrn   EndDialog:near
DialogBoxParam  equ     DialogBoxParamA
GetDlgItemText  equ     GetDlgItemTextA
SetDlgItemText  equ     SetDlgItemTextA
```

Между else и endif:

```
                extrn   __imp__DialogBoxParamA@20:dword
                extrn   __imp__GetDlgItem@8:dword
                extrn   __imp__SetFocus@4:dword
                extrn   __imp__GetDlgItemTextA@16:dword
                extrn   __imp__SetDlgItemTextA@12:dword
                extrn   __imp__EndDialog@8:dword
DialogBoxParam  equ     __imp__DialogBoxParamA@20
GetDlgItem      equ     __imp__GetDlgItem@8
SetFocus        equ     __imp__SetFocus@4
GetDlgltemText  equ     __imp__GetDlgItemTextA@16
SetDlgltemText  equ     __imp__SetDlgItemTextA@12
EndDialog       equ     __imp__EndDialog@8
```

Добавления к файлу def32.inc:

```
; Из winuser.h.
WM_INITDIALOG   equ     110h
```

```
WM_CLOSE        equ     10h
BN_CLICKED      equ     0
```

7.3.5. Полноценное приложение

Теперь, когда мы знаем, как строятся программы с меню и диалогами, попробуем написать настоящее полноценное приложение, включающее в себя все то, что требуется от программы, - меню, диалоги, комбинации клавиш для быстрого доступа к элементам меню и т. д. В качестве примера создадим простой текстовый редактор, аналогичный Notepad. В этом примере увидим, как получить параметры из командной строки, прочитать и записать файл, выделить и освободить память.

```
// winpad95.rc
// Файл ресурсов для программы winpad95.asm.

// Идентификаторы сообщений от пунктов меню.
#define IDM_NEW         0x100L
#define IDM_OPEN        0x101L
#define IDM_SAVE        0x102L
#define IDM_SAVEAS      0x103L
#define IDM_EXIT        0x104L
#define IDM_ABOUT       0x105L
#define IDM_UNDO        0x106L
#define IDM_CUT         0x107L
#define IDM_COPY        0x108L
#define IDM_PASTE       0x109L
#define IDM_CLEAR       0x10AL
#define IDM_SETSEL      0x10BL

// Идентификаторы основных ресурсов.
#define ID_MENU         0x700L
#define ID_ACCEL        0x701L
#define ID_ABOUT        0x702L

// Если есть иконка - можно раскомментировать эти две строки:
// «define ID_ICON      0x703L
// ID_ICON ICON "winpad95.ico"

// Основное меню.
ID_MENU MENU DISCARDABLE {
        POPUP "&File" {
                MENUITEM "&New\tCtrl+N", IDM_NEW
                MENUITEM "&Open...\tCtrl+O",IDM_OPEN
                MENUITEM "&Save\tCtrl+S", IDM_SAVE
                MENUITEM "Save &As...\tCtrl+Shift+S", IDM_SAVEAS
                MENUITEM  SEPARATOR
                MENUITEM "E&xit\tCtrl+Q",IDM_EXIT
        }
        POPUP "&Edit" {
                MENUITEM "&Undo\tCtrl-Z", IDM_UNDO
                MENUITEM SEPARATOR
```

```
                MENUITEM "Cu&t\tCtrl-X", IDM_CUT
                MENUITEM "&Copy\tCtrl-C", IDM_COPY
                MENUITEM "&Paste\tCtrl-V", IDM_PASTE
                MENUITEM "&Delete\tDel", IDM_CLEAR
                MENUITEM SEPARATOR
                MENUITEM "Select &All\tCtrl-A", IOM_SETSEL
        }
        POPUP "&Help" <
                MENUITEM "About", IDM_ABOUT
        }
}

// Комбинации клавиш.
ID_ACCEL ACCELERATORS DISCARDABLE {
        "N", IDM_NEW, CONTROL, VIRTKEY
        "O", IDM_OPEN, CONTROL, VIRTKEY
        "S", IDM_SAVE, CONTROL, VIRTKEY
        "S", IDM_SAVEAS, CONTROL, SHIFT, VIRTKEY
        "Q", IDM_EXIT, CONTROL, VIRTKEY
        "2", IDM_UNDO, CONTROL, VIRTKEY
        "A", IDM_SETSEL, CONTROL, VIRTKEY
}

// Все эти определения можно заменить #include <winuser.h>.
#define DS_MODALFRAME     0x80L
#define DS_3DLOOK         4
«define WS_POPUP          0x80000000L
#define WS_CAPTION        0xC00000L
#define WS_SYSMENU        0x80000L
#define IDOK              1
#define IDC_STATIC        -1
«define IDI_APPLICATION  32512
#define WS_BORDER         0x800000L

// Стандартный диалог "About".
ID_ABOUT DIALOG DISCARDABLE 0, 0, 125, 75
STYLE DS_MODALFRAME | DS_3DLOOK | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About Asmpad95"
{
        ICON            IDI_APPLICATION, IDC_STATIC, 12, 15, 21, 20
        CTEXT           "Asmpad95", IDC_STATIC, 0, 30, 40, 8
        CTEXT           "Prototype notepad-style editor for Windows 95 written
entirely in assembly language", IDC_STATIC, 45, 10, 70, 45, WS_BORDER
        DEFPUSHBUTTON   "OK", IDOK, 35, 60, 40, 10
}
```

Далее рассмотрим текст программы.

```
; winpad95.asm
; Графическое win32-приложение - текстовый редактор.
```

```
include def32.inc
include user32.inc
include kernel32.inc
include comdlg32.inc

ID_MENU         equ     700h
ID_ACCEL        equ     701h
ID_ABOUT        equ     702h

MAXSIZE         equ     260     ; Максимальное имя файла.
MEMSIZE         equ     65535   ; Максимальный размер временного буфера в памяти.

EditID          equ     1

                .386
                .model  flat
                .const
c_w_name db "Asmpad95",0            ; Это и имя класса, и имя основного окна.
edit_class      db      "edit",0 ; Предопределенное имя класса для редактора.
changes_msg     db      "Save changes?",0
filter_string   db      "All Files",0,'*.*',0    ; Маски для Get*FileName.
                db      "Text Files",0,'*.txt',0,0
        .data
; Структура, использующаяся Get*FileName.
ofn     OPENFILENAME <SIZE ofn,?,?,offset filter_string,?,?,?,offset buffer,\
                      MAXSIZE,0,?,?,?,?,?,?,0,?,?,?>
; Структура, описывающая наш основной класс.
wc      WNDCLASSEX <SIZE WNDCLASSEX,CS_HREDRAW or CS_VREDRAW,offset win_proc,0,\
                      0,?,?,?,COLOR_WINDOW+1,ID_MENU,offset c_w_name,0>
flag_untitled   db      1; = 1, если имя файла не определейо (новый файл).

        .data?
h_editwindow    dd      7       ; Идентификатор окна редактора.
h_accel         dd      7       ; Идентификатор массива акселераторов.
p_memory        dd      ?       ; Адрес буфера в памяти.
SizeReadWrite   dd      ?
msg_            MSG     <>
rec             RECT    <>
buffer          db      MAXSIZE dup(?)   ; Имя файла.
window_title    db      MAXSIZE dup(?), 12 dup(?)

        .code
_start:
        call    GetCommandLine  ; Получить нашу командную строку.
        mov     edi,eax
        mov     al,' '
        mov     ecx,MAXSIZE
        repne   scasb           ; Найти конец имени нашей программы.
        cmp     byte ptr [edi],0
        je      cmdline_empty
        mov     esi,edi
        mov     edi,offset buffer
```

```
        rep     movsb
        mov     flag_untitled,0
cmdline_empty:
; Подготовить и зарегистрировать класс.
        xor     ebx,ebx
        call    GetModuleHandle          ; Определить наш идентификатор
        mov     esi,eax
        fflov   wc.hInstance,eax         ; и сохранить его в wc.hInstance
        mov     ofn._hInstance,eax
        push    IDI_APPLICATION          ; или IDI_ICON, если иконка есть в ресурсах,
        push    ebx                      ; или esi, если иконка есть в ресурсах.
        call    LoadIcon
        mov     wc.hIcon,eax
        push    IDC_ARROW                ; Предопределенный курсор (стрелка).
        push    ebx
        call    LoadCursor
        mov     wc.hCursor,eax
        push    offset wc
        call    RegisterClassEx
; Создать основное окно.
        push    ebx
        push    esi
        push    ebx
        push    ebx
        push    200
        push    300
        push    CW_USEDEFAULT
        push    CW_USEDEFAULT
        push    WS_OVERLAPPEDWINDOW
        push    offset c_w_name
        push    offset c_w_name
        push    WS_EX_CLIENTEDGE
        call    CreateWindowEx
        push    eax                      ; Для pop esi перед message_loop.
        push    eax
        push    SW_SHOWNORMAL
        push    eax
        call    ShowWindow
        call    UpdateWindow
; Инициализировать акселераторы.
        push    ID_ACCEL
        push    esi
        call    LoadAccelerators
        mov     h_accel,eax
; Цикл ожидания сообщения.
        pop     esi                      ; ESI - идентификатор основного окна.
        mov     edi,offset msg_          ; EDI - структура с сообщением от него.
message_loop:
        push    ebx
        push    ebx
```

```
        push    ebx
        push    edi
        call    GetMessage              ; Получить сообщение.
        test    eax,eax                 ; Если это WM_QUIT,
        jz      exit_msg_loop           ; выйти из цикла.
        push    edi
        push    h_accel
        push    esi                     ; hWnd
        call    TranslateAccelerator    ; Преобразовать акселераторы в IDM*
        test    eax,eax
        jnz     message_loop
        push    edi
        call    TranslateMessage        ; Преобразовать сообщения от клавиш
        push    edi
        call    DispatchMessage         ; и отправить обратно.
        jmp     short message_loop
exit_msg_loop:
        push    msg_.wParam
        call    ExitProcess             ; Конец программы.

; Процедура win_proc.
;
; Процедура не должна изменять регистры EBP,EDI,ESI и EBX!
win_proc proc near
; Параметры. (с учетом push ebp).
wp_hWnd equ dword ptr [ebp+08h]
wp_uMsg equ dword ptr [ebp+0Ch]
wp_wParam equ dword ptr [ebp+10h]
wp_lParam equ dword ptr [ebp+14h]
; Инициализируем стековый кадр.
        push    ebp
        mov     ebp,esp                 ; Создать стековый кадр.
        pusha                           ; Сохранить все регистры.
        xor     ebx,ebx                 ; 0 для команд push 0.
        mov     esi,wp_hWnd             ; Для команд push hWnd.
        mov     eax,wp_uMsg
; Обработать пришедшее сообщение.
        cmp     eax,WM_CREATE
        je      h_wm_create
        cmp     eax,WM_SIZE
        je      h_wm_size
        cmp     eax,WM_DESTROY
        je      h_wm_destroy
        cmp     eax,WM_COMMAND
        je      h_wm_command
        cmp     eax,WM_ACTIVATE
        je      h_wm_activate
        cmp     eax,WM_CLOSE
        je      h_wm_close
```

```
def_proc:
        popa
        leave                           ; Если эта ненужное сообщение,
        jmp     DefWindowProc           ; оставить его обработчику по умолчанию.

; Обработчик WM_CLOSE.
; Если нужно, спрашивает, сохранить ли файл.
h_wm_close:
        call    save_contents
        jmp     short def_proc

; Обработчик WM_CREATE.
;
h_wm_create:
; Здесь также можно создать toolbar и statusbar.
; Создать окно редактора.
        push    ebx
        push    wc.hInstance            ; Идентификатор основной программы.
        push    EditID
        push    esi                     ; hWnd
        push    ebx                     ; 0
        push    ebx                     ; 0
        push    ebx                     ; 0
        push    ebx                     ; 0
        push    WS_VISIBLE or WS_CHILD or ES_LEFT or ES_MULTILINE or\
                ES_AUTOHSCROLL or ES_AUTOVSCROLL
        push    ebx                     ; 0
        push    offset edit_class
        push    ebx                     ; 0
        call    CreateWindowEx
        mov     h_editwindow,eax
; Передать ему фокус.
        push    eax
        call    SetFocus
        cmp     flag_untitled,1
        je      contlnue_create
        call    skip_getopen    ; Открыть файл, указанный в командной строке.
continue_create:
        call    set_title
        jmp     end_wm_check

; Обработчик WM_COMMAND.
;
h_wm_command:
        mov     eax,wp_wParam
        cwde                    ; Младшее слово содержит IDM_*.
        sub     eax,100h
        jb      def_proc
; Обработать сообщения от пунктов меню.
        call    dword ptr menu_handlers[eax*4]
        jmp     end_wm_check
```

```
menu_handlers
                dd      offset h_idm_new,offset h_idm_open,offset h_idm_save
                dd      offset h_idm_saveas,offset h_idm_exit,offset h_idm_about
                dd      offset h_idm_undo, offset h_idm_cut, offset h_idm_copy
                dd      offset h_idm_paste, offset h_idm_clear, offset h_idm_setsel
; Сообщения от пунктов меню должны быть описаны в win95pad.rc именно в таком
; порядке - от IDM_NEW 100h до IDM_CLEAR 10Ah.

h_idm_setsel:
        push    -1              ; -1
        push    ebx             ; 0
        push    EM_SETSEL       ; Выделить весь текст.
        push    h_editwindow
        call    SendMessage
        ret

; Обработчики сообщений из меню EDIT:
h_idm_clear:
        mov     eax,WM_CLEAR
        jmp     short send_to_editor
h_idm_paste:
        mov     eax,WM_PASTE
        jmp     short send_to_editor
h_idm_copy:
        mov     eax,WM_COPY
        jmp     short send_to_editor
h_idm_cut:
        mov     eax,WM_CUT
        jmp     short send_to_editor
h_idm_undo:
        mov     eax,EM_UNDO
send_to_editor:
        push    ebx             ; 0
        push    ebx             ; 0
        push    eax
        push    h_editwindow
        call    SendMessage
        ret

; Обработчик IDM_NEW.
h_idm_new:
        call    save_contents   ; Записать файл, если нужно.
        mov     byte ptr flag_untitled,1
        call    set_title       ; Отметить, что файл не назван.
        push    ebx
        push    ebx
        push    WM_SETTEXT
        push    h_editwindow
        call    SendMessage     ; Послать пустой WM_SETTEXT редактору.
        ret
```

```
; Обработчик IDM_ABOUT.
h_idm_about:
        push    ebx             ; 0
        push    offset about_proc
        push    esi             ; hWnd
        push    ID_ABOUT
        push    wc.hInstance
        call    DialogBoxParam
        ret

; Обработчик IDM_SAVEAS и IDM_SAVE.
h_idm_save:
        cmp     flag_untitled, 1  ; Если файл назван,
        jne     skip_getsave      ; пропустить вызов GetSaveFileName.
h_idm_saveas:
; Узнать имя файла.
        mov     ofn.Flags,OFN_EXPLORER or OFN_OVERWRITEPROMPT
        push    offset ofn
        call    GetSaveFileName
        test    eax,eax
        jz      file_save_failed
skip_getsave:
; Создать его.
        push    ebx
        push    FILE_ATTRIBUTE_ARCHIVE
        push    CREATE_ALWAYS
        push    ebx
        push    FILE_SHARE_READ or FILE_SHARE_WRITE
        push    GENERIC_READ or GENERIC_WRITE
        push    offset buffer
        call    CreateFlle
        mov     edi,eax
; Выделить память.
        push    MEMSIZE
        push    GMEM_MOVEABLE or GMEM_ZEROINIT
        call    GlobalAlloc
        push    eax             ; hMemory для GlobalFree.
        push    eax             ; hMemory для GlobalLock.
        call    GlobalLock
        mov     esi,eax         ; Адрес буфера в ESI.
; Забрать текст из редактора.
        push    esi
        push    MEMSIZE-1
        push    WM_GETTEXT
        push    h_editwindow
        call    SendMessage
; Записать в файл.
        push    esi             ; pMemory
        call    lstrlen
        push    ebx
```

```
        push    offset SizeReadWrite
        push    eax             ; Размер буфера.
        push    esi             ; Адрес буфера.
        push    edi             ; Идентификатор файла.
        call    WriteFile
        push    esi             ; pMemory
        call    GlobalUnlock
        call    GlobalFree      ; hMemory уже в стеке.
        push    edi             ; Идентификатор файла.
        call    CloseHandle
; Сбросить флаг модификации в редакторе.
        push    ebx
        push    ebx
        push    EM_SETMODIFY
        push    h_editwindow
        call    SendMessage
        mov     byte ptr flag_untitled,0
        call    set_title
file_save_failed:
        push    h_editwindow
        call    SetFocus
        ret

; Обработчик IDM_OPEN.
h_idm_open:
        call    save_contents
; Вызвать стандартный диалог выбора имени файла.
        mov     ofn.Flags, OFN_FILEMUSTEXIST or OFN_PATHMUSTEXIST or\
                OFN_EXPLORER
        push    offset ofn
        call    GetOpenFileName
        test    eax,eax
        jz      file_open_failed
skip_getopen:
; Открыть выбранный файл.
        push    ebx
        push    FILE_ATTRIBUTE_ARCHIVE
        push    OPEN_EXISTING
        push    ebx
        push    FILE_SHARE_READ or FILE_SHARE_WRITE
        push    GENERIC_READ or GENERIC_WRITE
        push    offset buffer
        call    CreateFile
        mov     edi,eax         ; Идентификатор для ReadFile.
; Выделить память.
        push    MEMSIZE
        push    GMEM_MOVEABLE or GMEM_ZEROINIT
        call    GlobalAlloc
        push    eax             ; hMemory для GlobalFree.
        push    eax             ; hMemory для GlobalLock.
```

```
        call    GlobalLock          ; Получить адрес выделенной памяти.
        push    eax                 ; pMemory для GlobalUnlock.
        push    eax                 ; pMemory для SendMessage.
; Прочитать файл.
        push    ebx
        push    offset SizeReadWrite
        push    MEMSIZE-1
        push    eax                 ; pMemory для ReadFlle.
        push    edi
        call    ReadFile
; Послать окну редактора сообщение wm_settext о том,
; что нужно забрать текст из буфера.
        push    ebx                 ; pMemory уже в стеке.
        push    WM_SETTEXT
        push    h_editwindow
        call    SendMessage
; А теперь можно закрыть файл и освободить память.
        call    GlobalUnlock        ; pMemory уже в стеке.
        call    GlobalFree          ; hMemory уже в стеке.
        push    edi                 ; hFile.
        call    CloseHandle
        mov     byte ptr flag_untitled,0
        call    set_title
file_open_failed:
        push    h_editwindow
        call    SetFocus
        ret

; Обработчик IDM_EXIT.
h_idm_exit:
        call    save_contents
        push    esi                 ; hWnd
        call    DestroyWindow       ; Уничтожить наше окно.
        ret

; Обработчик WM_SIZE.

h_wm_size:
; Здесь также надо послать WM_SIZE окнам toolbar и statusbar, изменить размер окна
; редактора так, чтобы оно по-прежнему было максимального размера.
        push    offset rec
        push    esi                 ; hWnd
        call    GetClientRect
        push    1                   ; true
        push    rec.bottom          ; height
        push    rec.right           ; width
        push    ebx                 ; y
        push    ebx                 ; x
        push    h_editwindow
        call    MoveWindow
        jmp     short end_wm_check
```

```
; Обработчик  WM_ACTIVATE.
;
h_wm_activate:
        push    h_editwindow
        call    SetFocus
        jmp     short end wm_check

; Обработчик WM_DESTROY.
;
h_wm_destroy:
        push    ebx
        call    PostQuitMessage          ; Послать WM_QUIT основной программе.

; Конец процедуры window_proc.
end_wm_check:
        popa
        xor     eax,eax                  ; Вернуть 0.
        leave
        ret     16

; Процедура set_title.
; Устанавливает новый заголовок для основного окна.
set_title:
        push    esi
        push    edi
        mov     edi,offset window_title
        cmp     byte ptr flag_untitled,1 ; Если у файла нет имени,
        je      untitled                 ; использовать Untitled.
        mov     esi,ofn.lpstrFile        ; [ESI] - имя файла с путем.
        movzx   eax,ofn.nFileOffset      ; EAX - начало имени файла.
        add     esi,eax
copy_filename:
        lodsb                    ; Скопировать файл побайтово в название окна,
        test    al,al
        jz      add_progname             ; пока не встретится ноль.
        stosb
        jmp     short copy_filename
add_progname:
        mov     dword ptr [edi],' - '    ; Приписать минус
        add     edi,3
        mov     esi,offset c_w_name
        mov     ecx,9                    ; и название программы.
        rep     movsb
        pop     edi
        pop     esi
        push    offset window_title
        push    esi                      ; Идентификатор окна.
        call    SetWindowText
        ret

untitled:
        mov     dword ptr [edi],'itnU'   ; Дописать "Unti".
```

```
        mov     dword ptr [edi+4],'delt' ; Дописать "tled".
        add     edi,8
        jmp     short add_progname

; Процедура save_contents.
; EBX = 0, ESI = hWnd
save_contents:
; Спросить редактор, изменялся ли текст.
        push    ebx
        push    ebx
        push    EM_GETMODIFY
        push    h_editwindow
        call    SendMessage
        test    eax,eax
        jz      not_modified
; Спросить пользователя, сохранять ли его.
        push    MB_YESNO + MB_ICONWARNING
        push    offset  c_w_name
        push    offset  changes_msg
        push    esi
        call    MessageBox
        cmp     eax,IDYES
        jne     notjnodified
; Сохранить его.
        call    h_idm_save
not_modified:
        ret

win_proc        endp

about_proc      proc near
; Параметры (с учетом push ebp).
ap_hDlg         equ dword ptr [ebp+08h]
ap_uMsg         equ dword ptr [ebp+0Ch]
ap_wParam       equ dword ptr [ebp+10h]
ap_lParam       equ dword ptr [ebp+14h]
        push    ebp
        mov     ebp,esp                 ; Создать стековый кадр.
        cmp     ap_uMsg,WM_COMMAND
        jne     dont_proceed
        cmp     ap_wParam,IDOK
        jne     dont_proceed
        push    1
        push    ap_hDlg
        call    EndDialog
dont_proceed:
        xor     eax,eax                 ; Не обрабатывается.
        leave
        ret     16
about_proc      endp

        end     _start
```

Размер этой программы - 6,5 Кб (скомпилированной ml/link), и даже версия, в которую добавлено все, что есть в Notepad (вывод файла на печать и поиск по тексту), получилась меньше notepad.exe почти в четыре раза. Чем значительнее Windows-приложение создается, тем сильнее сказывается выигрыш в размерах при использовании ассемблера, даже несмотря На то, что мы лишь вызываем системные функции, практически не занимаясь программированием.

Прежде чем можно будет скомпилировать winpad95.asm, следует внести необходимые дополнения в наши включаемые файлы.

Добавления в файл def32.inc:

```
; Из winuser.h.
WM_CREATE                   equ     1
WM_ACTIVATE                 equ     6
WM_SETTEXT                  equ     0Ch
WM_GETTEXT                  equ     0Dh
WM_CUT                      equ     300h
WM_COPY                     equ     301h
WM_PASTE                    equ     302h
WM_CLEAR                    equ     303h
WM_UNDO                     equ     304h
WM_SIZE                     equ     5
WS_VISIBLE                  equ     10000000h
WS_CHILD                    equ     40000000h
WS_EX_CLIENTEDGE            equ     200h
ES_LEFT                     equ     0
ES_MULTILINE                equ     4
ES_AUTOHSCROLL              equ     80h
ES_AUTOVSCROLL              equ     40h
EM_GETHANDLE                equ     0BDh
EM_GETMODIFY                equ     0B8h
EM_SETMODIFY                equ     0B9h
EM_UNDO                     equ     0C7h
EM_SETSEL                   equ     0B1h
MB_YESNO                    equ     4
MB_ICONWARNING              equ     30h
IDOK                        equ     1
IDYES                       equ     6

; Из winnt.h.
GENERIC_READ                equ     80000000h
GENERIC_WRITE               equ     40000000h
FILE_SHARE_READ             equ     1
FILE_SHARE_WRITE            equ     2
FILE_ATTRIBUTE_ARCHIVE      equ     20h

; Из commdlg.h.
OFN_PATHMUSTEXIST           equ     800h
OFN_FILEMUSTEXIST           equ     1000h
OFN_EXPLORER                equ     80000h
```

```
OFN_OVERWRITEPROMPT     equ     2
OPENFILENAME            struc
                StructSize              dd      ?
                hwndOwner               dd      ?
                _hInstance              dd      ?
                lpstrFilter             dd      ?
                lpstrCustomFilter       dd      ?
                nMaxCustFilter          dd      ?
                nFilterIndex            dd      ?
                lpstrFile               dd      ?
                nMaxFile                dd      ?
                lpstrFileTitle          dd      ?
                nMaxFileTitle           dd      ?
                lpstrInitialDir         dd      ?
                lpstrTitle              dd      ?
                Flags                   dd      ?
                nFileOffset             dw      ?
                nFileExtension          dw      ?
                lpstrDefExt             dd      ?
                lCustData               dd      ?
                lpfnHook                dd      ?
                lpTemplateName          dd      ?
OPENFILENAME    ends

; Из windef.h.
RECT            struc
                left                    dd      ?
                top                     dd      ?
                right                   dd      ?
                bottom                  dd      ?
RECT            ends

; Из winbase.h.
GMEM_MOVEABLE           equ     2
GMEM_ZEROINIT           equ     40h
OPEN_EXISTING           equ     3
CREATE_ALWAYS           equ     2
```

Добавления в файл kernel32.inc между ifdef _TASM_ и else:

```
                extrn           lstrlen:near
                extrn           GetCommandLineA:near
                extrn           CloseHandle:near
                extrn           GlobalAlloc:near
                extrn           GlobalLock:near
                extrn           GlobalFree:near
                extrn           CreateFileA:near
                extrn           ReadFile:near
                extrn           WriteFile:near
GetCommandLine  equ             GetCommandLineA
CreateFile      equ             •Cr'eateFileA
```

и между else и endif:

```
                    extrn       __imp__lstrlen@4:dword
                    extrn       __imp_GetCommandLineA@0:dword
                    extrn       __imp__CloseHandle@4:dword
                    extrn       __imp__GlobalAlloc@8:dword
                    extrn       __imp__GlobalLock@4:dword
                    extrn       __imp__GlobalFree@4:dword
                    extrn       __imp__CreateFileA@28:dword
                    extrn       __imp__ReadFile@20:dword
                    extrn       __imp__WriteFile@20:dword
lstrlen             equ         __imp__lstrlen@4
GetCommandLine      equ         __imp__GetCommandLineA@0
CloseHandle         equ         __imp__CloseHandle@4
GlobalAlloc         equ         __imp__GlobalAlloc@8
GlobalLock          equ         __imp__GlobalLock@4
GlobalFree          equ         __imp__GlobalFree@4
CreateFile          equ         __imp__CreateFileA@28
ReadFile            equ         __imp__ReadFile@20
WriteFile           equ         __imp__WriteFile@20
```

Добавления в файл user32.inc:

```
                    extrn       LoadAcceleratorsA:near
                    extrn       TranslateAccelerator:near
                    extrn       SendMessageA:near
                    extrn       SetWindowTextA:near
                    extrn       MoveWindow:near
                    extrn       GetClientRect:near
                    extrn       GlobalUnlock:near
LoadAccelerators    equ         LoadAcceleratorsA
SendMessage         equ         SendMessageA
SetWindowText       equ         SetWindowTextA
```

и между else и endif:

```
                    extrn       __imp__LoadAcceleratorsA@8:dword
                    extrn       __imp__TranslateAccelerator@12:dword
                    extrn       __imp__SendMessageA@16:dword
                    extrn       __imp__SetWindowTextA@8:dword
                    extrn       __imp__MoveWindow@24:dword
                    extrn       __imp__GetClientRect@8:dword
                    extrn       __imp__GlobalUnlock@4:dword
LoadAccelerators    equ         __imp__LoadAcceleratorsA@8
TranslateAccelerator equ        __imp__TranslateAccelerator@12
SendMessage         equ         __imp__SendMessageA@16
SetWindowText       equ         __imp__SetWindowTextA@8
MoveWindow          equ         __imp__MoveWindow@24
GetClientRect       equ         __imp__GetClientRect@8
GlobalUnlock        equ         __imp__GlobalUnlock@4
```

Кроме того, нам потребуется новый включаемый файл, comdlg32.inc, который описывает функции, связанные с вызовами стандартных диалогов (выбор имени файла, печать документа, выбор шрифта и т. д.):

```
; comdlg32.inc
; Включаемый файл с функциями из comdlg32.dll.

ifdef _TASM_
                includelib import32.lib
                        extrn           GetOpenFileNameA:near
                        extrn           GetSaveFileNameA:near
        GetOpenFileName  equ            GetOpenFileNameA
        GetSaveFileName  equ            GetSaveFileNameA
else
                includelib comdlg32.lib
; Истинные имена используемых функций.
                        extrn           __imp__GetOpenFileNameA@4:dword
                        extrn           __imp__GetSaveFileNameA@4:dword
; Присваивания для удобства использования.
GetOpenFileName         equ             __imp__GetOpenFileNameA@4
GetSaveFileName         equ             __imp__GetSaveFileNameA@4
endif
```

Конечно, эту программу можно еще очень долго совершенствовать — добавить toolbar и statusbar, написать документацию или сделать так, чтобы выделялось не фиксированное небольшое количество памяти для переноса файла в редактор, а равное его длине. Разрешается также применять функции отображения части файла в память (CreateFileMapping, OpenFileMapping, MapViewOfFile, UnmapViewOfFile), что позволит работать с файлами больших размеров. В Win32 API так много функций, что можно очень долго заниматься лишь их изучением.

7.4. Динамические библиотеки

Кроме обычных приложений в Windows появился специальный тип файла - *динамические библиотеки* (DLL). DLL - это файл, содержащий процедуры и данные, которые доступны программам, обращающимся к нему. Например, все системные функции Windows, которыми мы пользовались, на самом деле были процедурами, входящими в состав таких библиотек, - kernel32.dll, user32.dll, comdlg32.dll и т. д. Динамические библиотеки позволяют уменьшить использование памяти и размер исполняемых файлов в тех случаях, когда несколько программ (или даже копий одной программы) используют одну и ту же процедуру. Можно считать, что DLL - это аналог пассивной резидентной программы, с тем лишь отличием, что его нет в памяти, если ни одна программа, работающая с ним, не загружена.

С точки зрения программирования на ассемблере DLL — это самый обычный исполняемый файл формата PE, отличающийся только тем, что при входе в него в стеке передается три параметра (идентификатор DLL-модуля, причина вызова

процедуры и зарезервированный параметр), которые надо удалить, например командой ret 12. Помимо этой процедуры в DLL входят и другие, часть которых можно вызывать из разных программ. Список экспортируемых процедур должен быть задан во время компиляции DLL, и поэтому команды для создания нашего следующего примера будут отличаться от обычных.

Компиляция MASM:

```
ml /c /coff /Cp /D_MASM_ dllrus.asm
link dllrus.obj @dllrus.lnk
```

Содержимое файла dllrus.lnk:

```
/DLL
/entry:start
/subsystem:windows
/export:koi2win_asm
/export:koi2win
/export:koi2wins_asm
/export:koi2wins
```

Компиляция TASM:

```
tasm /m /x /ml /D_TASM_ dllrus.asm
tlink32 -Tpd -c dllrus.obj,,,dllrus.def
```

Содержимое файла dllrus.def:

```
EXPORTS koi2win_asm koi2win koi2wins koi2wins_asm
```

Компиляция WASM:

```
wasm dllrus.asm
wlink @dllrus.dir
```

Содержимое dllrus.dir:

```
file    dllrus.obj
form    windows nt DLL
exp     koi2win_asm,koi2win,koi2wins_asm,koi2wins
```

```
; dllrus.asm
; DLL для Win32 - перекодировщик из koi8 в cp1251.
;
                .386
                .model flat
; Функции, определяемые в DLL-файле.
ifdef _MASM_
        public _koi2win_asm@0   ; koi2win_asm - перекодирует символ в AL.
        public _koi2win@4       ; CHAR WINAPI koi2win(CHAR symbol).
        public _koi2wins_asm@0  ; koi2wins_asm - перекодирует строку в [EAX].
        public _koi2wins@4      ; VOID WINAPI koi2win(CHAR* string).
else
        public koi2win_asm      ; Те же функции для TASM и WASM.
        public koi2win
        public koi2wins_asm
        public koi2wins
endif

                .const
```

```
; Таблица для перевода символа из кодировки KOI8-r (RFC1489)
; в кодировку Windows (cp1251), таблица только для символов 80h - FFh
; (то есть надо будет вычесть 80h из символа, преобразовать его командой xlat
; и снова добавить 80h).
k2w_tbl         db      16 dup(0)       ; Символы, не существующие в cp1251,
                db      16 dup(0)       ; перекодируются в 80h.
                db      00h, 00h, 00h, 38h, 00h, 00h, 00h, 00h
                db      00h, 00h, 00h, 00h, 00h, 00h, 00h, 00h
                db      00h, 00h, 00h, 28h, 00h, 00h, 00h, 00h
                db      00h, 00h, 00h, 00h, 00h, 00h, 00h, 00h
                db      7Eh, 60h, 61h, 76h, 64h, 65h, 74h, 63h
                db      75h, 68h, 69h, 6Ah, 6Bh, 6Ch, 6Dh, 6Eh
                db      6Fh, 7Fh, 70h, 71h, 72h, 73h, 66h, 62h
                db      7Ch, 7Bh, 67h, 78h, 7Dh, 79h, 77h, 7Ah
                db      5Eh, 40h, 41h, 56h, 44h, 45h, 54h, 43h
                db      55h, 48h, 49h, 4Ah, 4Bh, 4Ch, 4Dh, 4Eh
                db      4Fh, 5Fh, 50h, 51h, 52h, 53h, 46h, 42h
                db      5Ch, 5Bh, 47h, 58h, 5Dh, 59h, 57h, 5Ah

        .code
; Процедура DLLEntry. Получает три параметра - идентификатор, причину вызова
; и зарезервированный параметр. Нам не нужен ни один из них.
_start@12:
        mov     al,1    ; Надо вернуть ненулевое число в EAX.
        ret     12

; Процедура BYTE WINAPI koi2win (BYTE symbol) - точка входа для вызова из C.
ifdef _MASM_
_koi2win@4      proc
else
koi2win         proc
endif
        pop     ecx     ; Обратный адрес в ECX.
        pop     eax     ; Параметр в ECX (теперь стек очищен
                        ; от параметров!).
        push    ecx     ; Обратный адрес вернуть в стек для RET.
; Здесь нет команды RET - управление передается следующей процедуре.
ifdef _MASM_
_koi2win@4      endp
else
 koi2win        endp
endif
; Процедура koi2win_asm.
; Точка входа для вызова из ассемблерных программ:
; ввод: AL - код символа в KOI,
; вывод: AL - код этого же символа в WIN.
ifdef _MASM_
_koi2win_asm@0  proc
else
 koi2win_asm    proc
endif
```

```
        test    al,80h          ; Если символ меньше 80h (старший бит 0),
        jz      dont_decode     ; не перекодировать,
        push    ebx             ; иначе -
        mov     ebx,offset k2w_tbl
        sub     al,80h          ; вычесть 80h,
        xlat                    ; перекодировать
        add     al,80h          ; и прибавить 80h.
        pop     ebx
dont_decode:
        ret                     ; Выйти.
ifdef _MASM_
_koi2win_asm@0  endp
else
 koi2win_asm    endp
endif

; VOID koi2wins(BYTE*koistring) -
; точка входа для вызова из C.
ifdef _MASM_
_koi2wins@4     proc
else
 koi2wins       proc
endif
        pop     ecx             ; Адрес возврата из стека.
        pop     eax             ; Параметр в EAX.
        push    ecx             ; Адрес возврата в стек.
ifdef _MASM_
_koi2wins@4     endp
else
 koi2wins       endp
endif
; Точка входа для вызова из ассемблера:
; ввод: EAX - адрес строки, которую надо преобразовать из KOI в WIN.
;
ifdef _MASM_
_koi2wins_asm@0 proc
else
 koi2wins_asm   proc
endif
        push    esi             ; Сохранить регистры, которые
                                ; нельзя изменять.
        push    edi
        push    ebx
        mov     esi,eax         ; Приемник строк
        mov     edi,eax         ; и источник совпадают.
        mov     ebx,offset k2w_tbl
decode_string:
        lodsb                   ; Прочитать байт,
        test    al,80h          ; если старший бит 0,
```

```
        jz      dont_decode2    ; не перекодировать,
        sub     al,80h          ; иначе - вычесть 80h,
        xlat                    ; перекодировать
        add     al,80h          ; и добавить 80h.
dont_decode2:
        stosb                   ; Вернуть байт на место,
        test    al,al           ; если байт - не ноль,
        jnz     decode_string   ; продолжить.
        pop     ebx
        pop     edi
        pop     esi
        ret
ifdef _MASM_
_koi2wins_asm@0  endp
else
 koi2wins_asm    endp
endif
                end     _start@12
```

Как видно из примера, нам пришлось назвать все процедуры по-разному для различных ассемблеров. В случае MASM понятно, что все функции должны иметь имена типа _start@12, а иначе программам, использующим их, придется обращаться к функциям с именами типа___imp_start, то есть такой DLL-файл нельзя будет загружать из программы, написанной на Microsoft C. В случае TASM и WASM процедуры могут иметь неискаженные имена (и более того, wlink.exe не позволяет экспортировать имя переменной, содержащее символ @), так как их компиляторы берут имена процедур не из библиотечного файла, а прямо из DLL посредством соответствующей программы - implib или wlib.

Итак, чтобы воспользоваться полученным DLL-файлом, напишем простую программу, которая перекодирует одну строку из KOI-8r в Windows cp1251.

```
; dlldemo.asm
; Графическое приложение для Win32, демонстрирующее работу с dllrus.dll,
; выводит строку в KOI8 и затем в cp1251, перекодированную функцией koi2wins.
;
; Компиляция:
; MASM
;  ml /c /coff /Cp /D_MASM_ dlldemo.asm
;  link dlldemo.obj /subsystem:windows
; (должен присутствовать файл dllrus.lib, созданный ори компиляции dllrus.dll)
;
; TASM
;  tasm /m /ml /D_TASM_ dlldemo.asm
;  implib dllrus.lib dllrus.dll
;  tlink32 /Tpe /aa /x /c dlldemo.obj
;
; WASM
;  wasm  dlldemo.asm
```

```
;  wlib dllrus.lib dllrus.dll
;  wlink file dlldemo.obj form windows nt
;
include def32.inc
include user32.inc
include kernel32.inc

includelib dllrus.lib
ifnde'f _MASM_
                extrn    koi2win_asm:near          Определения для функций
                                                  ;из DLL для
                extrn    koi2win:near               TASM и WASM
                extrn    koi2wins_asm:near          (хотя для WASM было бы
                                                  ;эффективнее
                extrn    koi2wins:near              использовать___imp__koi2win,
выделив
else                                              ; его в условный блок).
                extrn_____imp__koi2win_asm@0:dword ; А это для MASM.
                extrn_____imp__koi2win@4:dword
                extrn_____imp__koi2wins_asm@0:dword
                extrn_____imp__koi2wins@4:dword
 koi2win_asm    equ_______imp__koi2win_asm@0
 koi2win        equ_______imp__koi2win@4
 koi2wins_asm   equ_______imp__koi2wins_asm@0
 koi2wins       equ_______imp__koi2wins@4
endif
        .386
        .model flat
        .const
title_string1   db      "koi2win demo:  string in KOI8",0
title_string2   db      "koi2win demo:  string in cp1251",0
        .data
koi_string      db    0F3h, 0D4h, 0D2h, 0CFh, 0CBh, 0C1h, 20h, 0CEh, 0C1h
                db      20h, 0EBh, 0EFh, 0E9h, 2Dh, 28h, 0
        .code
_start:
        push    MB_OK
        push    offset  title_string1      ;  Заголовок окна MessageBox.
        push    offset  koi_string         ;  Строка на KOI.
        push    0
        call    MessageBox
        mov     eax, offset koi_string
        push    eax
        call    koi2wins
        push    MB_OK
        push    offset title_string2
        push    offset koi_string
        push    0
        call    MessageBox
```

```
        push    0                       ; Код выхода.
        call    ExitProcess             ; Конец программы.
end     _start
```

Этот небольшой DLL-файл может оказаться очень полезным для расшифровки текстов, взятых из Internet или других систем, где применяется кодировка KOI8. Воспользовавшись таблицами из приложения 1, вы можете расширить набор функций dllrus.dll, вплоть до перекодировки в какой угодно вариант.

7.5. Драйверы устройств

В Windows, так же как и в DOS, существует еще один вид исполняемых файлов - драйверы устройств. Windows 3.xx и Windows 95 используют одну модель драйверов, Windows NT - другую, а Windows 98 - уже третью, хотя и во многом близкую к модели Windows NT. В Windows 3.xx/95 применяются два типа драйверов устройств - виртуальные драйверы (VxD), выполняющиеся с уровнем привилегий 0 (обычно имеют расширение .386 для Windows 3.xx и .VXD для Windows 95), и непривилегированные драйверы, исполняющиеся, как и обычные программы, с уровнем привилегий 3 (как правило, они имеют расширение .DRV). Windows NT использует несовместимую модель драйверов, так называемую kernel-mode (режим ядра). На основе модели kernel-mode с добавлением поддержки технологии PNP И понятия потоков данных в 1996 году была создана модель WDM (win32 driver model), которая теперь используется в Windows 98/NT и, по-видимому, будет играть главную роль в дальнейшем.

Как и следовало ожидать, основным средством создания драйверов является ассемблер, и хотя использование C здесь возможно (в отличие от драйверов DOS), но сделать это сложнее: функции, к которым обращается драйвер, могут передавать параметры в регистрах; сегменты, из которых состоит драйвер, должны называться определенным образом и т. д. При создании драйверов часто пользуются одновременно C и ассемблером или C и специальным пакетом, который включает в себя все необходимые для инициализации средства.

Чтобы самостоятельно создавать драйверы для любой версии Windows, необходим комплект программ, документации, включаемых файлов и библиотек, распространяемый Microsoft, который называется DDK - Drivers Development Kit. (DDK для Windows NT/98 распространяются бесплатно.)

Мы не будем рассматривать программирование драйверов для Windows в деталях, так как этой теме посвящено много литературы, не говоря уже о документации, прилагающейся к DDK. Чтобы создать драйвер, в любом случае лучше всего начать с одного из прилагающихся примеров и изменять/добавлять процедуры инициализации, обработчики сообщений, прерываний и исключений, обработчики для API, предоставляемого драйвером, и т. д. Рассмотрим, как выглядит исходный текст драйвера, потому что он несколько отличается от привычных нам ассемблерных программ.

Любой драйвер начинается с директивы include vmm.inc, которая включает файл, содержащий определения используемых сегментов и макроопределений.

Макроопределения вида VXD_LOCKED_CODE_SEG/VXD_LOCKED_CODE_ENDS соответствуют директивам начала и конца сегментов (в данном случае сегмента _LTEXT). Другие важные макроопределения - Declare_Virtual_Device и VMMCall/WDMCall. Первое - просто определение, получающее в качестве параметров идентификатор драйвера, название, версию, порядок загрузки и адреса основных процедур драйвера, из которых строится его заголовок. Второе - замена команды call, получающая в качестве параметра имя функции VMM или WDM, к которой надо обратиться. Например, элемент кода драйвера BIOSXLAT, перехватывающий прерывание 10h, выглядит следующим образом:

```
VxD_ICODE_SEG                                   Начало сегмента _ITEXT (сегмент кода
                                              ; инициализации, исполняющийся
                                              ; в защищенном режиме, который удаляется
                                              ; из памяти после сообщения Init_Complete).
BeginProc  BIOSXlat_Sys_Critical_Init         ; Процедура, вызываемая для обработчика
                                              ; сообщения Sys_Critical_Init - первого
                                              ; сообщения, которое получает драйвер.
                                              ; Обычно обработчики сообщений должны
                                              ; сохранять регистры EBX, EDI, ESI и EBP,
                                              ; хотя в данном случае этого можно
                                              ; не делать.
        mov     esi,OFFSET32 BIOSXlat_Int10   ; Адрес обработчика INT 10h
                                              ; в регистр ESI. Важно использовать
                                              ; макроопределение OFFSET32 всюду
                                              ; вместо offset.
        mov     edx, 10h                      ; Любое число, которое будет
                                              ; помещаться в EDX при вызове
                                              ; регистрируемого обработчика.
        VMMCall Allocate_PM_Call_Back         ; Зарегистрировать точку входа,
                                              ; обращаясь к которой, программы
; из защищенного режима в VM будут передавать управление процедуре в драйвере (но не
; win32-программа) - они должны использовать DeviceIOControl для работы с драйверами.
        jc      BXLSCI_NoPM                 ; Если CF = 1 - произошла ошибка:
        xchg    edx,eax                     ; точку входа - в EDX, число 10h - в EAX
; (теперь это номер перехватываемого прерывания для Set_PM_Int_Vector).
        mov     ecx,edx
        shr     ecx,10h                       ; Селектор точки входа.
        movzx   edx,dx                        ; Смещение точки входа.
        VMMCall Set_PM_Int_Vector             ; Установить обработчик прерывания INT 10h.
; Если эта функция вызвана до Sys_VM_Init, установленный обработчик
; становится звеном цепочки обработчиков для всех виртуальных машин.
; После того как прерывание проходит по цепочке обработчиков
; в защищенном режиме, оно отображается в V86 точно так же, как в DPMI
[код перехвата других прерываний].
EndProc BIOSXlat_Sys_Critical_Init          ; Конец процедуры.
VxD_ICODE_ENDS                              ; Конец сегмента инициализации.
```

Соответственно, чтобы эта процедура была вызвана для сообщения Sys_Critical_Init, в сегменте фиксированного кода _LTEXT должна быть следующая запись:

```
VxD_LOCKED_CODE_SEG   .                    ; Начало сегмента _LTEXT.
BeginProc        BIOSXlat_Control          ; Начало процедуры.
                 Control_Dispatch Sys_Critical_Init, BIOSXlat_Sys_Critical_Init
; При помощи еще одного макроопределения из vmm.inc зарегистрировать процедуру
; BIOSXlat_Sys_Critical_Init как обработчик сообщения Sys_Critical_Init.
        clc                                ; Процедура-обработчик управляющих
                                           ; сообщений должна возвращать CF = 0.
        ret
EndProc          BIOSXlat_Control          ; Конец процедуры.
VxD_LOCKED_CODE_ENDS                       ; Конец сегмента _LTEXT.
```

И наконец, процедура BIOSXlat_Control регистрируется в заголовке драйвера как процедура, получающая управляющие сообщения:

```
; Первая строка после .386p и include vmm.inc:
Declare_Virtual_Device    BIOSXlat, 1, 0,  BIOSXlat_Control, BIOSXlat_Device_ID,
                          BIOSXlat_Init_Order
```

Это не слишком сложно и, пользуясь примерами и документацией из DDK, а также отладчиком SoftICE, можно справиться практически с любой задачей, для которой есть смысл создавать драйвер.

Глава 8. Ассемблер и языки высокого уровня

В предыдущей главе, занимаясь программированием для Windows, мы уже обращались к процедурам, написанным на языке высокого уровня из программ на ассемблере, а также создавали процедуры на ассемблере, к которым можно обращаться из языков высокого уровня. Для этого нужно было соблюдать определенные договоренности о передаче параметров - параметры помещались в стек справа налево, результат возвращался в EAX, стек освобождался от переданных параметров самой процедурой. Данная договоренность, известная как STDCALL, конечно, не единственная, и разные языки высокого уровня используют различные способы передачи параметров.

8.1. Передача параметров

Большинство языков высокого уровня передают параметры вызываемой процедуре в стеке и ожидают возвращения параметров в регистре AX (EAX). Иногда используется DX:AX (EDX:EAX), если результат не умещается в одном регистре, и ST(0), если результат - число с плавающей запятой.

8.1.1. Конвенция Pascal

Самый очевидный способ выражения вызова процедуры или функции языка высокого уровня, после того как решено, что параметры передаются в стеке и возвращаются в регистре AX/EAX, – это способ, принятый в языке PASCAL (а также в BASIC, FORTRAN, ADA, OBERON, MODULA2), - просто поместить параметры в стек в естественном порядке:

запись

```
some_proc(a,b,c,d,e)
```

превращается в:

```
push    a
push    b
push    c
push    d
push    e
call    some_proc
```

Это значит, что процедура some_proc, во-первых, должна очистить стек по окончании работы (например, завершившись командой ret 10) и, во-вторых, параметры, переданные ей, находятся в стеке в обратном порядке:

```
some_proc       proc
        push    bp
        mov     bp, sp    ; Создать стековый кадр.
a               equ     [bp+12] ; Определения для простого доступа к параметрам.
b               equ     [bp+10]
c               equ     [bp+8]
d               equ     [bp+6]
e               equ     [bp+4]

[текст процедуры, использующей параметры a, b, c, d, e]

        pop     bp
        ret     10
some_proc       endp
```

Этот код в точности соответствует усложненной форме директивы proc, которую поддерживают все современные ассемблеры:

```
some_proc       proc    PASCAL,a:word,b:word,c:word,d:word,e:word

[текст процедуры, с параметрами a, b, c, d, e. Так как BP применяется в качестве указателя стекового кадра, его использовать нельзя!]

        ret             ; Эта команда RET будет заменена на RET 10.
some_proc       endp
```

Главный недостаток этого подхода заключается в сложности создания функции с изменяемым числом параметров, аналогичных printf - функции языка C. Чтобы определить число параметров, переданных printf, процедура должна сначала прочитать первый параметр, но она не знает его расположения в стеке. Эту проблему решает подход, используемый в C, где параметры передаются в обратном порядке.

8.1.2. Конвенция C

Данный способ передачи параметров используется в первую очередь в языках C и C++, а также в PROLOG и др. Параметры помещаются в стек в обратном порядке, а удаление параметров из стека (в противоположность PASCAL-конвенции) выполняет вызывающая процедура:

запись

```
some_proc(a,b,c,d,e)
```

превращается в:

```
        push    e
        push    d
        push    c
        push    b
        push    a
        call    some_proc
        add     sp, 10    ; Освободить стек.
```

Вызванная процедура может инициализироваться следующим образом:

```
some_proc       proc
        push    bp
        mov     bp,sp    ; Создать стековый кадр.
a               equ     [bp+4]  ; Определения для простого доступа к параметрам.
b               equ     [bp+6]
c               equ     [bp+8]
d               equ     [bp+10]
e               equ     [bp+12]

[текст процедуры, использующей параметры a, b, c, d, e]

        pop     bp
        ret
some_proc       endp
```

Ассемблеры поддерживают и такой формат вызова посредством усложненной формы директивы proc с указанием языка C:

```
some_proc       proc    C,a:word,b:word,c:word,d:word,e:word

[текст процедуры с параметрами a, b, c, d, e. Так как BP применяется как указатель стекового кадра, его использовать нельзя!]

        ret
some_proc       endp
```

До сих пор этими формами записи процедур в ассемблере мы не пользовались потому, что они скрывают от нас следующий факт: регистр BP служит для хранения параметров и его ни в коем случае нельзя изменять, а в случае PASCAL команда ret на самом деле - ret N.

Преимущество по сравнению *с* PASCAL-конвенцией заключается в том, что освобождение стека от параметров в C возлагается на вызывающую процедуру, а это позволяет лучше оптимизировать код программы. Например, если мы должны вызвать несколько функций, принимающих одни и те же параметры подряд, можно не заполнять стек каждый раз заново:

```
        push    param2
        push    param1
        call    prod
        call    proc2
        add     sp,4
```

эквивалентно

```
proc1(param1,param2);
proc2(param1,param2);
```

вот почему компиляторы с языка C создают более компактный и быстрый код нежели компиляторы с других языков.

8.1.3. Смешанные конвенции

В главе 7 мы познакомились с договоренностью о передаче параметров STDCALL, отличавшейся и от C, и от PASCAL-конвенций, которая применяется для всех системных функций Win32 API. Здесь параметры помещаются в стек в обратном порядке, как в C, но процедуры должны очищать стек сами, как в PASCAL.

Еще одно интересное отклонение от C-конвенции можно наблюдать в Watcom C. Этот компилятор активно использует регистры для ускорения работы программы, и параметры в функции также передаются по возможности через регистры. Например, при вызове функции с шестью параметрами

```
some_proc(a,b,c,d,e,f);
```

первые четыре передаются соответственно в (E)AX, (E)DX, (E)BX, (E)CX, а с пятого параметры помещаются в стек в обычном обратном порядке:

```
e               equ     [bp+4]
f               equ     [bp+6]
```

8.2. Искажение имен

Компиляторы Microsoft C (а также многие компиляторы в UNIX, как мы узнаем далее) изменяют названия процедур, чтобы отразить используемый способ передачи параметров. Так, к названиям всех процедур, применяющих C-конвенцию, добавляется символ подчеркивания. То есть, если в C-программе записано

```
        some_proc();
```

то реально компилятор пишет

```
        call    _some_proc
```

и это означает: если процедура написана на ассемблере, она должна называться именно _some_proc (или использовать сложную форму записи директивы proc).

Названия процедур, использующих STDCALL, как можно было видеть из примера DLL-программы в разделе 7.4, искажаются еще более сложным образом: спереди к названию процедуры добавляется символ подчеркивания, а сзади - символ @ и размер области стека в байтах, которую занимают параметры (то есть число, стоящее после команды ret в конце процедуры).

```
        some_proc(a:word);
```

превращается в

```
        push    a
        call    _some_proc@4
```

8.3. Встроенный ассемблер

Если требуется выполнить совсем небольшую операцию на ассемблере, например вызвать какое-то прерывание или преобразовать сложную битовую структуру, порой нерационально создавать отдельный файл ради нескольких строк на

ассемблере. Во избежание этого многие языки высокого уровня поддерживают возможность вставки ассемблерного кода непосредственно в программу. Например, напишем процедуру, возвращающую слово, находящееся по адресу 0040h:006Ch, в BIOS - счетчик сигналов системного таймера, который удобно использовать для инициализации генераторов случайных чисел.

8.3.1. Ассемблер, встроенный в Pascal

```
function get_seed:longint
var
 seed:longint
begin
        asm
                push    es
                mov     ax,0040h
                mov     es,ax
                mov     ax,es:[006Ch]
                mov     seed,ax
                pop     es
        end;
get_seed:=seed;
end;
```

8.3.2. Ассемблер, встроенный в C

```
int get_seed()
int seed;
{
        _asm {
                push    es
                mov     ax,0040h
                mov     es,ax
                mov     ax,es:[006Ch]
                mov     seed,ax
                pop     es
        };
return(seed);
};
```

В данных ситуациях ассемблерная программа может свободно пользоваться переменными из языка высокого уровня, так как они автоматически преобразуются в соответствующие выражения типа word ptr [bp + 4].

Глава 9. Оптимизации

Наиболее популярным применением ассемблера обычно считается именно оптимизация программ, то есть уменьшение времени выполнения программ по сравнению с языками высокого уровня. Но если просто переписать текст, например с языка С на ассемблер, переводя каждую команду наиболее очевидным способом, окажется, что С-процедура выполнялась быстрее. Вообще говоря, ассемблер, как и любой другой язык, сам по себе не является панацеей от неэффективного программирования - чтобы действительно оптимизировать программу, требуется знать не только команды процессора, но и алгоритмы, оптимальные способы их реализации и владеть подробной информацей об архитектуре процессора.

Проблему оптимизации принято делить на три основных уровня:

1. Выбор самого оптимального алгоритма - высокоуровневая оптимизация.
2. Наиболее оптимальная реализация алгоритма - оптимизация на среднем уровне.
3. Подсчет тактов, тратящихся на выполнение каждой команды, и оптимизация их порядка для конкретного процессора — низкоуровневая оптимизация.

9.1. Высокоуровневая оптимизация

Выбор оптимального алгоритма для решения задачи всегда приводит к лучшим результатам, чем любой другой вид оптимизации. Действительно, при замене пузырьковой сортировки, время выполнения которой пропорционально N^2, на быструю сортировку, выполняющуюся как N X log(N), всегда найдется такое число сортируемых элементов N, что вторая программа будет выполняться быстрее, как бы она ни была реализована. Поиск лучшего алгоритма - универсальная стадия, и она относится не только к ассемблеру, но и к любому языку программирования, поэтому будем считать, что оптимальный алгоритм уже выбран.

9.2. Оптимизация на среднем уровне

Реализация алгоритма на данном конкретном языке программирования - самая ответственная стадия оптимизации. Именно здесь можно получить выигрыш в скорости в десятки раз или сделать программу в десятки раз медленнее при серьезных ошибках в реализации. Многие элементы, из которых складывается оптимизация, уже упоминались - хранение переменных, с которыми выполняется активная работа, в регистрах, использование таблиц переходов вместо длинных последовательностей проверок и условных переходов и т. п. Тем не менее даже плохо реализованные операции не вносят заметных замедлений в программу, если они не повторяются в цикле. Можно говорить, что практически все проблемы

оптимизации на среднем уровне так или иначе связаны с циклами, и именно поэтому мы рассмотрим основные правила, которые стоит иметь в виду при реализации любого алгоритма, содержащего циклы.

9.2.7. *Вычисление констант вне цикла*

Самым очевидным и самым важным правилом при создании цикла на любом языке программирования является вынос всех переменных, которые не изменяются на протяжении цикла, за его пределы. Программируя на ассемблере, имеет смысл также по возможности разместить все переменные, которые будут использоваться внутри цикла, в регистры, а старые значения нужных после цикла регистров сохранить в стеке.

9.2.2. *Перенос проверки условия в конец цикла*

Циклы типа WHILE или FOR, которые так часто применяются в языках высокого уровня, оказываются менее эффективными по сравнению с циклами типа UNTIL из-за того, что в них требуется лишняя команда перехода:

```
; Цикл типа WHILE.
        mov     si,counter              ; Число повторов.
        mov     dx,start_i              ; Начальное значение.
loop_start:
        cmp     dx,si                   ; Пока dx < si - выполнять.
        jnb     exit_loop
        [тело цикла]
        inc     dx
        jmp     loop_start

; Почти такой же цикл типа UNTIL:
        mov     si,counter
        mov     dx,start_i
loop_start:                             ; Выполнять. -
        [тело цикла]
        inc     dx
        cmp     dx,si                   ; Пока dx < si.
        jb      loop_start
```

Естественно, цикл типа UNTIL, в отличие от цикла типа WHILE, выполнится по крайней мере один раз, так что, если это нежелательно, придется добавить одну проверку перед телом цикла, но в любом случае даже небольшое уменьшение тела цикла всегда оказывается необходимой операцией.

9.2.3. *Выполнение цикла задом наперед*

Циклы, в которых значение счетчика растет от единицы или нуля до некоторой константы, можно реализовать вообще без операции сравнения, выполняя цикл в обратном направлении (и мы пользовались этим приемом неоднократно в наших примерах). Дело в том, что команда DEC counter устанавливает флаги почти так же, а команда SUB counter,1 - абсолютно так же, как и команда CMP

counter,1, то есть следующая команда условного перехода будет обрабатывать результат сравнения счетчика с единицей:

```
; Цикл от 10 до 1.
        mov     dx, 10
loop_start:
        [тело цикла]
        dec     dx              ; Уменьшить DX,
        jnz     loop_start      ; если DX не стал нулем - продолжить цикл.

; Цикл от 10 до 0.
        mov     dx, 10
loop_start:
        [тело цикла]
        dec     dx              ; Уменьшить DX,
        jns     loop_start      ; если DX не отрицательный - продолжить цикл.
```

Конечно, не все циклы можно заставить выполняться в обратном направлении сразу. Например, иногда приходится изменять формат хранения массива данных также на обратный или вносить другие изменения, но в целом, если это возможно, всегда следует стремиться к циклам, выполняющимся задом наперед. Кроме того, если цикл построен по этому образцу, выполняется до значения счетчика, равного нулю, и регистр CX можно освободить для выполнения роли счетчика, есть вариант воспользоваться командой LOOP, хотя в некоторых случаях в низкоуровневой оптимизации команды DEC/JNZ оказываются более эффективными.

9.2.4. *Разворачивание циклов*

Для небольших циклов время выполнения проверки условия и перехода на начало цикла может оказаться значительным по сравнению с временем выполнения самого тела цикла. Более того, Pentium Pro/Pentium II всегда тратят по крайней мере один такт процессора на цикл, хотя его тело может выполняться даже быстрее одного такта. С этим легко справиться, вообще не создавая цикл, а просто повторив его тело нужное число раз (разумеется, только в случае, если нам заранее известно это число!). Для очень коротких циклов можно, например, удваивать или утраивать тело цикла при условии, что число повторений кратно двум или трем. Кроме того, бывает удобно часть работы сделать в цикле, а часть развернуть, например продолжая цепочку циклов из предыдущего примера:

```
; Цикл от 10 до -1.
        mov     dx, 10
loop_start:
        [тело цикла]
        dec     dx              ; Уменьшить DX.
        jns     loop_start      ; Если DX не отрицательный -
                                ; продолжить цикл.
        [тело цикла]
```

Совершенно естественно, что эти простые методики не перечисляют Все возможности оптимизации среднего уровня, более того, они не описывают и десятой доли всех ее возможностей. Умение оптимизировать программы нельзя сформулировать в виде набора простых алгоритмов - слишком много таких ситуаций, когда любой алгоритм оказывается неоптимальным. При решении задачи оптимизации приходится постоянно что-то изменять. Именно потому, что оптимизация всегда занимает очень много времени, рекомендуется приступать к ней только после написания программы. Как и во многих других случаях, на любой стадии создания программы с оптимизацией нельзя торопиться, но и нельзя совсем забывать о ней.

9.3. Низкоуровневая оптимизация

9.3.7. *Общие принципы низкоуровневой оптимизации*

Так как процессоры Intel используют весьма сложный набор команд, большинство операций можно выполнить на низком уровне различными способами. При этом иногда оказывается, что наиболее очевидный способ - не самый быстрый или короткий. Часто простыми перестановками команд, зная механизм их реализации на современных процессорах, можно заставить ту же процедуру выполняться на 50-200% быстрее. Разумеется, переходить к этому уровню оптимизации разрешается только после того, как текст программы окончательно написан и максимально оптимизирован на среднем уровне.

Перечислим основные рекомендации, которым нужно следовать при оптимальном программировании для процессоров Intel Pentium, Pentium MMX, Pentium Pro и Pentium II.

Основные рекомендации

Используйте регистр EAX всюду, где возможно. Команды с непосредственным операндом, с операндом - абсолютным адресом переменной и команды XCHG с регистрами на байт меньше, если другой операнд - регистр EAX.

Применяйте регистр DS всюду, где возможно. Префиксы переопределения сегмента увеличивают размер программы на 1 байт и время на 1 такт.

Если к переменной в памяти, адресуемой со смещением, выполняется несколько обращений - загрузите ее в регистр.

Не используйте сложные команды (ENTER, LEAVE, LOOP, строковые команды), если аналогичное действие можно выполнить небольшой последовательностью простых команд.

Не используйте команду MOVZX для чтения байта - это требует четыре такта. Заменой может служить следующая пара команд:

```
xor     eax,eax
mov     al,source
```

Применяйте TEST для сравнения с нулем:

```
test    eax,eax
jz      if_zero          ; Переход, если EAX = 0.
```

Применяйте команду XOR, чтобы обнулять регистр (конечно, если текущее состояние флагов больше не потребуется); она официально поддерживается Intel как команда обнуления регистра:

```
        xor     eax,eax          ; EAX = 0
```

Не используйте умножение или деление на константу - его можно заменить другими командами, например:

```
; EAX = EAX * 10
        shl     eax,1            ; Умножение на 2.
        lea     eax,[eax+eax*4]  ; Умножение на 5.
; EAX = EAX * 7
        mov     ebx,eax
        shl     eax,3            ; Умножение на 8
        sub     eax,ebx          ; и вычитание сохраненного EAX.
; AX = AX/10
        mov     dx,6554          ; DX = 65 536/10.
        mul     dx               ; DX = AX/10 (умножение выполняется
                                 ; быстрее деления).
; EAX = EAX mod 64 (остаток от деления на степень двойки).
        and     eax,3Fh
```

Используйте короткую форму команды jmp, где возможно (jmp short метка).

Как можно реже загружайте сегментные регистры.

Как можно меньше переключайте задачи - это очень медленная процедура. Часто, если сохранение состояния процесса не требует больших затрат, например для реализации нитей, переключение быстрее организовать с помощью программы.

Команда LEA

LEA можно использовать (кроме прямого назначения - вычисления адреса сложно адресуемой переменной) для следующих двух ситуаций:

- быстрое умножение

```
        lea     eax,[eax*2]      ; EAX = EAX x 2 (shl eax,1 лучше).
        lea     eax,[eax+eax*2]  ; EAX = EAX x 3.
        lea     eax,[eax*4]      ; EAX = EAX x 4 (shl eax,2 лучше).
        lea     eax,[eax+eax*4]  ; EAX = EAX x 5.
        lea     eax,[eax+eax*8]  ; EAX = EAX x 9.
```

а трехоперандное сложение

```
        lea  ecx,[eax+ebx]       ; ECX = EAX + EBX.
```

Единственный недостаток LEA - увеличивается вероятность AGI с предыдущей командой (см. ниже).

Выравнивание

8-байтные данные должны быть выравнены по 8-байтным границам (то есть три младших бита адреса должны быть равны нулю).

4-байтные данные должны быть выравнены по границе двойного слова (то есть два младших бита адреса должны быть равны нулю).

2-байтные данные должны полностью содержаться в выравненном двойном слове (то есть два младших бита адреса не должны быть равны единице).

80-битные данные должны быть выравнены по 16-байтным границам.

Когда нарушается выравнивание при доступе к данным, находящимся в кэше, теряются 3 такта на каждое невыравненное обращение на Pentium и 9–12 тактов - на Pentium Pro/Pentium II.

Так как линейка кэша кода составляет 32 байта, метки для переходов, особенно метки, отмечающие начало цикла, должны быть выравнены по 16-байтным границам, а массивы данных, равные или большие 32 байт, должны начинаться с адреса, кратного 32.

Генерация адреса

AGI - это ситуация, при которой регистр, используемый командой для генерации адреса как базовый или индексный, являлся приемником предыдущей команды. В таком случае процессор тратит один дополнительный такт.
Последовательность команд

```
add     edx,4
mov     esl,[edx]
```

выполняется с AGI на любом процессоре.

Последовательность команд

```
add     esi,4           ; U-конвейер - 1 такт (на Pentium).
pop     ebx             ; V-конвейер - 1 такт.
inc     ebx             ; V-конвейер - 1 такт
mov     edi,[esi]       ; в U-конвейер - «AGI», затем 1 такт.
```

выполняется с AGI на Pentium за три такта процессора.

Кроме того, AGI может происходить неявно, например при изменении регистра ESP и обращении к стеку:

```
        sub     esp,24
        push    ebx             ; *AGI*
```

или

```
        mov     esp,ebp
        pop     ebp             ; *AGI*
```

но изменение ESP, производимое командами PUSH и POP, не приводит к AGI, если следующая команда тоже обращается к стеку.

Процессоры Pentium Pro и Pentium II не подвержены AGI.

Обращение к частичному регистру

Если команда обращается к 32-битному регистру, например EAX, сразу после команды, выполнявшей запись в соответствующий частичный регистр (AX, AL, AH), происходит пауза минимум в семь тактов на Pentium Pro и Pentium II и в один такт на 80486, но не на Pentium:

```
mov     ax,8
add     ecx,eax         ; Пауза.
```

На Pentium Pro и Pentium II эта пауза не появляется, если сразу перед командой записи в AX была команда XOR EAX,EAX или SUB EAX,EAX.

Префиксы

Префиксы LOCK, переопределения сегмента и изменения адреса операнда увеличивают время выполнения команды на 1 такт.

9.3.2. *Особенности архитектуры процессоров Pentium и Pentium MMX*

Выполнение команд

Процессор Pentium содержит два конвейера исполнения целочисленных команд (U и V) и один конвейер для команд FPU. Он может выполнять две целочисленные команды одновременно и поддерживает механизм предсказания переходов, значительно сокращающий частоту сброса очереди предвыборки из-за передачи управления по другому адресу.

На стадии загрузки команды процессор анализирует сразу две следующие команды, находящиеся в очереди, и, если возможно, выполняет одну из них в U-конвейере, а другую в V. Если это невозможно, первая команда загружается в U-конвейер, а V-конвейер пустует.

V-конвейер имеет определенные ограничения на виды команд, которые могут в нем исполняться. Приложение 2 содержит для каждой команды информацию о том, может ли она выполняться одновременно с другими командами и в каком конвейере. Кроме того, две команды не будут запущены одновременно, если:

- команды подвержены одной из следующих регистровых зависимостей:
  - первая команда пишет в регистр, а вторая читает из него;
  - обе команды пишут в один и тот же регистр (кроме записи в EFLAGS).

  Исключения из этих правил - пары PUSH/PUSH, PUSH/POP и PUSH/CALL, осуществляющие запись в регистр ESP;
- одна из команд не находится в кэше команд (кроме случая, если первая команда - однобайтная);
- одна из команд длиннее семи байт (для Pentium);
- одна команда длиннее восьми байт, а другая - семи (для Pentium MMX).

Помните, что простыми перестановками команд можно выиграть до 200% скорости в критических ситуациях.

Кэш-память

Процессор Pentium состоит из двух 8-килобайтных блоков кэш-памяти, один для кода и один для данных с длиной линейки 32 байта. Кэш данных состоит из восьми банков, причем он доступен из обоих конвейеров одновременно, только если обращения происходят к разным банкам. Когда данные или код находятся в не кэше, минимальная дополнительная задержка составляет 4 такта.

Если происходит два кэш-промаха Одновременно при записи в память из обоих конвейеров и обе записи попадают в одно учетверенное слово (например, при записи двух слов в последовательные адреса), процессор затрачивает столько же времени, сколько и на один кэш-промах.

Очередь предвыборки

Перед тем как команды распределяются по конвейерам, они загружаются из памяти в одну из четырех 32-байтных очередей предвыборки. Если загружается команда перехода и блок предсказания переходов подтверждает, что переход произойдет, начинает работать следующая очередь предвыборки. Это сделано для того, чтобы, если предсказание было неверным, первая очередь продолжила выборку команд после невыполненной команды перехода.

Если условный переход не был предугадан, затрачивается 3 такта, когда команда перехода находилась в U-конвейере, и 4 такта, когда в V.

Если безусловный переход или вызов процедуры не был предугадан, затрачивается 3 такта в любом случае;

Конвейер FPU

Конвейер исполнения команд FPU состоит из трех участков, на каждом из которых команда тратит по крайней мере один такт. Многие команды, однако, построены таким образом, что позволяют другим командам выполняться на ранних участках конвейера, пока данные команды выполняются на более поздних. Кроме того, параллельно с длинными командами FPU, например FDIV, могут выполняться команды в целочисленных конвейерах.

Команда FXCH может выполняться одновременно почти с любой командой FPU, что позволяет использовать ST(n) как неупорядоченный набор регистров практически без потерь в производительности.

Конвейер MMX

Команды MMX, так же как команды FPU, используют дополнительный конвейер, содержащий два блока целочисленной арифметики (и логики), один блок умножения, блок сдвигов, блок доступа к памяти и блок доступа к целочисленным регистрам. Все блоки, кроме умножителя, выполняют свои стадии команды за один такт, умножение требует трех тактов, но имеет собственный буфбр, позволяющий принимать по одной команде каждый такт. Так как блоков арифметики два, соответствующие операции могут выполняться одновременно в U- или V-конвейере. Команды, использующие блок сдвигов или умножитель, способны осуществляться в любом конвейере, но не одновременно с другими командами, применяющими тот же самый блок. А команды, обращающиеся к памяти или обычным регистрам, в состоянии выполняться только в U-конвейере и только одновременно с MMX-командами.

Если перед командой, копирующей MMX-регистр в память или в обычный регистр, происходила запись в MMX-регистр, затрачивается один лишний такт.

9.3.3. Особенности архитектуры процессоров Pentium Pro и Pentium II

Процессоры Pentium Pro и Pentium II включают в себя целый набор средств для ускорения выполнения программ. В них происходит выполнение команд не по порядку, предсказание команд и переходов, аппаратное переименование регистров.

Выполнение команд

За каждый такт процессора из очереди предвыборки может быть прочитано и декодировано на микрооперации до трех команд. В этот момент работают три декодера, первый из которых декодирует команды, содержащие до четырех микроопераций, а другие два - только команды из одной микрооперации. Если в ассемблерной программе команды упорядочены в соответствии с этим правилом (4-1-1), то на каждый такт будет происходить декодирование трех команд. Например: если в последовательности команд

```
add     eax,[ebx]       ; 2m - в декодер 0 на первом такте.
mov     ecx,[eax]       ; 2т - пауза 1 такт, пока декодер 0
                        ; не освободится.
add     edx,8           ; 1m - декодер 1 на втором такте.
```

переставить вторую и третью команды, то add edx,8 будет декодирована в тот же такт, что и первая команда.

Число микроопераций для каждой команды приведено в приложении 2, но можно сказать, что команды, работающие только с регистрами, как правило, выполняются за одну микрооперацию, команды чтения из памяти - тоже за одну, команды записи в память — за две, а команды, выполняющие чтение-изменение-запись, - за четыре. Сложные команды содержат больше четырех микроопераций и требуют несколько тактов для декодирования. Кроме того, команды длиннее семи байт не могут быть декодированы за один такт. В среднем время ожидания в этом буфере составляет около трех тактов.

Затем микрооперации поступают в буфер накопления, где они ждут, пока все необходимые им данные не будут доступны. Далее они посылаются в ядро системы неупорядоченного исполнения, состоящей из пяти конвейеров, каждый из которых обслуживает несколько блоков исполнения. Если все данные для микрооперации готовы и в ядре есть свободный элемент, исполняющий конкретную микрооперацию, в буфере накопления не будет потрачено ни одного лишнего такта. После выполнения микрооперации скапливаются в буфере завершения, где результаты их записываются, операции записи в память упорядочиваются и микрооперации завершаются (три за один такт).

Время выполнения команд в пяти конвейерах исполнения приведено в табл. 21.

Указанное в таблице время требуется для выполнения микрооперации, а скорость демонстрирует, с какой частотой элемент может принимать микрооперации в собственный конвейер (1 - каждый такт, 2 - каждый второй такт). То есть, например, одиночная команда FADD выполняется за три такта, а три последовательные команды FADD - тоже за три такта.

Микрооперации чтения и записи, обращающиеся к одному и тому же адресу в памяти, выполняются за один такт.

Существует особая группа синхронизирующих команд, любая из которых начинает выполняться только после того, как завершаться все микрооперации, находящиеся в процессе выполнения. К таким командам относятся привилегированные команды WRMSR, INVD, INVLPG, WBINVD, LGDT, LLDT, LIDT, LTR,

Таблица 21. Конвейеры процессора Pentium Pro/Pentium II

	Время выполнения	Скорость
Конвейер 0		
Блок целочисленной арифметики	1	1
Блок команд LEA	1	1
Блок команд сдвига	1	1
Блок целочисленного умножения	4	1
Блок команд FADD	3	1
Блок команд FMUL	5	2
Блок команд FDIV	17 для 32-битных 36 для 64-битных 56 для 80-битных	17 36 56
Блок MMX-арифметики	1	1
Блок MMX-умножений	3	1
Конвейер 1		
Блок целочисленной арифметики	1	1
Блок MMX-арифметики	1	1
Блок MMX-сдвигов	1	1
Конвейер 2		
Блок чтения	3 при кэш-попадании	1
Конвейер 3		
Блок записи адреса	не меньше 3	1
Конвейер 4		
Блок записи данных	не меньше 1	1

RSM и MOV в управляющие и отладочные регистры, а также две непривилегированные команды - IRET и CPUID. Когда, например, измеряют скорость исполнения процедуры при помощи команды RDTSC (см. раздел 10.2), полезно выполнить одну из синхронизирующих команд, чтобы убедиться в том, что все измеряемые команды полностью завершились.

Кэш-память

Процессоры Pentium Pro включают в себя 8-килобайтный кэш L1 для данных и 8-килобайтный кэш L1 для кода, а процессоры Pentium II соответственно по 16 Кб, но не все кэш-промахи приводят к чтению из памяти: существует кэш второго уровня - L2, который маскирует промахи L1. Минимальная задержка при промахе в оба кэша составляет 10–14 тактов в зависимости от состояния цикла обновления памяти.

Микрооперации чтения и записи могут произойти одновременно, если они обращаются к разным банкам кэша L1.

Очередь предвыборки

Очередь предвыборки считывает прямую линию кода 16-байтными выровненными блоками. Это значит, что следует организовывать условные переходы так,

чтобы наиболее частым исходом было бы отсутствие перехода, и что полезно выравнивать команды на границы слова. Кроме того, желательно располагать редко используемый код в конце процедуры, чтобы он не считывался в очередь предвыборки впустую.

Предсказание переходов

Процессор поддерживает 512-байтный буфер выполненных переходов и их целей. Система предсказания может обнаруживать последовательность до четырех повторяющихся переходов, то есть четыре вложенных цикла будут иметь процент предсказания близкий к 100. Кроме того, дополнительный буфер адресов возврата позволяет правильно предсказывать циклы, из которых вызываются подпрограммы.

На неправильно предсказанный переход затрачивается как минимум девять тактов (в среднем от 10 до 15). На правильно предсказанный невыполняющийся переход не затрачивается никаких дополнительных тактов вообще. На правильно предсказанный выполняющийся переход затрачивается один дополнительный такт. Именно поэтому минимальное время исполнения цикла на Pentium Pro или Pentium II - два такта и, если цикл может выполняться быстрее, он должен быть развернут.

Если команда перехода не находится в буфере, система предсказания делает следующие предположения:

- безусловный переход предсказывается как происходящий, и на его выполнение затрачивается 5-6 тактов;
- условный переход назад предсказывается как происходящий, и на его выполнение также затрачивается 5-6 тактов;
- условный переход вперед предсказывается как непроисходящий. При этом команды, следующие за ним, предварительно загружаются и начинают исполняться, вот почему не размещайте данные сразу после команды перехода.

Глава 10. Процессоры Intel в защищенном режиме

Мы уже неоднократно сталкивались с защищенным режимом и даже программировали приложения, которые работали в нем (см. главы 6 и 7), при этом пользовались только средствами, которые предоставляла операционная система, и до сих пор не рассматривали, как процессор переходит и функционирует в защищенном режиме, то есть как работают современные операционные системы. Дело вот в чем: управление защищенным режимом в современных процессорах Intel - это самый сложный раздел программирования и самая сложная глава в этой книге. Но материал можно легко освоить, если рассматривать этот раздел шаг за шагом - отдельные механизмы работы процессора достаточно мало перекрываются друг с другом. Прежде чем рассматривать собственно программирование, познакомимся с регистрами и командами процессора, которые пока были от нас скрыты.

10.1. Регистры

Рассматривая регистры процессора в разделе 2.1, мы специально ничего не рассказали о регистрах, которые не используются в обычном программировании, в основном именно потому, что они управляют защищенным режимом.

10.1.1. Системные флаги

Регистр флагов EFLAGS - это 32-битный регистр, в то время как в разделе 2.1.4 рассмотрена только часть из младших 16 бит. Теперь мы можем обсудить все:

биты 31-22: нули
бит 21: флаг идентификации (ID)
бит 20: флаг ожидания виртуального прерывания (VIP)
бит 19: флаг виртуального прерывания (VIF)
бит 18: флаг контроля за выравниванием (AC)
бит 17: флаг режима V86 (VM)
бит 16: флаг продолжения задачи (RF)
бит 15: 0
бит 14: флаг вложенной задачи (NT)
биты 13-12: уровень привилегий ввода-вывода (IOPL)
бит 11: флаг переполнения (OF)
бит 10: флаг направления (DF)
бит 9: флаг разрешения прерываний (IF)

бит 8: флаг трассировки (TF)
биты 7-0: флаги состояния (SF, ZF, AF, PF, CF) были рассмотрены подробно раньше

Флаг TF: если он равен 1, перед выполнением каждой команды генерируется исключение #DB (INT 1).

Флаг IF: если он равен 0, процессор не реагирует ни на какие маскируемые аппаратные прерывания.

Флаг DF: если он равен 1, регистры EDI/ESI при выполнении команд строковой обработки уменьшаются, иначе - увеличиваются.

Поле IOPL: уровень привилегий ввода-вывода, с которым выполняется текущая программа или задача. Чтобы программа могла обратиться к порту ввода-вывода, ее текущий уровень привилегий (CPL) должен быть меньше или равен IOPL. Это поле можно модифицировать, только имея нулевой уровень привилегий.

Флаг NT: равен 1, если текущая задача является вложенной по отношению к какой-то другой - в обработчиках прерываний и исключений и вызванных командой call задачах. Флаг влияет на работу команды IRET.

Флаг RF: когда этот флаг равен 1, отладочные исключения временно запрещены. Он устанавливается командой IRETD из обработчика отладочного прерывания, чтобы #DB не произошло перед выполнением команды, которая его вызвала, еще раз. На флаг не влияют команды POPF, PUSHF и IRET.

Флаг VM: установка этого флага переводит процессор в режим V86 (виртуальный 8086).

Флаг AC: если установить этот флаг и флаг AM в регистре CR0, каждое обращение к памяти из программ, выполняющихся с CPL = 3, не выравненное на границу слова для слов и на границу двойного слова для двойных слов, будет вызывать исключение #AC.

Флаг VIF: это виртуальный образ флага IF (только для Pentium и выше).

Флаг VIP: этот флаг указывает процессору, что произошло аппаратное прерывание. Флаги VIF и VIP используются в многозадачных средах для того, чтобы каждая задача имела собственный виртуальный образ флага IF (только для Pentium и выше - см. раздел 10.9.1).

Флаг ID: если программа может изменить значение этого флага - процессор поддерживает команду CPUID (только для Pentium и выше).

10.1.2. Регистры управления памятью

Перечисленные ниже четыре регистра используются для указания положения структур данных, ответственных за сегментацию в защищенном режиме.

GDTR: 6-байтный регистр, в котором содержатся 32-битный линейный адрес начала таблицы глобальных дескрипторов (GDT) и ее 16-битный размер (минус 1). Каждый раз, когда происходит обращение к памяти, по

селектору, находящемуся в сегментном регистре, определяется дескриптор из таблицы GDT или LDT, в котором записан адрес начала сегмента и другая информация (см. раздел 6.1).

IDTR: 6-байтный регистр, в котором содержатся 32-битный линейный адрес начала таблицы глобальных дескрипторов обработчиков прерываний (IDT) и ее 16-битный размер (минус 1). Каждый раз, когда происходит прерывание или исключение, процессор передает управление на обработчик, описываемый дескриптором из IDT *с* соответствующим номером.

LDTR: 10-байтный регистр, в котором содержатся 16-битный селектор для GDT и весь 8-байтный дескриптор из GDT, описывающий текущую таблицу локальных дескрипторов (LDT).

TR: 10-байтный регистр, в котором содержатся 16-битный селектор для GDT и весь 8-байтный дескриптор из GDT, описывающий TSS текущей задачи.

10.1.3. Регистры управления процессором

Пять 32-битных регистров CRO - CR4 управляют функционированием процессора и работой отдельных его внутренних блоков.

CRO: флаги управления системой

бит 31: PG - включает и выключает режим страничной адресации

бит 30: CD - запрещает заполнение кэша. При этом чтение из кэша все равно будет происходить

бит 29: NW - запрещает сквозную запись во внутренний кэш - данные, записываемые в кэш, не появляются на внешних выводах процессора

бит 18: AM - разрешает флагу AC включать режим, в котором невыровненные обращения к памяти на уровне привилегий 3 вызывают исключение #AC

бит 16: WP - запрещает запись в страницы, помеченные как «только для чтения» на всех уровнях привилегий (если WP = 0, защита распространяется лишь на уровень 3). Этот бит предназначен для реализации метода копирования процесса, популярного в UNIX, в котором вся память нового процесса сначала полностью совпадает со старым, а затем, при попытке записи, создается копия страницы, к которой происходит обращение

бит 5: NE - включает режим, в котором ошибки FPU вызывают исключение #MF, а не IRQ13

бит 4: ET - использовался только на 80386DX и указывал, что FPU присутствует

бит 3: TS - устанавливается процессором после переключения задачи. Если затем выполнить любую команду FPU, произойдет исключение #NM, обработчик которого может сохранить/восстановить состояние FPU, очистить этот бит командой CLTS и продолжить программу

бит 2: EM - эмуляция сопроцессора. Каждая команда FPU вызывает исключение #NM

бит 1: MP - управляет тем, как исполняется команда WAIT. Должен быть установлен для совместимости с программами, написанными для 80286 и 80386 и использующими эту команду

бит 0: PE - если он равен 1, процессор находится в защищенном режиме

(остальные биты зарезервированы, и программы не должны изменять их значения)

CR1: зарезервирован

CR2: регистр адреса ошибки страницы

Когда происходит исключение #PF, из этого регистра можно прочитать линейный адрес, обращение к которому вызвало исключение.

CR3 (PDBR): регистр основной таблицы страниц

биты 31–11: 20 старших бит физического адреса начала каталога страниц, если бит PAE в CR4 равен нулю, или

биты 31-5: 27 старших бит физического адреса таблицы указателей на каталоги страниц, если бит PAE = 1

бит 4 (80486+): бит PCD (запрещение кэширования страниц) - этот бит запрещает загрузку текущей страницы в кэш-память (например, если произошло прерывание и система не хочет, чтобы обработчик прерывания вытеснил основную программу из кэша)

бит 3 (80486+): бит PWT (бит сквозной записи страниц) - управляет методом записи страниц во внешний кэш

CR4: этот регистр (появился только в процессорах Pentium) управляет новыми возможностями процессоров. Все эти возможности необязательно присутствуют, и их надо сначала проверять с помощью команды CPUID

бит 9: FSR - разрешает команды быстрого сохранения/восстановления состояния FPU/MMX FXSAVE и FXRSTOR (Pentium II)

бит 8: PMC - разрешает выполнение команды RDPMC для программ на всех уровнях привилегий (при PMC = 0 - только на уровне 0) - Pentium Pro и выше

бит 7: PGE - разрешает глобальные страницы (бит 8 атрибута страницы), которые не удаляются из TLB при переключении задач и записи в CR3 (Pentium Pro и выше)

бит 6: MCE — разрешает исключение #MC

бит 5: PAE - включает 36-битное физическое адресное пространство - Pentium Pro и выше

бит 4: PSE - включает режим адресации с 4-мегабайтными страницами

бит 3: DE - запрещает отладочные прерывания по обращению к портам

бит 2: TSD - запрещает выполнение команды RDTSC для всех программ, кроме программ, выполняющихся на уровне привилегий О

бит 1: PVI - разрешает работу флага VIF в защищенном режиме, что может позволить некоторым программам, написанным для уровня привилегий О, работать на более низких уровнях

бит 0: VME - включает расширения режима V86 - разрешает работу флага VIF для V86-приложений

10.1.4. Отладочные регистры

Эти восемь 32-битных регистров (DRO - DR7) позволяют программам, выполняющимся на уровне привилегий 0, определять точки останова, не модифицируя код программ, например для отладки ПЗУ или программ, применяющих сложные схемы защиты от трассировки. Пример отладчика, использующего эти регистры, — SoftICE.

DR7 (DCR) - регистр управления отладкой

биты 31–30: поле LEN для точки останова 3 (размер точки останова)

00 - 1 байт

01 – 2 байта

00 - не определен (например, для останова при выполнении) 11–4 байта

биты 29-28: поле R/W для точки останова 3 (тип точки останова)

00 - при выполнении команды

01 - при записи

10 - при обращении к порту (если бит DE в регистре CR4 = 1)

11 - при чтении или записи

биты 27-26: поле LEN для точки останова 2

биты 25-24: поле R/W для точки останова 2

биты 23-22: поле LEN для точки останова 1

биты 21–20: поле R/W для точки останова 1

биты 19–18: поле LEN для точки останова О

биты 17–16: поле R/W для точки останова О

биты 15-14: 00

бит 13: бит GD - включает режим, в котором любое обращение к отладочному регистру, даже из кольца защиты 0, вызывает исключение #DB (этот бит автоматически сбрасывается внутри обработчика исключения)

биты 12-10: 001

бит 9: бит GE - если этот бит 0, точка останова по обращению к данным может не сработать или сработать на несколько команд позже, так что его лучше всегда сохранять равным 1

бит 7: бит G3 - точка останова 3 включена

бит 5: бит G2 - точка останова 2 включена

бит 3: бит G1 - точка останова 1 включена

бит 2: бит GO - точка останова 0 включена

биты 8, 6, 4, 2, 0: биты LE, L3, L2, L1, LO - действуют так же, как GE - GO, но обнуляются при переключении задачи (локальные точки останова)

DR6 (DSR) - регистр состояния отладки - содержит информацию о причине отладочного останова для обработчика исключения #DB
биты 31–16: единицы
бит 15: BT - причина прерывания - отладочный бит в TSS задачи, в которую только что произошло переключение
бит 14: BS - причина прерывания - флаг трассировки TF из регистра FLAGS
бит 13: BD - причина прерывания - следующая команда собирается писать или читать отладочный регистр, и бит GD в DR7 установлен в 1.
бит 12: 0
биты 11–4: единицы
бит 3: B3 - выполнился останов в точке 3
бит 2: B2 - выполнился останов в точке 2
бит 1: B1 - выполнился останов в точке 1
бит 0: B0 - выполнился останов в точке 0

Процессор не очищает биты причин прерывания в данном регистре, так что обработчику исключения #DB следует делать это самостоятельно. Кроме того, одновременно может произойти прерывание по нескольким причинам, тогда более одного бита будет установлено.

DR4 - DR5 зарезервированы. На процессорах до Pentium или в случае, если бит DE регистра CR4 равен нулю, обращение к этим регистрам приводит к обращению к DR6 и DR7 соответственно. Если бит DE = 1, происходит исключение #UD
DR0 - DR3 содержат 32-битные линейные адреса четырех возможных точек останова по доступу к памяти

Если условия для отладочного останова выполняются, процессор вызывает исключение #DB.

70.7.5. *Машинно-специфичные регистры*

Это большая группа регистров (более ста), назначение которых отличается в моделях процессоров Intel и даже иногда в процессорах одной модели, но разных версий. Например, регистры Pentium Pro MTRR (30 регистров) описывают, какой механизм страничной адресации используют различные области памяти - не кэшируются, защищены от записи, кэшируются прозрачно и т. д. Регистры Pentium Pro MCG/MCI (23 регистра) используются для автоматического обнаружения и обработки аппаратных ошибок, регистры Pentium TR (12 регистров) - для тестирования кэша и т. п. При описании соответствующих команд мы рассмотрим только регистр Pentium TSC - счетчик тактов процессора и группу из четырех регистров Pentium Pro, необходимую для подсчета различных событий (число обращений к кэшу, умножений, команд MMX и т. п.). Эти регистры оказались настолько полезными, что для работы с ними были введены дополнительные команды - RDTSC и RDPMC.

10.2. Системные и привилегированные команды

Команда	Назначение	Процессор
LGOT источник	Загрузить регистр GDTR	80286

Команда загружает значение источника (6-байтная переменная в памяти) в регистр GDTR. Если текущая разрядность операндов 32 бита, в качестве размера таблицы глобальных дескрипторов используются младшие два байта операнда, а в качестве ее линейного адреса — следующие четыре. Если текущая разрядность операндов. - 16 бит, для линейного адреса используются только байты 3, 4, 5 из операнда, а ъ самый старший байт адреса записываются нули.

Команда выполняется исключительно в реальном режиме или при CPL = 0.

Команда	Назначение	Процессор
SGDT приемник	Прочитать регистр GDTR	80286

Помещает содержимое регистра GDTR в приемник (6-байтная переменная в памяти). Если текущая разрядность операндов - 16 бит, самый старший байт этой переменной заполняется нулями (начиная с 80386, а 286 заполнял его единицами).

Команда	Назначение	Процессор
LLDT источник	Загрузить регистр LDTR	80286

Загружает регистр LDTR, основываясь на селекторе, находящемся в источнике (16-битном регистре или переменной). Если источник - 0, все команды, кроме LAR, LSL, VERR и VERW, обращающиеся к дескрипторам из LDT, будут вызывать исключение #GP.

Команда выполняется только в защищенном режиме с CPL = 0.

Команда	Назначение	Процессор
SLDT приемник	Прочитать регистр LDTR	80286

Помещает селектор, находящийся в регистре LDTR, в приемник (16- или 32-битный регистр или переменная). Этот селектор указывает на дескриптор в GDT текущей LDT. Если приемник 32-битный, старшие 16 бит обнуляются на Pentium Pro и не определены на предыдущих процессорах.

Команда выполняется только в защищенном режиме.

Команда	Назначение	Процессор
LTR источник	Загрузить регистр TR	80286

Загружает регистр задачи TR, основываясь на селекторе, находящемся в источнике (16-битном регистре или переменной), который указывает на сегмент состояния задачи (TSS). Эта команда обычно используется для загрузки первой задачи при инициализации многозадачной системы.

Команда выполняется только в защищенном режиме с CPL = 0.

Команда	Назначение	Процессор
STR приемник	Прочитать регистр TR	80286

Помещает селектор, находящийся в регистре TR, в приемник (16- или 32-битный регистр или переменная). Этот селектор указывает на дескриптор в GDT, описывающий TSS текущей задачи. Если приемник 32-битный, старшие 16 бит обнуляются на Pentium Pro и не определены на предыдущих процессорах.

Команда выполняется только в защищенном режиме.

Команда	Назначение	Процессор
LIDT источник	Загрузить регистр IDTR	80286

Загружает значение источника (6-байтная переменная в памяти) в регистр IDTR. Если текущая разрядность операндов - 32 бита, в качестве размера таблицы глобальных дескрипторов используются младшие два байта операнда, а в качестве ее линейного адреса - следующие четыре. Если текущая разрядность операндов - 16 бит, для линейного адреса используются только байты 3, 4, 5 из операнда, а самый старший байт адреса устанавливается нулевым.

Команда выполняется исключительно в реальном режиме или при CPL = 0.

Команда	Назначение	Процессор
SIDT приемник	Прочитать регистр IDTR	80286

Помещает содержимое регистра GDTR в приемник (6-байтная переменная в памяти). Если текущая разрядность операндов - 16 бит, самый старший байт этой переменной заполняется нулями (начиная с 80386, а 286 заполнял его единицами).

Команда	Назначение	Процессор
MOV приемник,источник	Пересылка данных в/из управляющие/их и отладочные/ых регистры/ов	80386

Приемником или источником команды MOV могут быть регистры CRO - CR4 и DRO - DR7. В этом случае другой операнд команды обязательно должен быть 32-битным регистром общего назначения. При записи в регистр CR3 сбрасываются все записи в TLB, кроме глобальных страниц в Pentium Pro. Во время модификации бит PE или PG в CRO и PGE, PSE или PAE в CR4 сбрасываются все записи в TLB без исключения.

Команды выполняются только в реальном режиме или с CPL = 0.

Команда	Назначение	Процессор
LMSW источник	Загрузить слово состояния процессора	80286

Копирует младшие четыре бита источника (16-битный регистр или переменная) в регистр CRO, изменяя биты PE, MP, EM и TS. Кроме того, если бит PE = 1. этой командой его нельзя обнулить, то есть нельзя выйти из защищенного режима.

Команда LMSW существует только для совместимости с процессором 80286, и вместо нее всегда удобнее использовать mov cr0,eax

Команда выполняется только в реальном режиме или с CPL = 0.

Команда	Назначение	Процессор
SMSW приемник	Прочитать слово состояния процессора	80286

Копирует младшие 16 бит регистра CR0 в приемник (16- или 32-битный регистр или 16-битная переменная). Если приемник 32-битный, значения его старших битов не определены. Команда SMSW нужна для совместимости в процессором 80286, и вместо нее удобнее использовать mov eax,cr0.

Команда	Назначение	Процессор
CLTS	Сбросить флаг TS в CR0	80286

Команда сбрасывает в 0 бит TS регистра CR0, который устанавливается процессором в 1 после каждого переключения задач. CLTS предназначена для синхронизации сохранения/восстановления состояния FPU в многозадачных операционных системах: первая же команда FPU в новой задаче при TS = 1 вызовет исключение #NM, обработчик которого сохранит состояние FPU для старой задачи и восстановит сохраненное ранее для новой, после чего выполнит команду CLTS и вернет управление.

Команда выполняется только в реальном режиме или с CPL = 0.

Команда	Назначение	Процессор
ARPL приемник,источник	Коррекция поля RPL селектора	80286

Команда сравнивает поля RPL двух сегментных селекторов. Приемник (16-битный регистр или переменная) содержит первый, а источник (16-битный регистр) - второй. Если RPL приемника меньше, чем RPL источника, устанавливается флаг ZF, и RPL приемника становится равным RPL источника. В противном случае ZF = 0 и никаких изменений не происходит. Обычно эта команда используется операционной системой, чтобы увеличить RPL селектора, переданного ей приложением, с целью удостовериться, что он соответствует уровню привилегий приложения (который система может взять из RPL сегмента кода приложения, находящегося в стеке).

Команда выполняется только в защищенном режиме (с любым CPL).

Команда	Назначение	Процессор
LAR приемник,источник	Прочитать права доступа сегмента	80286

Копирует байты, отвечающие за права доступа из дескриптора, описываемого селектором, который находится в источнике (регистр или переменная), в источник (регистр) и устанавливает флаг ZE Если используются 16-битные операнды, копируется только байт 5 дескриптора в байт 1 (биты 8–15) приемника. Для 32-битных

операндов дополнительно копируются старшие четыре бита (для сегментов кода и данных) или весь шестой байт дескриптора (для системных сегментов) в байт 2 приемника. Остальные биты приемника обнуляются. Если CPL > DPL или RPL > DPL - для неподчиненных сегментов кода, если селектор или дескриптор ошибочны или в других ситуациях, когда программа не сможет пользоваться этим селектором, команда LAR возвращает ZF = 0.

Команда выполняется только в защищенном режиме.

Команда	Назначение	Процессор
LSL приемник,источник	Прочитать лимит сегмента	80286

Копирует лимит сегмента (размер минус 1) из дескриптора, селектор для которого находится в источнике (регистр или переменная), в приемник (регистр) и устанавливает флаг ZF в 1. Если бит гранулярности в дескрипторе установлен и лимит хранится в единицах по 4096 байт, команда LSL переведет его значение в байты. Если используются 16-битные операнды и лимит не умещается в приемнике, его старшие биты теряются. Как и в случае с LAR, эта команда проверяет доступность сегмента из текущей программы: если сегмент недоступен, в приемник ничего не загружается и флаг ZF сбрасывается в 0.

Команда выполняется только в защищенном режиме.

Команда	Назначение	Процессор
VERR источник	Проверить права на чтение	80286
VERW источник	Проверить права на запись	80286

Команды проверяют, доступен ли сегмент кода или данных, селектор которого находится в источнике (16-битный регистр или переменная), для чтения (VERR) или записи (VERW) с текущего уровня привилегий. Если сегмент доступен, то команды возвращают ZF = 1, иначе - ZF = 0.

Команды выполняются только в защищенном режиме.

Команда	Назначение	Процессор
INVD	Сбросить кэш-память	80486
WBINVD	Записать и сбросить кэш-память	80486

Эти команды объявляют все содержимое внутренней кэш-памяти процессора недействительным и подают сигнал для сброса внешнего кэша, так что после этого все обращения к памяти приводят к заполнению кэша заново. Команда WBINVD предварительно сохраняет содержимое кэша в память, команда INVD приводит к потере всей информации, которая попала в кэш, но еще не была перенесена в память.

Команды выполняются только в реальном режиме или с CPL = 0.

Команда	Назначение	Процессор
INVLPG источник	Аннулировать страницу	80486

Аннулирует (объявляет недействительным) элемент буфера TLB, описывающий страницу памяти, которая содержит источник (адрес в памяти).

Команда выполняется только в реальном режиме или с CPL = 0.

Команда	Назначение	Процессор
HLT	Остановить процессор	8086

Переводит процессор в состояние останова, из которого его может вывести только аппаратное прерывание или перезагрузка. Если причиной было прерывание, то адрес возврата, помещаемый в стек для обработчика прерывания, указывает на следующую после HLT команду.

Команда выполняется только в реальном режиме или с CPL = 0.

Команда	Назначение	Процессор
RSM	Выйти из режима SMM	P5

Применяется для вывода процессора из режима SMM, использующегося для сохранения состояния системы в критических ситуациях (например, при выключении электроэнергии). При входе в SMM (происходит во время поступления соответствующего сигнала на процессор от материнской платы) все регистры, включая системные, и другая информация сохраняются в специальном блоке памяти - SMRAM, а при выходе (который и осуществляется командой RSM) все восстанавливается.

Команда выполняется только в режиме SMM.

Команда	Назначение	Процессор
ROMSR	Чтение из MSR-регистра	P5
WRMSR	Запись в MSR-регистр	P5

Помещает содержимое машинно-специфичного регистра с номером, указанным в ECX, в пару регистров EDX:EAX (старшие 32 бита в EDX и младшие в EAX) (RDMSR) или содержимое регистров EDX:EAX - в машинно-специфичный регистр с номером в ECX. Попытка чтения/записи зарезервированного или отсутствующего в данной модели MSR приводит к исключению #GP(0).

Команда выполняется только в реальном режиме или с CPL = 0.

Команда	Назначение	Процессор
RDTSC	Чтение из счетчика тактов процессора	P5

Помещает в регистровую пару EDX:EAX текущее значение счетчика тактов - 64-битного машинно-специфичного регистра TSC, значение которого увеличивается на 1 каждый такт процессора с момента его последней перезагрузки. Этот машинно-специфичный регистр доступен для чтения и записи с помощью команд RDMSR/WRMSR как регистр номер 10h, причем на Pentium Pro при записи в него старшие 32 бита всегда обнуляются. Так как машинно-специфичные регистры

могут отсутствовать на отдельных моделях процессоров, их наличие всегда следует определять посредством CPUID (бит 4 в EDX - наличие TSC).

Команда выполняется на любом уровне привилегий, если бит TSD в регистре CRO равен нулю, и только в реальном режиме или с CPL = 0, если бит TSD = 1.

Команда	Назначение	Процессор
RDPMC	Чтение из счетчика событий	P6

Помещает значение одного из двух программируемых счетчиков событий (40-битные машинно-специфичные регистры C1h и C2h для Pentium Pro и Pentium II) в регистровую пару EDX:EAX. Выбор читаемого регистра определяется числом О или 1 в ECX. Аналогичные регистры есть и на Pentium (и Cyrix 6x86MX), но они имеют номера 11h и 12h, и к ним можно обращаться только при помощи команд RDMSR/WRMSR.

Способ выбора типа подсчитываемых событий тоже различается в Pentium и Pentium Pro - для Pentium надо выполнить запись в 64-битный регистр MSR 0llh, разные двойные слова которого управляют выбором режима каждого из счетчиков и типа событий, а для Pentium Pro/Pentium II надо выполнить запись в регистр 187h для счетчика 0 и 188hдля счетчика 1. Соответственно и наборы событий между этими процессорами сильно различаются: 38 событий на Pentium, 83 - на Pentium Pro и 96 - на Pentium II.

Команда	Назначение	Процессор
SYSENTER	Быстрый системный вызов	PII
SYSEXIT	Быстрый возврат из системного вызова	PИ

Команда SYSENTER загружает в регистр CS число из регистра MSR #174h, в регистр EIP - число из регистра MSR #176h, в регистр SS - число, равное CS + 8 (селектор на следующий дескриптор), и в регистр ESP - число из MSR #175h. Эта команда предназначена для передачи управления операционной системе - ее можно вызывать с любым CPL, а вызываемый код должен находиться в бессегментной памяти с CPL = 0. На самом деле SYSENTER модифицирует дескрипторы используемых сегментов - сегмент кода будет иметь DPL = 0, базу 0, лимит 4 Гб, станет доступным для чтения и 32-битным, а сегмент стека также получит базу О, лимит 4 Гб, DPL = 0, 32-битный режим, доступ для чтения/записи и установленный бит доступа. Кроме того, селекторы CS и SS получают RPL = 0.

Команда SYSEXIT загружает в регистр CS число, равное содержимому регистра MSR #174h плюс 16, в EIP - число из EDX, в SS - число, равное содержимому регистра MSR #174h плюс 24, и в ESP - число из ECX. Эта команда предназначена для передачи управления в бессегментную модель памяти с CPL = 3 и тоже модифицирует дескрипторы. Сегмент кода получает DPL = 3, базу 0, лимит 4 Гб, доступ для чтения, перестает быть подчиненным и становится 32-битным. Сегмент стека также получает базу 0, лимит 4 Гб, доступ для чтения/записи и 32-битную разрядность. Поля RPL в CS и SS устанавливаются в 3.

Поддержку команд SYSENTER/SYSEXIT всегда следует проверять при помощи команды CPUID (бит 11). Кроме того, надо убедиться, что номер модели процессора не меньше трех, так как Pentium Pro (тип процессора 6, модель 1) не имеет команд SYSENTER/SYSEXIT, но бит в CPUID возвращается равным 1.

SYSENTER выполняется лишь в защищенном режиме, а SYSEXIT - только с CPL = 0.

10.3. Вход и выход из защищенного режима

Итак, чтобы перейти в защищенный режим, достаточно установить бит PE - нулевой бит в управляющем регистре CR0, и процессор немедленно окажется в защищенном режиме. Единственное дополнительное требование, которое предъявляет Intel, - в этот момент все прерывания, включая немаскируемое, должны быть отключены.

```
; pm0.asm
; Программа, выполняющая переход в защищенный режим и немедленный возврат.
; Работает в реальном режиме DOS и в DOS-окне Windows 95 (Windows перехватывает
; исключения, возникающие при попытке перехода в защищенный режим из V86,
; и позволяет нам работать, но только на минимальном уровне привилегий).
;
; Компиляция:
; TASM:
;  tasm /m pm0.asm
;  tlink /x /t pm0.obj
; MASM:
;  ml /c pm0.asm
;  link pm0.obj,,NUL,,,
;  exe2bin pm0.exe pm0.com
; WASM:
;  wasm pm0.asm
;  wlink file pm0.obj form DOS COM

        .model  tiny
        .code
        .386p                   ; Все наши примеры рассчитаны на 80386.
        org     100h            ; Это COM-программа.
start:
; Подготовить сегментные регистры.
        push    cs
        pop     ds              ; DS - сегмент данных (и кода) нашей программы.
        push    0B800h
        pop     es              ; ES - сегмент видеопамяти.
; Проверить, находимся ли мы уже в защищенном режиме.
        mov     eax,cr0         ; Прочитать регистр CR0.
        test    al,1            ; Проверить бит PE,
        jz      no_V86          ; если он ноль - мы можем продолжать,
                                ; иначе - сообщить об ошибке и выйти.
        mov     ah, 9           ; Функция DOS 09h.
```

```
        mov     dx,offset v86_msg; DS:DX - адрес строки.
        int     21h             ; Вывод на экран.
        ret                     ; Конец COM-программы
; (поскольку это защищенный режим, в котором работает наша DOS-программа,
; то это должен быть V86).
v86_msg         db      "Процессор в режиме V86 - нельзя переключиться в PM$"

; Сюда передается управление, если мы запущены в реальном режиме.
no_V86:
; Запретить прерывания.
        cli
; Запретить немаскируемое прерывание.
        in      al,70h          ; Индексный порт CMOS.
        or      al,80h          ; Установка бита 7 в нем запрещает NMI.
        out     70h,al
; Перейти в защищенный режим.
        mov     eax,cr0         ; Прочитать регистр CR0.
        or      al,1            ; Установить бит PE,
        mov     cr0,eax         ; с этого момента мы в защищенном режиме.
; Вывод на экран.
        xor     di,di           ; ES:DI - начало видеопамяти.
        mov     si,offset message; DS:SI - выводимый текст.
        mov     cx,message_l
        rep     movsb           ; Вывод текста.
        mov     ax,0720h        ; Пробел с атрибутом 07h.
        mov     cx,rest_scr     ; Заполнить этим символом остаток экрана.
        rep     stosw
; Переключиться в реальный режим.
        mov     eax,cr0         ; Прочитать CR0.
        and     al,0FEh         ; Сбросить бит PE.
        mov     cr0,eax ; С этого момента процессор работает в реальном режиме.
; Разрешить немаскируемое прерывание.
        in      al,70h          ; Индексный порт CMOS.
        and     al,07FH         ; Сброс бита 7 отменяет блокирование NMI.
        out     70h,al
; Разрешить прерывания.
        sti
; Подождать нажатия любой клавиши.
        mov     ah,0
        int     16h
; Выйти из COM-программы.
        ret
; Текст сообщения с атрибутом после каждого символа для прямого вывода на экран.
message         db      'H',7,'e',7,'l',7,'l',7,'o',7,' ',7,'V',7,'3',7
                db      ' ',7,'P',7,'M',7
; Его длина в байтах.
message_l = $-message
; Длина оставшейся части экрана в словах.
rest_scr = (80*25)-(2*message_l)
        end     start
```

В разделе 6.1 при рассмотрении адресации в защищенном режиме говорилось о том, что процессор, обращаясь к памяти, должен определить адрес начала сегмента из дескриптора в таблице дескрипторов, находящейся в памяти, используя селектор, находящийся в сегментном регистре, в качестве индекса. Одновременно при этом мы обращаемся к памяти из защищенного режима, вообще не описав никаких дескрипторов, и в сегментных регистрах у нас находятся те же числа, что и в реальном режиме.

Дело в том, что, начиная с процессора 80286, размер каждого сегментного регистра - CS, SS, DS, ES, FS и GS - не два *байта, а* десять, восемь из которых недоступны для программ, точно так же, как описанные выше регистры LDTR и TR. В защищенном режиме при записи селектора в сегментный регистр процессор копирует весь определяемый этим селектором дескриптор в скрытую часть сегментного регистра и больше не пользуется этим селектором вообще. Таблицу дескрипторов можно уничтожить, а обращения к памяти все равно будут выполняться, как и раньше. В реальном режиме при записи числа в сегментный регистр процессор сам создает соответствующий дескриптор в его скрытой части. Он описывает 16-битный сегмент, начинающийся по указанному адресу с границей 64 Кб. Когда мы переключились в защищенный режим в программе pmO.asm, эти дескрипторы остались на месте и мы могли обращаться к памяти, не принимая во внимание, что у нас написано в сегментном регистре. Разумеется, в такой ситуации любая попытка записать в сегментный регистр число привела бы к немедленной ошибке (исключение #GP с кодом ошибки, равным загружаемому значению).

10.4. Сегментная адресация

10.4.1. Модель памяти в защищенном режиме

Мы уже неоднократно описывали сегментную адресацию, рассказывая о назначении сегментных регистров в реальном режиме или о программировании для расширителей DOS в защищенном режиме, но каждый раз для немедленных нужд требовалась только часть всей этой сложной модели. Теперь самое время рассмотреть ее полностью.

Для любого обращения к памяти в процессорах Intel используется логический адрес, состоящий из 16-битного селектора, который определяет сегмент, и 32- или 16-битного смещения — адреса внутри сегмента. Отдельный сегмент памяти - это независимое защищенное адресное пространство. Для него указаны размер, разрешенные способы доступа (чтение/запись/исполнение кода) и уровень привилегий (см. раздел 10.7). Если доступ к памяти удовлетворяет всем условиям защиты, процессор преобразует логический адрес в 32- или 36-битный (на P6) линейный. *Линейный адрес* - это адрес в несегментированном непрерывном адресном пространстве, совпадающий с физическим адресом в памяти, если отключен режим страничной адресации (см. раздел 10.6). Чтобы получить линейный адрес из логического, процессор добавляет к смещению линейный адрес начала сегмента, который хранится в поле базы в сегментном дескрипторе. *Сегментный дескриптор* - это 8-байтная структура данных, расположенная в таблице

GDT или LDT; адрес таблицы находится в регистре GDTR или LDTR, а номер дескриптора в таблице определяется по значению селектора.

Дескриптор для селектора, находящегося в сегментном регистре, не считывается из памяти при каждом обращении, а хранится в скрытой части сегментного регистра и загружается только при выполнении команд MOV (в сегментный регистр), POP (в сегментный регистр), LDS, LES, LSS, LGS, LFS и дальних команд перехода.

70.4.2. Селектор

Селектор - это 16-битное число следующего формата:
биты 16-3: номер дескриптора в таблице (от 0 до 8191)
бит 2: 1 - использовать LDT, 0 - использовать GDT
биты 1-0: запрашиваемый уровень привилегий при обращении к сегменту и текущий уровень привилегий для селектора, загруженного в CS

Селектор, содержащий нулевые биты 16-3, называется *нулевым* и требуется для загрузки в неиспользуемые сегментные регистры. Любое обращение в сегмент, адресуемый нулевым селектором, приводит к исключению #GP(0), в то время как загрузка в сегментный регистр ошибочного селектора вызывает исключение #GP(селектор). Попытка загрузки нулевого селектора в SS или CS также вызывает #GP(0), поскольку эти селекторы используются всегда.

10.4.3. Дескрипторы

Дескриптор - это 64-битная (восьмибайтная) структура данных, которая может встречаться в таблицах GDT и LDT. Он способен описывать сегмент кода, сегмент данных, сегмент состояния задачи, быть шлюзом вызова, ловушки, прерывания или задачи. В GDT также может находиться дескриптор LDT.

Дескриптор сегмента данных или кода (подробно рассмотрен в разделе 6.1)
байт 7: биты 31-24 базы сегмента
байт 6: бит 7: бит гранулярности (0 - лимит в байтах, 1 - лимит в 4-килобайтных единицах)
бит 6: бит разрядности (0 - 16-битный, 1 - 32-битный сегмент)
бит 5: 0
бит 4: зарезервировано для операционной системы
биты 3-0: биты 19 - 16 лимита
байт 5: (байт доступа)
бит 7: бит присутствия сегмента
биты 6-5: уровень привилегий дескриптора (DPL)
бит 4: 1 (тип дескриптора - не системный)
бит 3: тип сегмента (0 - данных, 1 - кода)
бит 2: бит подчиненности для кода, бит расширения вниз для данных
бит 1: бит разрешения чтения для кода, бит разрешения записи для данных
бит 0: бит доступа (1 - к сегменту было обращение)
байт 4: биты 23-16 базы сегмента
байты 3-2: биты 15-0 базы
байты 1-0: биты 15-0 лимита

Таблица 22. Типы системных дескрипторов

0	Зарезервированный тип	8	Зарезервированный тип
1	Свободный 16-битный TSS	9	Свободный 32-битный TSS
2	Дескриптор таблицы LDT	A	Зарезервированный тип
3	Занятый 16-битный TSS	B	Занятый 32-битный TSS
4	16-битный шлюз вызова	C	32-битный шлюз вызова
5	Шлюз задачи	D	Зарезервированный тип
6	16-битный шлюз прерывания	E	32-битный шлюз прерывания
7	16-битный шлюз ловушки	F	32-битный шлюз ловушки

Если в дескрипторе бит 4 байта доступа равен 0, дескриптор называется *системным*. В этом случае биты 0-3 байта доступа определяют один из 16 возможных типов дескриптора (см. табл. 22).

Дескрипторы шлюзов

Дальние CALL или JMP на адрес с любым смещением и с селектором, указывающим на дескриптор шлюза вызова, приводят к передаче управления по адресу, который есть в дескрипторе. Обычно такие дескрипторы используются для передачи управления между сегментами с различными уровнями привилегий (см. раздел 10.7).

CALL или JMP на адрес с селектором, указывающим на шлюз задачи, приводят к переключению задач (см. раздел 10.8).

Шлюзы прерываний и ловушек используются для вызова обработчиков прерываний и исключений типа ловушки (см. раздел 10.5).

байты 7-6: биты 31 - 16 смещения (0 для 16-битных шлюзов и шлюза задачи)
байт 5: (байт доступа)
    бит 7 - бит присутствия сегмента
    биты 6-5: DPL - уровень привилегий дескриптора
    бит 4: 0
    биты 3–0: тип шлюза (4, 5, 6, 7, C, E, F)
байт 4: биты 7-5: 000
    биты 4-0: 00000 или (для шлюза вызова) число двойных слов, которые будут скопированы из стека вызывающей задачи в стек вызываемой
байты 3-2: селектор сегмента
байты 1-0: биты 15-0 смещения (0 для шлюза задачи)

Дескрипторы TSS и LDT

Эти два типа дескрипторов применяются в многозадачном режиме, о котором рассказано далее. TSS - сегмент состояния задачи, используемый для хранения всей необходимой информации о каждой задаче в многозадачном режиме. LDT - таблица локальных дескрипторов, своя для каждой задачи.

Форматы дескрипторов совпадают с форматом дескриптора для сегмента кода или данных, но при этом бит разрядности всегда равен нулю и, естественно,

системный бит равен нулю, а биты 3-0 байта доступа содержат номер типа сегмента (1, 2, 3, 9, B). Команды JMP и CALL на адрес с селектором, соответствующим TSS незанятой задачи, приводят к переключению задач.

70.4.4. *Пример программы*

Мы будем пользоваться различными дескрипторами по мере надобности, а для начала выполним переключение в 32-битную модель памяти flat, где все сегменты имеют базу 0 и лимит 4 Гб. Нам потребуются два дескриптора - один для кода и один для данных - и два 16-битных дескриптора с лимитами 64 Кб, чтобы загрузить их в CS и DS перед возвратом в реальный режим.

В комментариях к примеру pmO.asm мы заметили, что его можно выполнять в DOS-окне Windows 95, хотя программа запускается уже в защищенном режиме. Это происходит потому, что Windows 95 перехватывает обращения к контрольным регистрам и позволяет программе перейти в защищенный режим, но только с минимальным уровнем привилегий. Все следующие наши примеры в этом разделе будут рассчитаны на работу с максимальными привилегиями, поэтому добавим в программу проверку на запуск из-под Windows (функция 1600h прерывания мультиплексора INT 2Fh).

Еще одно дополнительное действие, которое мы выполним при переключении в защищенный режим, - управление линией A20. После запуска компьютера для совместимости с 8086 используются 20-разрядные адреса (работают адресные линии AO - A19), так что попытка записать что-то по линейному адресу 100000h приведет к записи по адресу OOOOh. Этот режим отменяется установкой бита 2 в порту 92h и снова включается сбрасыванием этого бита в 0. (Существуют и другие способы, зависящие от набора микросхем, которые используются на материнской плате, но они необходимы в том случае, если требуется максимально возможная скорость переключения.)

```
; pm1.asm
; Программа, демонстрирующая работу с сегментами в защищенном режиме.
; Переключается в модель flat, выполняет вывод на экран и возвращается в DOS.
;
; Компиляция:
; TASM:
;  tasm /m pm1.asm
;  tlink /x /3 pm1.obj
; MASM:
;  ml /c pm1.asm
;  link pm1.obj,,NUL,,,
; WASM:
;  wasm pm1.asm
;  wlink file pm1.obj form DOS

        .386p           ; 32-битный защищенный режим появился в 80386.

; 16-битный сегмент, в котором находится код для входа
; и выхода из защищенного режима.
```

```
RM_seg segment para public "code" use16
        assume CS:RM_seg,SS:RM_stack
start:
; Подготовить сегментные регистры.
        push    cs
        pop     ds
; Проверить, не находимся ли мы уже в PM.
        mov     eax,cr0
        test    al,1
        jz      no_V86
; Сообщить и выйти.
        mov     dx,offset v86_msg
err_exit:
        mov     ah,9
        int     21h
        mov     ah,4Ch
        int     21h
v86_msg         db      "Процессор в режиме V86 - нельзя переключиться в PM$"
win_msg         db      "Программа запущена под Windows - нельзя перейти в кольцо 0$"

; Может быть, это Windows 95 делает вид, что PE = 0?
no_V86:
        mov     ax,1600h                        ; Функция 1600h
        int     2Fh                             ; прерывания мультиплексора.
        test    al,al                           ; Если AL = 0,
        jz      no_windows                      ; Windows не запущена.
; Сообщить и выйти, если мы под Windows.
        mov     dx,offset win_msg
        jmp     short err_exit

; Итак, мы точно находимся в реальном режиме.
no_windows:
; Если мы собираемся работать с 32-битной памятью, стоит открыть A20.
        in      al,92h
        or      al,2
        out     92h,al
; Вычислить линейный адрес метки PM_entry.
        xor     eax,eax
        mov     ax,PM_seg                       ; AX - сегментный адрес PM_seg.
        shl     eax,4                           ; EAX - линейный адрес PM_seg.
        add     eax,offset PM_entry             ; EAX - линейный адрес PM_entry.
        mov     dword ptr pm_entry_off,eax      ; Сохранить его.
; Вычислить базу для GDT_16bitCS и GDT_16bitDS.
        xor     eax,eax
        mov     ax,cs                           ; AX - сегментный адрес RM_seg.
        shl     eax,4                           ; EAX - линейный адрес RM_seg.
        push    eax
        mov     word ptr GDT_16bitCS+2,ax       ; Биты 15-0
```

```
        mov     word ptr GDT_16bitDS+2,ax
        shr     eax,16
        mov     byte ptr GDT_16bitCS+4,al ; и биты 23-16.
        mov     byte ptr GDT_16bitDS+4,al
; Вычислить абсолютный адрес метки GDT.
        pop     eax                       ; EAX - линейный адрес RM_seg.
        add     ax,offset GDT             ; EAX - линейный адрес GDT.
        mov     dword ptr gdtr+2,eax      ; Записать его для GDTR.
; Загрузить таблицу глобальных дескрипторов.
        lgdt    fword ptr gdtr
; Запретить прерывания.
        cli
; Запретить немаскируемое прерывание.
        in      al,70h
        or      al,80h
        out     70h,al
; Преключиться в защищенный режим.
        mov     eax,cr0
        or      al,1
        mov     cr0,eax
; Загрузить новый селектор в регистр CS.
                db      66h             ; Префикс изменения разрядности операнда.
                db      0EAh            ; Код команды дальнего jmp.
pm_entry_off    dd      ?               ; 32-битное смещение.
                dw      SEL_flatCS      ; Селектор.

RM_return:      ; Сюда передается управление при выходе из защищенного режима.
; Переключиться в реальный режим.
        mov     eax,cr0
        and     al,0FEh
        mov     cr0,eax
; Сбросить очередь предвыборки и загрузить CS реальным сегментным адресом.
                db      0EAh            ; Код дальнего jmp.
                dw      $+4             ; Адрес следующей команды,
                dw      RM_seg          ; Сегментный адрес RM_seg,
; Разрешить NMI.
        in      al,70h
        and     al,07FH
        out     70h,al
; Разрешить другие прерывания.
        sti
; Подождать нажатия любой клавиши.
        mov     ah,0
        int     16h
; Выйти из программы.
        mov     ah,4Ch
        int     21h
```

```
; Текст сообщения с атрибутами, который мы будем выводить на экран.
message       db      'H',7,'e',7,'l',7,'l',7,'o',7,' ',7,'и',7,'з',7,' ',7
              db      '3',7,'2',7,'-',7,'б',7,'и',7,'т',7,'н',7,'о',7,'г',7
              db      'о',7,' ',7,'Р',7,'М'
message_l = $-message                 ; Длина в байтах.
rest_scr = (80*25*2-message_l)/4 ; Длина оставшейся части экрана в двойных словах.
; Таблица глобальных дескрипторов.
GDT           label   byte
; Нулевой дескриптор (обязательно должен быть на первом месте).
              db      8 dup(0)
; 4-гигабайтный код, DPL = 00:
GDT_flatCS    db      0FFh,0FFh,0,0,0,10011010b,11001111b,0
; 4-гигабайтные данные, DPL = 00:
GDT_flatDS    db      0FFh,0FFh,0,0,0,10010010b,11001111b,0
; 64-килобайтный код, DPL = 00:
GDT_16bitCS   db      0FFh,0FFh,0,0,0,10011010b,0,0
; 64-килобайтные данные, DPL = 00:
GDT_160itDS   db      0FFh,0FFh,0,0,0,10010010b,0,0
GDT_l = $-GDT                         ; Размер GDT.

gdtr          dw      60T_1-1  ; 16-битный лимит GDT.
              dd      ?        ; Здесь будет 32-битный линейный адрес GDT.
; Названия для селекторов (все селекторы для GDT, с RPL = 00).
SEL_flatCS    equ     00001000b
SEL_flatDS    equ     000100000
SEL_16bitCS   equ     00011000b
SEL_16bitDS   equ     00100000b

RM_seg ends

; 32-битный сегмент, содержащий код, который будет исполняться в защищенном режиме.
PM_seg segment para public "CODE" use32
        assume  cs:PM_seg
PM_entry:
; Загрузить сегментные регистры (кроме SS).
        mov     ax,SEL_16bitDS
        mov     ds,ax
        mov     ax,SEL_flatDS
        mov     es,ax
; Вывод на экран.
        mov     esi,offset message      ; DS:ESI - сообщение.
        mov     edi,0B8000h             ; ES:EDI - видеопамять.
        mov     ecx,message_l           ; ECX - длина.
        rep     movsb                   ; Вывод на экран.
        mov     eax,07200720h           ; Два символа 20h с атрибутами 07.
        mov     ecx,rest_scr            ; Остаток экрана / 2.
        rep     stosd                   ; Очистить остаток экрана.
; Загрузить в CS селектор 16-битного сегмента RM_seg.
              db      0EAh              ; Код дальнего jmp.
              dd      offset RM_return  ; 32-битное смещение.
```

```
                dw      SEL_16bitCS     ; Селектор.
PM_seg  ends

; Сегмент стека - используется как в 16-битном, так и в 32-битном режимах;
; поскольку мы не трогали SS, он все время оставался 16-битным.
RM_stack segment para stack "STACK" use16
                db      100h dup(?)
RM_stack        ends
        end     start
```

10.4.5. *Нереальный режим*

Как мы уже знаем, при изменении режима скрытые части сегментных регистров сохраняют содержимое своих дескрипторов и их разрешено применять. Мы воспользовались этой возможностью в нашем первом примере, когда значения, занесенные в сегментные регистры в реальном режиме, загружались в защищенном. Возникает вопрос - а если поступить наоборот: в защищенном режиме загрузить сегментные регистры дескрипторами 4-гигабайтных сегментов с базой 0 и перейти в реальный режим? Оказывается, это прекрасно срабатывает, и мы попадаем в особый режим, который называется нереальным режимом (unreal mode), большим реальным режимом (BRM) или реальным flat-режимом (RFM). Чтобы перейти в нереальный режим, перед переходом в реальный режим надо загрузить в CS дескриптор 16-битного сегмента кода с базой 0 и лимитом 4 Гб и в остальные сегментные регистры - точно такие же дескрипторы сегментов данных.

Теперь весь дальнейший код программы, написанный для реального режима, больше не ограничен рамками 64-килобайтных сегментов и способен работать с любыми массивами. Можно подумать, что первый же обработчик прерывания от таймера загрузит в CS обычное значение и все нормализуется, однако при создании дескриптора в скрытой части сегментного регистра в реальном режиме процессор не трогает поле лимита, а только изменяет базу: что бы мы ни записали в сегментный регистр, сегмент будет иметь размер 4 Гб. Если попробовать вернуться в DOS - она по-прежнему будет работать. Можно запускать программы такого рода:

```
                .model  tiny
                .code
                org     100h
start:  xor     ax,ax
        mov     ds,ax                   ; DS = 0
; Вывести символ в видеопамять:
        mov     word ptr ds:[0B8000h],8403h
        ret
                end     start
```

и они тоже будут работать. Единственное, что отключает этот режим, - программы, переходящие в защищенный режим и обратно, которые устанавливают границы сегментов в 64 Кб, например любые программы, использующие расширители DOS.

Нереальный режим - идеальный вариант для программ, которым необходима 32-битная адресация и свободное обращение ко всем прерываниям BIOS и DOS (традиционный способ состоял бы в работе в защищенном режиме с переключением в V86 для вызова BIOS или DOS, как это делается в случае с DPMI).

Для перехода в этот режим можно воспользоваться, например, такой процедурой:

```
; Область данных:
GDT label byte
                db      8 dup(0)          ; Нулевой дескриптор.
; 16-битный 4 Гб сегмент:
                db      0FFh,0FFh,0,0,0,1001001b,11001111b,0
gdtr            dw      16                ; Размер GDT.
gdt_base        dd      ?                 ; Линейный адрес GDT.

; Код программы.
; Определить линейный адрес GDT.
        xor     eax,eax
        mov     ax,cs
        shl     eax,4
        add     ax,offset GDT
; Загрузить GDT из одного дескриптора (не считая нулевого).
        mov     gdt_base,eax
        lgdt    fword ptr gdtr
; Перейти в защищенный режим.
        cli
        mov     eax,cr0
        or       al,1
        mov     cr0,eax
        jmp     start_PM          ; Сбросить очередь предвыборки.
                                  ; Intel рекомендует
start_PM:                         ; делать jmp после каждой смены режима.
; Загрузить все сегментные регистры дескриптором с лимитом 4 Гб.
        mov     ax,8              ; 8 - селектор нашего дескриптора.
        mov     ds,ax
        mov     es,ax
        mov     fs,ax
        mov     gs,ax
; Перейти в реальный режим.
        mov     eax,cr0
        and     al,0FEh
        mov     cr0,eax
        jmp     exit_PM
exit_PM:
; Записать что-нибудь в каждый сегментный регистр.
        xor     ax,ax
        mov     ds,ax
        mov     es,ax
        mov     fs,ax
        mov     gs,ax
```

```
        sti
        mov     ax,cs
        mov     ds,ax
; И все - теперь процессор находится в реальном режиме с неограниченными сегментами.
```

10.5. Обработка прерываний и исключений

До сих пор все наши программы работали в защищенном режиме с полностью отключенными прерываниями - ими нельзя было управлять с клавиатуры, они не могли работать с дисками и вообще не делали ничего, кроме чтения или записи в те или иные области памяти. Разумеется, ни одна программа не может выполнить что-то серьезное в таком режиме - нам рано или поздно придется обрабатывать прерывания.

В реальном режиме адрес обработчика прерывания считывался процессором из таблицы, находящейся по адресу 0 в памяти. В защищенном режиме эта таблица, называемая IDT - таблицей дескрипторов прерываний, может находиться где угодно. Достаточно того, чтобы ее адрес и размер были загружены в регистр IDTR. Содержимое IDT - не просто адреса обработчиков, как это было в реальном режиме, а дескрипторы трех типов: шлюз прерывания, шлюз ловушки и шлюз задачи (форматы данных дескрипторов рассматривались в предыдущем разделе).

Шлюзы прерываний и ловушек указывают точку входа обработчика, а также его разрядность и уровень привилегий. При передаче управления обработчику процессор помещает в стек флаги и адрес возврата, так же как и в реальном режиме, но после этого для некоторых исключений в стек помещается дополнительный код ошибки, следовательно, не все обработчики можно завершать простой командой IRETD (или IRET для 16-битного варианта). Единственное различие между шлюзом прерывания и ловушки состоит в том, что при передаче управления через шлюз прерывания автоматически запрещаются дальнейшие прерывания, пока обработчик не выполнит IRETD. Этот механизм считается предпочтительным для обработчиков аппаратных прерываний, а шлюз ловушки, который не запрещает прерывания на время исполнения обработчика, лучше использовать для обработки программных прерываний (которые фактически и являются исключениями типа ловушки). Кроме того, в защищенном режиме при вызове обработчика прерывания сбрасывается флаг трассировки TE

Сначала рассмотрим пример программы, обрабатывающей только аппаратное прерывание клавиатуры с помощью шлюза прерываний. Для этого надо составить IDT, загрузить ее адрес командой LIDT и не забыть загрузить то, что содержится в регистре IDTR в реальном режиме, - адрес 0 и размер 4x256, соответствующие: таблице векторов прерываний реального режима.

```
; pm2.asm
; Программа, демонстрирующая обработку аппаратных прерываний в защищенном режиме.
; Переключается в 32-битный защищенный режим и позволяет набирать текст при помощи
; клавиш от 1 до +. Нажатие Backspace стирает предыдущий символ,
; нажатие Esc - выход из программы.
```

```
;
; Компиляция TASM:
;       tasm /m /D_TASM_ pm2.asm
; (или, для версий 3.x, достаточно tasm /m pm2.asm)
;       tlink /x /3 pm2.obj
; Компиляция WASM:
;       wasm /D pm2.asm
;       wlink file pm2.obj form DOS
;
; Варианты того, как разные ассемблеры записывают смещение из 32-битного
; сегмента в 16-битную переменную:
ifdef _TASM_
so              equ     small offset            ; TASM 4.x
else
so              equ     offset                  ; WASM
endif
; Для MASM, по-видимому, придется добавлять лишний код, который преобразует
; смещения, используемые в IDT.

        .386p
RM_seg segment para public "CODE" use16
        assume cs:RM_seg,ds:PM_seg,ss:stack_seg
start:
; Очистить экран.
        mov     ax,3
        int     10h
; Подготовить сегментные регистры.
        push    PM_seg
        pop     ds
; Проверить, не находимся ли мы уже в PM.
        mov     eax,cr0
        test    al,1
        jz      no_V86
; Сообщить и выйти.
        mov     dx,so v86_msg
err_exit:
        mov     ah,9
        int     21h
        mov     ah,4Ch
        int     21h
; Может быть, это Windows 95 делает вид, что PE = 0?
no_V86:
        mov     ax,1600h
        int     2Fh
        test    al,al
        jz      no_windows
; Сообщить и выйти.
        mov     dx,so win_msg
        jmp     short err_exit
```

```
; Итак, мы точно находимся в реальном режиме.
no_windows:
; Вычислить базы для всех используемых дескрипторов сегментов.
        xor     eax,eax
        mov     ax,RM_seg
        shl     eax,4
        mov     word ptr GDT_16bitCS+2,ax ; Базой 16bitCS будет RM_seg.
        shr     eax,16
        mov     byte ptr GDT_16bitCS+4,al
        mov     ax,PM_seg
        shl     eax,4
        mov     word ptr GDT_32bitCS+2,ax ; Базой всех 32bit« будет
        mov     word ptr GDT_32bitSS+2,ax ; PM_seg.
        mov     word ptr GDT_32bitDS+2,ax
        shr     eax,16
        mov     byte ptr GDT_32bitCS+4,al
        mov     byte ptr GDT_32bitSS+4,al
        mov     byte ptr GDT_32bitDS+4,al
; Вычислить линейный адрес GDT.
        xor     eax,eax
        mov     ax,PM_seg
        shl     eax,4
        push    eax
        add     eax,offset GDT
        mov     dword ptr gdtr+2,eax
; Загрузить GDT.
        lgdt    fword ptr gdtr
; Вычислить линейный адрес IDT.
        pop     eax
        add     eax,offset IDT
        mov     dword ptr idtr+2,eax
; Загрузить IDT.
        lidt    fword ptr idtr
; Если мы собираемся работать с 32-битной памятью, стоит открыть A20.
        in      al,92h
        or      al,2
        out     92h,al
; Отключить прерывания,
        cli
; включая NMI.
        in      al,70h
        or      al,80h
        out     70h,al
; Перейти в PM.
        mov     eax,cr0
        or      al,1
        mov     cr0,eax
```

```
; Загрузить SEL_32bitCS в CS.
                db      66h
                db      0EAh
                dd      offset PM_entry
                dw      SEL_32bitCS

RM_return:
; Перейти в RM.
        mov     eax,cr0
        and     al,0FEh
        mov     cr0,eax
; Сбросить очередь и загрузить CS реальным числом.
                db      0EAh
                dw      $+4
                dw      RM_seg
; Установить регистры для работы в реальном режиме.
        mov     ax,PM_seg
        mov     ds,ax
        mov     es,ax
        mov     ax, stack_seg
        mov     bx,stack_l
        mov     ss,ax
        mov     sp,bx
; Загрузить IDTR для реального режима.
        mov     ax,PM_seg
        mov     ds,ax
        lidt    fword ptr idtr_real
; Разрешить NMI.
        in      al,70h
        and     al,07FH
        out     70h,al
; Разрешить прерывания
        sti
; и выйти.
        fflov   ah,4Ch
        int     21h
RM_seg          ends

; 32-битный сегмент.
PM_seg segment para public "CODE" use32
        assume  cs:PM_seg

; Таблицы GDT и IDT должны быть выравнены, так что будем их размещать
; в начале сегмента.
GDT     label   byte
                db      8 dup(0)
; 32-битный 4-гигабайтный сегмент с базой = 0.
GDT_flatDS      db      0FFh,0FFh,0,0,0,10010010b,11001111b,0
; 16-битный 64-килобайтный сегмент кода с базой RM_seg.
GDT_16bitCS     db      0FFh,0FFh,0,0,0,10011010b,0,0
```

```
; 32-битный 4-гигабайтный сегмент кода с базой PM_seg.
GDT_32bitCS   ,   db      0FFh,0FFh,0,0,0,10011010b,11001111b,0
; 32-битный 4-гигабайтный сегмент данных с базой PM_seg.
GDT_32bitDS     db      0FFh,0FFh,0,0,0,10010010b,11001111b,0
; 32-битный 4-гигабайтный сегмент данных с базой stack_seg.
GDT_32bitSS     , db      0FFh,0FFh,0,0,0,10010010b,11001111b,0
gdt_size = $-GDT
gdtr            dw      gdt_size-1              ; Лимит GDT.
                dd      ?                       ; Линейный адрес GDT.
; Имена для селекторов.
SEL_flatDS      equ     001000b
SEL_16bitCS     equ     010000b
SEL_32bitCS     equ     011000b
SEL_32bitDS     equ     100000b
SEL_32bitSS     equ     101000b

; Таблица дескрипторов прерываний IDT.
IDT             label   byte
; Все эти дескрипторы имеют тип 0Eh - 32-битный шлюз прерывания.
; INT 00 - 07
                dw      8 dup(so int_handler,SEL_32bitCS,8E00h,0)
; INT 08 (irq0)
                dw      so irq0_7_handler,SEL_32bitCS,8E00h,0
; INT 09 (irq1)
                dw      so irq1_handler,SEL_32bitCS,8E00h,0
; INT 0Ah - 0Fh (IRQ2 - IRQ8)
                dw      6 dup(so irq0_7_handler,SEL_32bitCS,8E00h,0)
; INT 10h - 6Fh
                dw      97 dup(so int_handler,SEL_32bitCS,8E00h,0)
; INT 70h - 78h (IRQ8 - IRQ15)
                dw      8 dup(so irq8_15_handler,SEL_32bitCS,8E00h,0)
; INT 79h - FFh
                dw      135 dup(so int_handler,SEL_32bitCS,8E00h,0)
idt_size = $-IDT                                ; Размер IDT.
idtr            dw      idt_size-1              ; Лимит IDT.
                dd      ?                       ; Линейный адрес начала IDT.
; Содержимое регистра IDTR в реальном режиме.
idtr_real       dw      3FFh,0,0

; Сообщения об ошибках при старте.
v86_msg         db      "Процессор в режиме V86 - нельзя переключиться в PM$"
win_msg         db      "Программа запущена под Windows - нельзя перейти в кольцо 0$"

; Таблица для перевода 0E скан-кодов в ASCII.
scan2ascii      db      0,1Bh,'1','2','3','4','5','6','7','8','9','0','-','=',8
screen_addr     dd      0                       ; Текущая позиция на экране.

; Точка входа в 32-битный защищенный режим.
PM_entry:
; Установить 32-битный стек и другие регистры.
        mov     ax,SEL_flatDS
```

```
        mov     ds,ax
        mov     es,ax
        mov     ax,SEL_32bitSS
        mov     ebx,stack_1
        mov     ss,ax
        mov     esp,ebx
; Разрешить прерывания
        sti
; и войти в вечный цикл.
        jmp short $

; Обработчик обычного прерывания.
int_handler:
        iretd
; Обработчик аппаратного прерывания IRQ0 - IRQ7.
irq0_7_handler:
        push    eax
        mov     al,20h
        out     20h,al
        pop     eax
        iretd
; Обработчик аппаратного прерывания IRQ8 - IRQ15.
irq8_15_handler:
        push    eax
        mov     al,20h
        out     0A1h,al
        pop     eax
        iretd
; Обработчик IRQ1 - прерывания от клавиатуры.
irq1_handler:
        push    eax             ; Это аппаратное прерывание - сохранить регистры.
        push    ebx
        push    es
        push    ds
        in      al,60h                  ; Прочитать скан-код нажатой клавиши.
        cmp     al,0Eh                  ; Если он больше, чем
        ja      skip_translate          ; обслуживаемый нами,- не обрабатывать.
        cmp     al,1                    ; Если это Esc,
        je      esc_pressed             ; выйти в реальный режим.
        mov     bx,SEL_32bitDS          ; Иначе:
        mov     ds,bx                   ; DS:EBX - таблица для перевода скан-кода
        mov     ebx,offset scan2ascii   ; в ASCII,
        xlatb                           ; Преобразовать.
        mov     bx,SEL_flatDS
        mov     es,bx                   ; ES:EBX - адрес текущей
        mov     ebx,screen_addr         ; позиции на экране.
        cmp     al,8                    ; Если не была нажата Backspace,
        je      bs_pressed
```

```
        mov     es:[ebx+0B8000h],al     ; послать символ на экран.
        add     dword ptr screen_addr,2 ; Увеличить адрес позиции на 2.
        jmp     short skip_translate
bs_pressed:                             ; Иначе:
        mov     al,' '                  ; нарисовать пробел
        sub     ebx,2                   ; в позиции предыдущего символа
        mov     es:[ebx+0B8000h],al
        mov     screen_addr,ebx         ; и сохранить адрес предыдущего символа
skip_translate:                         ; как текущий.
; Разрешить работу клавиатуры.
        in      al,61h
        or      al,80h
        out     61h,al
; Послать EOI контроллеру прерываний.
        mov     al,20h
        out     20h,al
; Восстановить регистры и выйти.
        pop     ds
        pop     es
        pop     ebx
        pop     eax
        iretd

; Сюда передается управление из обработчика IRQ1, если нажата Esc.
esc_pressed:
; Разрешить работу клавиатуры, послать EOI и восстановить регистры.
        in      al,61h
        or      al,80h
        out     61h,al
        mov     al,20h
        out     20h,al
        pop     ds
        pop     es
        pop     ebx
        pop     eax
; Вернуться в реальный режим.
        cli
                db      0EAh
                dd      offset RM_return
                dw      SEL_16bitCS
PM_seg  ends

; Сегмент стека. Используется как 16-битный в 16-битной части программы и как
; 32-битный (через селектор SEL_32bitSS) в 32-битной части.
stack_seg segment para stack "STACK"
stack_start     db      100h dup(?)
stack_l = $-stack_start                 ; Длина стека для инициализации ESP.
stack_seg       ends
        end     start
```

В этом примере обрабатываются только 13 скан-кодов клавиш для сокращения размеров программы. Полную информацию преобразования скан-кодов в ASCII можно найти в таблицах приложения 1 (см. рис. 18, табл. 25 и 26). Кроме того, в этом примере курсор все время остается в нижнем левом углу экрана – для его перемещения можно воспользоваться регистрами 0Eh и OFh контроллера CRT (см. раздел 5.10.4).

Как уже упоминалось в разделе 5.8, кроме прерываний от внешних устройств процессор может вызывать исключения при различных внутренних ситуациях (их механизм обслуживания похож на механизм обслуживания аппаратных прерываний). Номера прерываний, на которые отображаются аппаратные прерывания, вызываемые первым контроллером по умолчанию, совпадают с номерами отдельных исключений. Конечно, можно из обработчика опрашивать контроллер прерывание, чтобы определить, выполняется ли аппаратное прерывания или это исключение, но Intel рекомендует перенастраивать контроллер прерываний (см. раздел 5.10.10) так, чтобы никакие аппаратные прерывания не попадали на область от 0 до 1Fh. В нашем примере исключения не обрабатывались, но, если программа планирует запускать другие программы или задачи, без обработки исключений обойтись нельзя.

Часть исключений (исключения типа ошибки) передает в качестве адреса возврата команду, вызвавшую исключение, а часть - адрес следующей команды. Кроме того, некоторые исключения помещают в стек код ошибки, который нужно считать, прежде чем выполнять IRETD. Поэтому пустой обработчик из одной команды IRETD в нашем примере не был корректным и многие исключения привели бы к немедленному зависанию системы.

Рассмотрим исключения в том виде, в каком они определены для защищенного режима.

Формат кода ошибки:

биты 15–3: биты 15-3 селектора, вызвавшего исключение
бит 2: TI - установлен, если причина исключения - дескриптор, находящийся в LDT; и сброшен, если в GDT
бит 1: IDT - установлен, если причина исключения — дескриптор, находящийся в IDT
бит 0: EXT — установлен, если причина исключения - аппаратное прерывание

INT 00 - ошибка #DE «Деление на ноль»

Вызывается командами DIV или IDIV, если делитель - ноль или если происходит переполнение.

INT 01 - исключение #DB «Отладочное прерывание»

Вызывается как ловушка при пошаговой трассировке (флаг TF = 1), при переключении на задачу с установленным отладочным флагом и при срабатывании точки останова во время доступа к данным, определенной в отладочных регистрах.

Вызывается как ошибка при срабатывании точки останова по выполнению команды с адресом, определенным в отладочных регистрах.

INT 02 - прерывание NMI
Немаскируемое прерывание.

INT 03 - ловушка #BP «Точка останова»
Вызывается однобайтной командой INT3.

INT 04 - ловушка #OF «Переполнение»
Вызывается командой INTO, если флаг OF = 1.

INT 05 - ошибка #BR «Переполнение при BOUND»
Вызывается командой BOUND при выходе операнда за допустимые границы.

INT 06 - ошибка #UD «Недопустимая операция»
Вызывается, когда процессор пытается исполнить недопустимую команду или команду с недопустимыми операндами.

INT 07 - ошибка #NM «Сопроцессор отсутствует»
Вызывается любой командой FPU, кроме WAIT, если бит EM регистра CRO установлен в 1, и командой WAIT, если MP и TS установлены в 1.

INT 08 - ошибка #DF «Двойная ошибка»
Вызывается, если одновременно произошли два исключения, которые не могут быть обслужены последовательно. К ним относятся #DE, #TS, #NP, #SS, #GP и #PF.
Обработчик этого исключения получает код ошибки, который всегда равен нулю.
Если при вызове обработчика #DF происходит еще одно исключение, процессор отключается и может быть выведен из этого состояния только сигналом NMI или перезагрузкой.

INT 09h - зарезервировано
Эта ошибка вызывалась сопроцессором 80387, если происходило исключение #PF или #GP при передаче операнда команды FPU.

INT 0Ah - ошибка #TS «Ошибочный TSS»
Вызывается при попытке переключения на задачу с ошибочным TSS.
Обработчик этого исключения должен вызываться через шлюз задачи.
Обработчик этого исключения получает код ошибки.
Бит EXT кода ошибки установлен, если переключение пыталось выполнить аппаратное прерывание, использующее шлюз задачи. Индекс ошибки равен селектору TSS, если TSS меньше 67h байт, селектору LDT, если LDT отсутствует или ошибочен, селектору сегмента стека, кода или данных, если ими нельзя пользоваться (из-за нарушений защиты или ошибок в селекторе).

INT 0Bh - ошибка #NP «Сегмент недоступен»
Вызывается при попытке загрузить в регистр CS, DS, ES, FS или GS селектор сегмента, в дескрипторе которого сброшен бит присутствия сегмента (загрузка в SS вызывает исключение #SS), а также при попытке использовать шлюз, помеченный как отсутствующий, или при загрузке таблицы локальных дескрипторов командой LLDT (загрузка при переключении задач приводит к исключению #TS).

Если операционная система реализует виртуальную память на уровне сегментов, обработчик этого исключения может загрузить отсутствующий сегмент в память, установить бит присутствия и вернуть управление.

Обработчик этого исключения получает код ошибки.

Бит EXT кода ошибки устанавливается, если причина ошибки - внешнее прерывание, бит IDT устанавливается, если причина ошибки - шлюз из IDT, помеченный как отсутствующий. Индекс ошибки равен селектору отсутствующего сегмента.

INT 0Ch - ошибка #SS «Ошибка стека»

Это исключение вызывается при попытке выхода за пределы сегмента стека во время выполнения любой команды, работающей со стеком, - как явно (POP, PUSH, ENTER, LEAVE), так и неявно (MOV AX, [BP + 6]), а также при попытке загрузить в регистр SS селектор сегмента, помеченного как отсутствующий (не только во время выполнения команд MOV, POP и LSS, но и во время переключения задач, вызова и возврата из процедуры на другом уровне привилегий).

Обработчик этого исключения получает код ошибки.

Код ошибки равен селектору сегмента, вызвавшего ошибку, если она произошла из-за отсутствия сегмента или при переполнении нового стека в межуровневой команде GALL. Во всех остальных случаях код ошибки - ноль.

INT 0Dh - исключение #GP «Общая ошибка защиты»

Все ошибки и ловушки, не приводящие к другим исключениям, вызывают #GP - в основном нарушения привилегий.

Обработчик этого исключения получает код ошибки.

Если ошибка произошла при загрузке селектора в сегментный регистр, код ошибки равен этому селектору, во всех остальных случаях код ошибки - ноль.

INT 0Eh - ошибка #PF «Ошибка страничной адресации»

Вызывается, если в режиме страничной адресации программа пытается обратиться к странице, которая помечена как отсутствующая или привилегированная.

Обработчик этого исключения получает код ошибки.

Код ошибки использует формат, отличающийся для других исключений:

бит 0: 1, если причина ошибки - нарушение привилегий;
О, если было обращение к отсутствующей странице;

бит 1: 1, если выполнялась операция записи; 0, если чтения;

бит 2: 1, если операция выполнялась из CPL = 3; 0, если CPL < 3;

бит 3: 0, если ошибку вызвала попытка установить зарезервированный бит в каталоге страниц.

Остальные биты зарезервированы.

Кроме кода ошибки обработчик этого исключения может прочитать из регистра CR2 линейный адрес, преобразование которого в физический вызвало исключение.

Исключение #PF - основное исключение для создания виртуальной памяти с использованием механизма страничной адресации.

INT 0Fh - зарезервировано

INT 10h - ошибка #MF «Ошибка сопроцессора»

Вызывается, только если бит NE в регистре CRO установлен в 1 при выполнении любой команды FPU, кроме управляющих команд и WAIT/FWAIT, при условии, что в FPU произошло одно из исключений FPU (см. раздел 2.4.3).

INT 11h - ошибка #AC «Ошибка выравнивания»

Вызывается, только если бит AM в регистре CRO и флаг AC из EFLAGS установлены в 1, если CPL = 3 и произошло невыравненное обращение к памяти. (Выравнивание должно быть по границе слова при обращении к слову, к границе двойного слова, к двойному слову и т. д.)

Обработчик этого исключения получает код ошибки равный нулю.

INT 12h - останов #MC «Машинно-зависимая ошибка»

Вызывается (начиная с Pentium) при обнаружении некоторых аппаратных ошибок с помощью специальных машинно-зависимых регистров MCG_*. Наличие кода ошибки, так же как и способ вызова этого исключения, зависит от модели процессора.

INT 13h - 1Fh - зарезервировано Intel для будущих исключений

INT 20h - 0FFh - выделены для использования программами

Обычно для отладочных целей многие программы, работающие с защищенным режимом, устанавливают обработчики всех исключений, которые выдают список регистров процессора и их содержимое, а также иногда участок кода, вызвавший исключение. В качестве примера обработчика исключения типа ошибки можно рассматривать программу, обрабатывающую #BR (см. раздел 5.8.1).

10.6. Страничная адресация

Линейный адрес, который формируется процессором из логического адреса, соответствует адресу из линейного непрерывного пространства памяти. В обычном режиме в это пространство могут попадать области памяти, куда нежелательно разрешать запись, - системные таблицы и процедуры, ПЗУ BIOS и т. д. Чтобы этого избежать, система может позволять программам создавать только небольшие сегменты, но тогда теряется привлекательная идея flat-памяти. Сегментация - не единственный вариант организации памяти, который поддерживают процессоры Intel. Существует второй, совершенно независимый механизм - *страничная адресация* (pagination).

При страничной адресации непрерывное пространство линейных адресов памяти разбивается на страницы фиксированного размера (обычно 4 Кб (4096 или 1000h байт), но Pentium Pro может поддерживать и страницы по 4 Мб). При обращении к памяти процессор физически обращается не по линейному адресу, а по тому физическому адресу, с которого начинается данная страница. Описание каждой страницы из линейного адресного пространства, включающее в себя ее физический адрес и дополнительные атрибуты, хранится в одной из специальных

системных таблиц, как и в случае сегментации, но при этом страничная адресация абсолютно невидима для программы.

Страничная адресация включается при установке бита PG регистра CR0, если бит PE зафиксирован в 1 (попытка установить PG, оставаясь в реальном режиме, приводит к исключению #GP(0)). Кроме того, в регистр CR3 предварительно надо поместить физический адрес начала каталога страниц - главной из таблиц, описывающих страничную адресацию. Каталог страниц имеет размер 4096 байт (ровно одна страница) и содержит 1024 4-байтных указателя на таблицы страниц. Каждая таблица страниц тоже имеет размер 4096 байт и содержит указатели до 1024 4-килобайтных страниц. Если одна страница описывает 4 Кб, то полностью заполненная таблица страниц описывает 4 Мб, а полный каталог полностью заполненных таблиц - 4 Гб, то есть все 32-битное линейное адресное пространство. Когда процессор выполняет обращение к линейному адресу, он сначала использует его биты 31–22 как номер таблицы страниц в каталоге, затем биты 21–12 как номер страницы в выбранной таблице, а затем биты 11–0 как смещение от физического адреса начала страницы в памяти. Поскольку эта процедура занимает много времени, в процессоре предусмотрен специальный кэш страниц - TLB (буфер с ассоциативной выборкой), так что, если к странице обращались не очень давно, ее физический адрес будет сразу определен.

Элементы каталога страниц и таблиц страниц имеют общий формат:

биты 31–12: биты 31-12 физического адреса (таблицы страниц или самой страницы)

биты 11–9: доступны для использования операционной системой

бит 8: G - «глобальная страница» - страница не удаляется из буфера TLB при переключении задач или перезагрузке регистра CR3 (только на Pentium Pro, если установлен бит PGE регистра CR4)

бит 7: PS - размер страницы. 1 - для страницы размером 2 или 4 Мб, иначе - 0

бит 6: D - «грязная страница» - устанавливается в 1 при записи в страницу; всегда равен нулю для элементов каталога страниц

бит 5: A - бит доступа (устанавливается в 1 при любом обращении к таблице страниц или отдельной странице)

бит 4: PCD - бит запрещения кэширования

бит 3: PWT - бит разрешения сквозной записи

бит 2: U - страница/таблица доступна для программ с CPL = 3

бит 1: W - страница/таблица доступна для записи

бит 0: P - страница/таблица присутствует. Если этот бит - 0, остальные биты элемента система может использовать по своему усмотрению, например, чтобы хранить информацию о том, где физически находится отсутствующая страница

Процессоры Pentium Pro (и старше) могут поддерживать расширения страничной адресации. Если установлен бит PAE, физический адрес оказывается не 32-битным (до 4 Гб), а 36-битным (до 64 Гб). Если установлен бит PSE регистра

CR4, включается поддержка расширенных страниц размером 4 Мб для PAE = 0 и 2 Мб для PAE = 1. Такие страницы описываются не в таблицах страниц, а в основном каталоге. Intel рекомендует помещать ядро операционной системы и все, что ему необходимо для работы, на одну 4-мегабайтную страницу, а для приложений пользоваться 4-килобайтными страницами. Расширенные страницы кэшируются в отдельном TLB, так что, если определена всего одна расширенная страница, она будет оставаться в TLB все время.

Для расширенных страниц формат элемента каталога совпадает с форматом для обычной страницы (кроме того, что бит PS = 1), но в качестве адреса используются только биты 31-22 — они соответствуют битам 31-22 физического адреса начала страницы (остальные биты адреса - нули).

Для расширенного физического адреса (PAE = 1) изменяется формат регистра CR3 (см. раздел 10.1.3), размеры всех элементов таблиц становятся равными 8 байтам (причем используются только биты 0-3 байта 4), поэтому их число сокращается до 512 элементов в таблице и вводится новая таблица - таблица указателей на каталоги страниц. Она состоит из четырех 8-байтных элементов, каждый из которых может указывать на отдельный каталог страниц. В этом случае биты 31-30 линейного адреса определяют используемый каталог страниц, биты 29-21 - таблицу, биты 20–12 - страницу, а биты 11–0 - смещение от начала страницы в физическом пространстве (следовательно, если биты 29-21 выбрали расширенную страницу, биты 20-0 соответствуют смещению в ней).

Основная цель страничной адресации - организация виртуальной памяти в ОС. Система может использовать внешние устройства (обычно диск) для расширения виртуального размера памяти. При этом, если к какой-то странице долгое время нет обращений, система копирует ее на диск и помечает как отсутствующую в таблице страниц. Затем, когда программа обращается по адресу в отсутствующей странице, вызывается исключение #PE Обработчик исключения читает адрес, приведший к ошибке из CR2, определяет, какой странице он соответствует, загружает ее с диска, устанавливает бит присутствия, удаляет копию старой страницы из TLB командой INVLPG и возвращает управление (не забыв снять со стека код ошибки). Команда, вызывавшая исключение типа ошибки, выполняется повторно.

Кроме того, система может периодически сбрасывать бит доступа и, если он не установится за достаточно долгое время, копировать страницу на диск и объявлять ее отсутствующей. Если при этом бит D равен нулю, в страницу не выполнялось никаких записей (с того момента, как этот бит последний раз обнулили) и ее вообще можно не сохранять.

Второе не менее важное применение страничной адресации - безопасная реализация flat-модели памяти. Операционная система может разрешить программам обращаться к любому линейному адресу, но отображение линейного пространства на физическое не будет взаимно однозначным. Скажем, если система использует первые 4 Кб памяти, физическим адресом нулевой страницы будет не ноль, а 4096 и пользовательская программа даже не узнает, что обращается не к нулевому

адресу. В этом случае, правда, и сама система не сможет воспользоваться первой физической страницей без изменения таблицы страниц, но эта проблема/решается при применении механизма многозадачности, о котором рассказано далее.

В следующем примере мы построим каталог и таблицу страниц (для первых 4 Мб), отображающие линейное пространство в физическое один в один, затем изменим физический адрес страницы с линейным адресом 0A1000h и попытаемся выполнить обычный цикл закраски экрана в режиме 320x200x256, заполнив видеопамять байтом с номером цвета, но у нас останется незакрашенным участок, соответствующий перенесенной странице.

```
; pm3.asm
; Программа, демонстрирующая страничную адресацию.
; Переносит одну из страниц, составляющих видеопамять, и пытается закрасить экран.

; Компиляция:
; TASM:
;  tasm /m pm3.asm
;  tlink /x /3 pm3.obj
; MASM:
;  ml /c pm3.asm
;  link pm3.obj,,NUL,,,
; WASM:
;  wasm pm3.asm
;  wlink file pm3.obj form DOS

        .386p
RM_seg segment para public "CODE" use16
        assume cs:RM_seg,ds:PM_seg,ss:stack_seg
start:
; Подготовить сегментные регистры.
        push    PM_seg
        pop     ds
; Проверить, не находимся ли мы уже в PM.
        mov     eax,cr0
        test    al,1
        jz      no_V86
; Сообщить и выйти.
        mov     dx,offset v86_msg
err_exit:
        push    cs
        pop     ds
        mov     ah,9
        int     21h
        mov     ah,4Ch
        int     21h

; Убедиться, что мы не под Windows.
no_V86:
        mov     ax,1600h
        int     2Fh
```

```
        test    al,al
        jz      no_windows
; Сообщить и выйти.
        mov     dx,offset win_msg
        jmp     short err_exit

; Сообщения об ошибках при старте.
v86_msg  db     "Процессор в режиме V86 - нельзя переключиться в PM$"
winjnsg  db     "Программа запущена под Windows - нельзя перейти в кольцо 0$"

; Итак, мы точно находимся в реальном режиме.
no_windows:
; Очистить экран и переключиться в нужный видеорежим.
        mov     ax,13h
        int     10h
; Вычислить базы для всех дескрипторов.
        xor     eax,eax
        mov     ax,RM_seg
        shl     eax,4
        mov     word ptr GDT_16bitCS+2,ax
        shr     eax,16
        mov     byte ptr GDT_16bitCS+4,al
        mov     ax,PM_seg
        shl     eax,4
        mov     word ptr GDT_32bitCS+2,ax
        shr     eax,16
        mov     byte ptr GDT_32bitCS+4,al
; Вычислить линейный адрес GDT.
        xor     eax,eax
        mov     ax,PM_seg
        shl     eax,4
        posh    eax
        add     eax,offset GDT
        mov     dword ptr gdtr+2,eax
; Загрузить GDT.
        lgdt    fword ptr gdtr
; Открыть A20 - в этом примере мы будем пользоваться памятью выше 1 Мб.
        mov     al,2
        out     92h,al
; Отключить прерывания
        cli
; и NMI.
        in      al,70h
        or      al,80h
        out     70h,al
; Перейти в защищенный режим (пока без страничной адресации).
        mov     eax,cr0
        or      al,1
        mov     cr0,eax
```

```
; Загрузить CS.
                db      66h
                db      OEAh
                dd      offset PM_entry
                dw      SEL_32bitCS

RM_return:
; Переключиться в реальный режим с отключением страничной адресации.
        mov     eax,cr0
        and     eax,7FFFFFFEh
        mov     cr0,eax
; Сбросить очередь и загрузить CS.
                db.     OEAh
                dw      $+4
                dw      RM_seg
; Загрузить остальные регистры.
        mov     ax,PM_seg
        mov     ds,ax
        mov     es,ax
;  Разрешить MI.
        in      al,70h
        and     al,07FH
        out     70h,al
; Разрешить другие прерывания.
        sti
;  Подождать нажатия клавиши.
        mov     ah,1
        int     21h
; Переключиться в текстовый режим
        mov     ax,3
        int     10h
; и завершить программу.
        mov     ah,4Ch
        int     21h
RM_seg  ends

PM_seg segment para public "CODE" use32
        assume  cs:PM_seg
; Таблица глобальных дескрипторов.
GDT             label   byte
                db      8 dup(0)
GDT_flatDS      db      0FFh,0FFh,0,0,0, 10010010b,'11001111b,0
GOT_16bitCS     db      0FFh,0FFh,0,0,0,10011010b,0,0
GDT_32bitCS     db      0FFh,0FFh,0,0,0,10011010b,11001111b,0
gdt_size = $-GDT
gdtr            dw      gdt_size-1      ; Ее лимит
                dd      ?               ; и адрес.
SEL_flatDS      equ     001000b         ; Селектор 4-гигабайтного сегмента данных.
SEL_16bitCS     equ     010000b         ; Селектор сегмента кода RM_seg.
SEL_32bitCS     equ     011000b         ; Селектор сегмента кода PM_seg.
```

```
; Точка входа в 32-битный защищенный режим.
PM_entry:
; Загрузить сегментные регистры, включая стек.
        xor     eax,eax
        mov     ax,SEL_flatDS
        mov     ds,ax
        mov     es,ax
; Создать каталог страниц.
        mov     edi,00100000h           ; Его физический адрес - 1 Мб.
        mov     eax,00101007h           ; Адрес таблицы 0 = 1 Мб + 4 Кб.
        stosd                           ; Записать первый элемент каталога.
        mov     ecx,1023                ; Остальные элементы каталога -
        xor     eax,eax                 ; нули.
        rep     stosd
; Заполнить таблицу страниц 0.
        mov     eax,00000007h           ; 0 - адрес страницы 0.
        mov     ecx,1024                ; Число страниц в таблице.
page_table:
        stosd                           ; Записать элемент таблицы.
        add     eax,00001000h           ; Добавить к адресу 4096 байт
        loop    page_table              ; и повторить для всех элементов.
; Поместить адрес каталога страниц в CR3.
        mov     eax,00100000h           ; Базовый адрес = 1 Мб.
        mov     cr3,eax
; Включить страничную адресацию.
        mov     eax,cr0
        or      eax,80000000h
        mov     cr0,eax
; А теперь изменить физический адрес страницы A1000h на A2000h.
        mov     eax,000A2007h
        mov     es:00101000h+0A1h*4,eax
; Если закомментировать предыдущие две команды, следующие четыре
; закрасят весь экран синим цветом, но из-за того, что мы переместили одну
; страницу, останется черный участок.
        mov     ecx,(320*200)/4         ; Размер экрана в двойных словах.
        mov     edi,0A0000h             ; Линейный адрес начала видеопамяти.
        mov     eax,01010101h           ; Код синего цвета в VGA - 1.
        rep     stosd
; Вернуться, в реальный режим.
                db      0EAh
                dd      offset RM_return
                dw      SEL_16bitCS
PM_seg          ends

; Сегмент стека - используется как 16-битный.
stack_seg segment para stack "STACK"
stack_start     db      100h dup(?)
stack_seg       ends
        end     start
```

10.7. Механизм защиты

Теперь рассмотрим механизм, который дал название режиму процессора, - механизм защиты. Защита может действовать как на уровне сегментов, так и на уровне страниц, ограничивая доступ в зависимости от уровня привилегий (четыре уровня привилегий для сегментов и два для страниц). Она предотвращает возможность вносить изменения в области памяти, занятые операционной системой или более привилегированной программой. Процессор проверяет привилегии непосредственно перед каждым обращением к памяти и, если происходит нарушение защиты, вызывает исключение #GP.

Когда процессор находится в защищенном режиме, проверки привилегий выполняются всегда и их нельзя отключить, но можно использовать во всех дескрипторах и селекторах один и тот же максимальный уровень привилегий - нулевой, и создастся видимость отсутствия защиты. Именно так мы поступали в вышеописанных примерах - поля DPL и RPL инициализировались нулями. Для осуществления незаметной проверки прав на уровне страничной адресации надо установить биты U и W во всех элементах таблиц страниц, что мы также выполняли в программе pm3.asm.

За механизм защиты отвечают следующие биты и поля:

- в дескрипторах сегментов:
  - бит S (системный сегмент);
  - поле типа (тип сегмента, включая запреты на чтение/запись);
  - поле лимита сегмента;
  - поле DPL, определяющее привилегии сегмента или шлюза, указывает, по крайней мере, какой уровень привилегий должна иметь программа, чтобы обратиться к этому сегменту или шлюзу;
- в селекторах сегментов:
  - поле RPL, определяющее запрашиваемые привилегии, позволяет программам, выполняющимся на высоких уровнях привилегий, обращаться к сегментам, как будто их уровень привилегий ниже;
  - поле RPL селектора, загруженного в CS, называется CPL и является текущим уровнем привилегий программы;
- в элементах таблиц страниц:
  - бит U (определяет уровень привилегий страницы);
  - бит W (разрешает/запрещает запись).

Уровни привилегий в процессорах Intel:

- 0 - максимальный (для операционной системы);
- а 1 и 2 - промежуточные (для вспомогательных программ);
- 3 - минимальный (для пользовательских приложений).

Перед обращением к памяти процессор выполняет несколько типов проверок, использующих все указанные флаги и поля. Рассмотрим их по порядку.

10.7.1. Проверка лимитов

Поле лимита в дескрипторе сегмента запрещает доступ к памяти за пределами сегмента. Если бит G дескриптора равен нулю, значения лимита могут быть от 0 до

0FFFFFh (1 Мб). Если бит G установлен - от OFFFh (4 Кб) до OFFFFFFFFh (4 Гб). Для сегментов, растущих вниз, лимит принимает значения от указанного плюс 1 до OFFFFh для 16-битных сегментов данных и до OFFFFFFFFh - для 32-битных. Эти проверки отлавливают такие ошибки, как неправильные вычисления адресов.

Перед проверкой лимита в дескрипторе процессор выясняет лимит самой таблицы дескрипторов на тот случай, если указано слишком большое значение селектора.

Во всех случаях исключение #GP вызывается с кодом ошибки, равным индексу селектора, посредством которого нарушается защита.

70.7.2. Проверка типа сегмента

1. Загрузка селектора (и дескриптора) в регистр:
   - в CS можно загрузить только сегмент кода;
   - в DS, ES, FS, GS можно загрузить только селектор сегмента данных, сегмента кода, доступного для чтения, или нулевой селектор;
   - в SS можно загрузить только сегмент данных, доступный для записи;
   - в LDTR можно загрузить только сегмент LDT;
   - в TR можно загрузить только сегмент TSS.
2. Обращение к памяти:
   - никакая команда не может писать в сегмент кода;
   - никакая команда не может писать в сегмент данных, защищенный от записи;
   - никакая команда не может читать из сегмента кода, защищенного от чтения;
   — нельзя обращаться к памяти, если селектор в сегментном регистре нулевой.
3. Исполнение команды, использующей селектор в качестве операнда:
   - дальние CALL и JMP могут выполняться только в сегмент кода, шлюз вызова, шлюз задачи или сегмент TSS;
   — команда LLDT может обращаться только к сегменту LDT;
   - команда LTR может обращаться только к сегменту TSS;
   - команда LAR может обращаться только к сегментам кода и данных, шлюзам вызова и задачи, LDT и TSS;
   - команда LSL может обращаться только к сегментам кода, данных, LDT и TSS;
   — элементами IDT могут быть только шлюзы прерываний, ловушек и задач.
4. Некоторые внутренние операции:
   - при переключении задач целевой дескриптор может быть только TSS или шлюзом задачи;
   - при передаче управления через шлюз сегмент, на который шлюз указывает, должен быть сегментом кода (или TSS для шлюза задачи);
   - при возвращении из вложенной задачи селектор в поле связи TSS должен быть селектором сегмента TSS.

10.7.3. Проверка привилегий

Все неравенства здесь арифметические, то есть A > B означает, что уровень привилегий A меньше, чем B:

- при загрузке регистра DS, ES, FS или GS должно выполняться условие: DPL ⩾ max(RPL,CPL);

а при загрузке регистров SS должно выполняться условие: DPL = CPL = RPL;

- при дальних JMP, CALL, RET на неподчиненный сегмент кода должно выполняться условие: DPL = CPL (RPL игнорируется);
- при дальних JMP, CALL, RET на подчиненный сегмент кода должно выполняться условие: CPL ⩾ DPL. При этом CPL не изменяется;
- при дальнем CALL на шлюз вызова должны выполняться условия: CPL < DPL шлюза, RPL ⩽ DPL шлюза, CPL ⩾ DPL сегмента;
- при дальнем JMP на шлюз вызова должны выполняться условия: CPL ⩽ DPL шлюза, RPL ⩽ DPL шлюза, CPL > DPL сегмента, если он подчиненный, CPL = DPL сегмента, если он неподчиненный.

При вызове процедуры через шлюз на неподчиненный сегмент кода *с* другим уровнем привилегий процессор выполняет переключение стека. В сегменте TSS текущей задачи всегда хранятся значения SS:ESP для стеков уровней привилегий О, 1 и 2 (но не стек для уровня привилегий 3, потому что нельзя выполнять передачу управления на уровень 3, кроме как при помощи команд RET/IRET). При переключении стека в новый стек помещаются, до обратного адреса, параметры (их число указано в дескрипторе шлюза вызова), флаги или код ошибки (в случае INT), старые значения SS:ESP, которые команда RET/IRET использует для обратного переключения. То, что надо выполнить возврат из процедуры, RET определяет так: RPL селектора, оставленного в стеке, больше (менее привилегированный), чем CPL.

Даже если операционная система не поддерживает многозадачность, она должна оформить сегмент TSS с действительными SS:ESP для стеков всех уровней, если она собирается использовать уровни привилегий.

10.7.4. Выполнение привилегированных команд

1. Команды LGDT, LLDT, LTR, LIDT, MOV CRn, LMSW, CLTS, MOV DRn, INVD, WBINVD, INVLPG, HLT, RDMSR, WRMSR, RDPMC, RDTSC, SYSEXIT могут выполняться, только если CPL = 0 (хотя биты PCE и TSD сегмента CR4 разрешают использование команд RDPMC и RDTSC с любого уровня).
2. Команды LLDT, SLDT, LTR, STR, LSL, LAR, VERR, VERW и ARPL можно выполнять только в защищенном режиме - в реальном и V86 возникает исключение #UD.
3. Команды CLI и STI выполняются, только если CPL ⩽ IOPL (IOPL - это двухбитная область в регистре флагов). Если установлен бит PVI в регистре CR4, эти команды выполняются с любым CPL, но управляют флагом VIE а не IE
4. Команды IN, OUT, INSB, INSW, INSD, OUTSB, OUTSW, OUTSD выполняются, только если CPL ⩽ IOPL и если бит в битовой карте ввода-вывода, соответствующий данному порту, равен нулю. (Эта карта - битовое поле в сегменте TSS, каждый бит которого отвечает за один порт ввода-вывода. Признаком ее конца служит слово, в котором все 16 бит установлены в 1.)

10.7.5. Защита на уровне страниц

1. Обращение к странице памяти с битом U в атрибуте страницы или таблицы страниц, равным нулю, приводит к исключению #PF, если CPL = 3.
2. Попытка записи в страницу с битом W в атрибуте страницы или таблицы страниц, равным нулю, с CPL = 3 приводит к исключению #PE
3. Попытка записи в страницу с битом W в атрибуте страницы или таблицы страниц, равным нулю, если бит WP в регистре CRO равен 1, приводит к исключению #PE

10.8. Управление задачами

Следующий очень важный механизм, действующий только в защищенном режиме, - многозадачность. Задача - это элемент работы, которую процессор может исполнять, запустить или отложить. Задачи используют для выполнения программ, процессов, обработчиков прерываний и исключений, ядра операционной системы и пр. Любая программа, выполняющаяся в защищенном режиме, должна осуществляться как задача (хотя мы пока игнорировали это требование). Процессор предоставляет средства для сохранения состояния задачи, запуска задачи и передачи управления из одной задачи в другую.

Задача состоит из сегмента состояния задачи (TSS), сегмента кода, одного или нескольких (для разных уровней привилегий) сегментов стека и одного или нескольких сегментов данных. Она определяется селектором своего сегмента TSS. Когда задача выполняется, ее селектор TSS (вместе с дескриптором в скрытой части) загружен в регистр TR процессора.

Запуск задачи осуществляется при помощи команды GALL или JMP на сегмент TSS или на шлюз задачи, а также при запуске обработчика прерывания или исключения, который описан как шлюз задачи. При этом автоматически осуществляется переключение задач. Состояние текущей задачи записывается в ее TSS, состояние вызываемой задачи считывается из ее TSS, и управление передается на новые CS:EIP. Если задача не была запущена командой JMP, селектор сегмента TSS старой задачи сохраняется в TSS новой и устанавливается флаг NT, так что следующая команда IRET выполнит обратное переключение задач.

Задача не может вызываться рекурсивно. В дескрипторе TSS-задачи, которая была запущена, но не была завершена, тип изменяется на «занятый TSS» и переход на такой TSS невозможен.

Задача может иметь собственную таблицу дескрипторов (LDT) и полный комплект собственных таблиц страниц, так как регистры LDTR и CR3 входят в состояние задачи.

10.8.1. Сегмент состояния задачи

Сегмент состояния задачи (TSS) - это структура данных, в которой сохраняется вся информация о задаче, если ее выполнение временно прерывается.

TSS имеет следующую структуру:

+00h: 4 байта - селектор предыдущей задачи (старшее слово содержит нули - здесь и для всех остальных селекторов)
+04h: 4 байта - ESP для CPL = O
+08h: 4 байта - SS для CPL = O
+0Ch: 4 байта - ESP для CPL = 1
+10h: 4 байта - SS для CPL = 1
+14h: 4 байта – ESP для CPL = 2
+18h: 4 байта - SS для CPL = 2
+1Ch: 4 байта – CR3
+20h: 4 байта – EIP
+24h: 4 байта - EFLAGS
+28h: 4 байта – EAX
+2Ch: 4 байта - ECX
+30h: 4 байта - EDX
+34h: 4 байта – EBX
+38h: 4 байта – ESP
+3Ch: 4 байта - EBP
+40h: 4 байта - ESI
+44h: 4 байта – EDI
+48h: 4 байта - ES
+4Ch: 4 байта - CS
+50h: 4 байта - SS
+54h: 4 байта - DS
+58h: 4 байта - FS
+5Ch: 4 байта - GS
+60h: 4 байта - LDTR
+64h: 2 байта - слово флагов задачи
бит 0 - флаг T: вызывает #DB при переключении на задачу
остальные биты не определены и равны нулю
+66h: 2 байта - адрес битовой карты ввода-вывода. Это 16-битное смещение от начала TSS, по которому начинается битовая карта разрешения ввода-вывода (см. разделы 10.7.4 и 10.9.2) и заканчивается битовая карта перенаправления прерываний (см. раздел 10.9.1) данной задачи.

TSS является полноценным сегментом и описывается сегментным дескриптором, формат которого мы рассматривали раньше (см. раздел 10.4.3). Кроме того, лимит TSS не может быть меньше 67h — обращение к такому дескриптору приводит к исключению #TS. Размер TSS может быть больше, если в него входят битовые карты ввода-вывода и перенаправления прерываний и если операционная система хранит в нем дополнительную информацию. Дескриптор TSS способен находиться только в GDT - попытка загрузить его из LDT вызывает исключение #GP. Для передачи управления задачам удобнее использовать дескрипторы шлюза задачи, которые можно помещать как в GDT, так и в LDT или IDT.

70.8.2. *Переключение задач*

Переключение задач осуществляется, если:

- текущая задача выполняет дальний JMP или CALL на шлюз задачи или прямо на TSS;
- текущая задача выполняет IRET, если флаг NT равен 1;
- происходит прерывание или исключение, в качестве обработчика которого в IDT записан шлюз задачи.

При переключении процессор выполняет следующие действия:

1. Для команд CALL и JMP проверяет привилегии (CPL текущей задачи и RPL селектора новой задачи не могут быть больше, чем DPL шлюза или TSS, на который передается управление).
2. Проверяется дескриптор TSS (его бит присутствия и лимит).
3. Проверяется, что новый TSS, старый TSS и все дескрипторы сегментов находятся в страницах, отмеченных как присутствующие.
4. Сохраняется состояние задачи.
5. Загружается регистр TR. Если на следующих шагах происходит исключение, его обработчику придется доделывать переключение задач, вместо того чтобы повторять ошибочную команду.
6. Тип новой задачи в дескрипторе изменяется на занятый, и флаг TS устанавливается в CR0.
7. Загружается состояние задачи из нового TSS: LDTR, CR3, EFLAGS, EIP, регистры общего назначения и сегментные регистры.

Если переключение задачи вызывается командами JUMP, CALL, прерыванием или исключением, селектор TSS предыдущей задачи записывается в поле связи новой задачи и устанавливается флаг NT. Если флаг NT установлен, команда IRET выполняет обратное переключение задач.

При любом запуске задачи ее тип изменяется в дескрипторе на занятый. Попытка вызвать такую задачу приводит к #GP. Сделать задачу снова свободной можно, только завершив ее командой IRET или переключившись на другую задачу командой JMP.

На следующем примере покажем, как создавать задачи и переключаться между ними.

```
; pm4.asm
; Пример программы, выполняющей переключение задач.
; Запускает две задачи, передающие управление друг другу 80 раз, задачи выводят
; на экран символы ASCII с небольшой задержкой.
; Компиляция:
; TASM:
;  tasm /m pm4.asm
;  tlink /x /3 pm4.obj
; WASM:
;  wasm pm4.asm
;  wlink file pm4.obj form DOS
```

```
; MASM:
;  ml /c pm4.asm
;  link pm4.obj,,NUL,,,

        .386p
RM_seg segment para public "CODE" use16
        assume cs:RM_seg,ds:PM_seg,ss:stack_seg
start:
; Подготовить сегментные регистры.
        push    PM_seg
        pop     ds
; Проверить, не находимся ли мы уже в PM.
        mov     eax,cr0
        test    al,1
        jz      no_V86
; Сообщить и выйти.
        mov     dx,offset v86_msg
err_exit:
        push    cs
        pop     ds
        mov     ah,9
        int     21h
        mov     ah,4Ch
        int     21h

; Убедиться, что мы не под Windows.
no_V86:
        mov     ax,1600h
        int     2Fh
        test    al,al
        jz      no_windows
; Сообщить и выйти.
        mov     dx,offset win_msg
        jmp     short err_exit

; Сообщения об ошибках при старте.
v86_msg         db      "Процессор в режиме V86 - нельзя переключиться в PM$"
win_msg         db      "Программа запущена под Windows - нельзя перейти в кольцо 0$"

; Итак, мы точно находимся в реальном режиме.
no_windows:
; Очистить экран.
        mov     ax,3
        int     10h
; Вычислить базы для всех дескрипторов сегментов данных.
        xor     eax,eax
        mov     ax,RM_seg
        shl     eax,4
        mov     word ptr GDT_16bitCS+2,ax
        shr     eax,16
        mov     byte ptr GDT_16bitCS+4,al
```

```
        mov     ax,PM_seg
        shl     eax,4
        mov     word ptr GDT_32bitCS+2,ax
        mov     word ptr GDT_32bitSS+2,ax
        shr     eax,16
        mov     byte ptr GDT_32bitCS+4,al
        mov     byte ptr GDT_32bitSS+4,al
; Вычислить линейный адрес GDT.
        xor     eax,eax
        mov     ax,PM_seg
        shl     eax,4
        push    eax
        add     eax,offset GDT
        mov     dword ptr gdtr+2,eax
; Загрузить GDT.
        lgdt    fword ptr gdtr
; Вычислить линейные адреса сегментов TSS наших двух задач.
        pop     eax
        push    eax
        add     eax,offset TSS_0
        mov     word ptr GDT_TSS0+2,ax
        shr     eax,16
        mov     byte ptr GDT_TSS0+4,al
        pop     eax
        add     eax,offset TSS_1
        mov     word ptr GDT_TSS1+2,ax
        shr     eax,16
        mov     byte ptr GDT_TSS1+4,al
; Открыть A20.
        mov     al,2
        out     92h,al
; Запретить прерывания.
        cli
; Запретить NMI.
        in      al,70h
        or      al,80h
        out     70h,al
; Переключиться в PM.
        mov     eax,cr0
        or      al,1
        mov     cr0,eax
; Загрузить CS.
                db      66h
                db      OEAh
                dd      offset PM_entry
                dw      SEL_32bitCS

RM_return:
; Переключиться в реальный режим RM.
        mov     eax,cr0
```

```
        and     al,0FEh
        mov     cr0,eax
; Сбросить очередь предвыборки и загрузить CS.
        db      0EAh
        dw      $+4
        dw      RM_seg
; Настроить сегментные регистры для реального режима.
        mov     ax,PM_seg
        mov     ds,ax
        mov     es,ax
        mov     ax,stack_seg
        mov     bx,stack_l
        mov     ss,ax
        mov     sp,bx
; Разрешить NMI.
        in      al,70h
        and     al,07FH
        out     70h,al
; Разрешить прерывания.
        sti
; Завершить программу.
        mov     ah,4Ch
        int     21h
RM_seg ends

PM_seg segment para public "CODE" use32
        assume  cs:PM_seg

; Таблица глобальных дескрипторов.
GDT             label   byte
                db      8 dup(0)
GDT_flatDS      db      0FFh,0FFh,0,0,0,10010010b,11001111b,0
GDT_16bitCS     db      0FFh,0FFh,0,0,0,10011010b,0,0
GDT_32bitCS     db      0FFh,0FFh,0,0,0,10011010b,11001111b,0
GOT_32bitSS     db      0FFh,0FFh,0,0,0,10010010b,11001111b,0
; Сегмент TSS задачи 0 (32-битный свободный TSS).
GDT_TSS0        db      067h,0,0,0,0,10001001b,01000000b,0
; Сегмент TSS задачи 1 (32-битный свободный TSS).
GDT_TSS1        db      067h,0,0,0,0,10001001b,01000000b,0
gdt_size = $-GDT
gdtr            dw      gdt_size-1              ; Размер GDT.
                dd      ?                       ; Адрес GDT.
; Используемые селекторы.
SEL_flatDS      equ     001000b
SEL_16bitCS     equ     010000b
SEL_32bitCS     equ     011000b
SEL_32bitSS     equ     100000b
SEL_TSS0        equ     101000b
SEL_TSS1        equ     110000b
```

```
; Сегмент TSS_0 будет инициализирован, как только мы выполним переключение
; из нашей основной задачи. Конечно, если бы мы собирались использовать
; несколько уровней привилегий, то нужно было бы инициализировать стеки.
TSS_0           do      68h dup(0)
; Сегмент TSS_1. В него будет выполняться переключение, поэтому надо
; инициализировать все, что может потребоваться:
TSSJ            dd      0,0,0,0,0,0,0,0                 ; Связь, стеки, CR3
                dd      offset task_1                   ; EIP.
; Регистры общего назначения.
                dd      0,0,0,0,0,stack_12,0,0,0B8140h  ; (ESP и EDI)
; Сегментные регистры.
                dd      SEL_flatDS,SEL_32bitCS,SEL_32bitSS,SEL_flatDS,0,0
                dd      0                               ; LDTR.
                dd      0                               ; Адрес таблицы ввода-
                                                        ; вывода.

; Точка входа в 32-битный защищенный режим.
PM_entry:
; Подготовить регистры.
        xor     eax,eax
        mov     ax,SEL_flatDS
        mov     ds,ax
        mov     es,ax
        mov     ax,SEL_32bitSS
        mov     ebx,stack_l
        mov     ss,ax
        mov     esp,ebx
; Загрузить TSS задачи 0 в регистр TR.
        mov     ax,SEL_TSS0
        ltr     ax
; Только теперь наша программа выполнила все требования к переходу
; в защищенный режим.

        xor     eax,eax
        mov     edi,0B8000h                     ; DS:EDI - адрес начала экрана.
task_0:
        mov     byte ptr ds:[edi],al            ; Вывести символ AL на экран.
; Дальний переход на TSS задачи 1.
                db      0EAh
                dd      0
                dw      SEL_TSS1
        add     edi,2                           ; DS:EDI - адрес следующего
                                                ; символа.
        inc     al                              ; AL - код следующего символа.
        cmp     al,80                           ; Если это 80,
        jb      task_0                          ; выйти из цикла.
; Дальний переход на процедуру выхода в реальный режим.
                db      0EAh
                dd      offset RM_return
                dw      SEL_16bitCS
```

```
; Задача 1.
task_1:
        mov     byte ptr ds:[edi],al     ; Вывести символ на экран.
        inc     al                       ; Увеличить код символа.
        add     edi,2                    ; Увеличить адрес символа.
; Переключиться на задачу 0.
                db      0EAh
                dd      0
                dw      SEL_TSS0
; Сюда будет приходить управление, когда задача 0 начнет выполнять переход
; на задачу 1 во всех случаях, кроме первого.
        mov     ecx,02000000h            ; Небольшая пауза, зависящая от скорости
        loop    $                        ; процессора.
        jmp     task_1

PM_seg ends

stack_seg segment para stack "STACK"
stack_start     db      100h dup(?)      ; Стек задачи 0.
stack_l = $-stack_start
stack_task2     db      100h dup(?)      ; Стек задачи 1.
stack_12 = $-stack_start
stack_seg       ends
        end     start
```

Чтобы реализовать многозадачность в реальном времени на нашем примере, достаточно создать обработчик прерывания системного таймера IRQO в виде отдельной (третьей) задачи и поместить в ЮТ шлюз этой задачи. Текст обработчика для нашего примера мог быть крайне простым:

```
task_3: ; Это отдельная задача - не нужно сохранять регистры!
        mov     al,20h
        out     20h,al
        jmp     task_0
        mov     al,20h
        out     20h,al
        jmp     task_1
        jmp     task_3
```

Но при вызове обработчика прерывания старая задача помечается как занятая в GDT и повторный JMP на нее приведет к ошибке. Вызов задачи обработчика прерывания, так же как и вызов задачи командой CALL, подразумевает, что она завершится командой IRET. Именно команду IRET проще всего вызвать для передачи управления из такого обработчика - достаточно лишь подменить селектор вызвавшей нас задачи в поле связи и выполнить IRET.

```
task_3: ; При инициализации DS должен быть установлен на PM_seg.
        mov     al,20h
        out     20h,al
        mov     word ptr TSS_3,SEL_TSS0
        iret
        mov     al,20h
```

```
out     20h,al
mov     word ptr TSS_3,SEL_TSS1
iret
jmp     task_3
```

Единственное дополнительное изменение, которое нужно внести, - инициализировать дескриптор TSS задачи task_1 как уже занятый, поскольку управление на него будет передаваться командой IRET, что, впрочем, не вызывает никаких проблем.

Помните, что во вложенных задачах команда IRET не означает конца программы - следующий вызов задачи всегда передает управление на очередную команду после IRET.

10.9. Режим виртуального 8086

Режим V86 - это задача, исполняющаяся в защищенном режиме, в которой флаг VM регистра EFLAGS равен единице. Внутри задачи процессор ведет себя так, как если бы он находился в реальном режиме, за исключением того, что прерывания и исключения передаются обработчикам защищенного режима вне ее (кроме случая, когда используется карта перенаправления прерываний).

Программы не могут изменить флаг VM. Его допускается указать, только записав образ EFLAGS с установленным VM при создании TSS новой задачи и затем переключившись на нее. Кроме этой задачи для нормальной реализации V86 требуется монитор режима (VMM) - модуль, который выполняется с CPL = O и обрабатывает прерывания, исключения и обращения к портам ввода-вывода из задачи V86, осуществляя фактически эмуляцию всего компьютера.

Чтобы выполнять в системе сразу несколько V86-задач, применяется страничная адресация. Каждая V86-задача использует ровно один мегабайт линейного адресного пространства, который можно отобразить на любую область физического.

Процессор переключается в V86 в трех ситуациях:

- при переключении в задачу, TSS которой содержит установленный флаг VM;
- при выполнении команды IRET, если NT = 0 и копия EFLAGS в стеке содержит установленный флаг VM;
- при выполнении команды IRET, если NT = 1 и копия EFLAGS в TSS содержит установленный флаг VM.

10.9.1. Прерывания в V86

Если происходит прерывание или исключение в режиме V86, процессор анализирует биты IOPL регистра флагов, бит VME регистра CR4 (Pentium и выше) и соответствующий бит из карты перенаправления прерываний данной задачи (только если VME = 1).

Эта карта - 32-байтное поле, находящееся в регистре TSS данной задачи, на первый байт за концом которой указывает смещение в TSS по адресу +66h. Каждый из 256 бит этого поля соответствует одному номеру прерывания. Если он установлен в 1, прерывание должно подготавливаться обработчиком из IDT в защищенном режиме, если он 0 - то 16-битным обработчиком из реального режима.

Если VME = 0, прерывание обрабатывается (через IDT) при условии, что IOPL = 3, иначе вызывается исключение #GP.

Если бит VME = 1 и IOPL = 3, обработка прерывания определяется битом из битовой карты перенаправления прерываний.

Если VME = 1, IOPL < 3 и бит в битовой карте равен единице, вызывается обработчик из ШТ.

Если VME = 1, IOPL < 3 и бит в битовой карте равен нулю, происходит следующее:

- если VIF = 0 или если VIF = 1, но произошло исключение или NMI - вызывается обработчик из реального режима;
- если VIF = 1 и произошло аппаратное прерывание - вызывается обработчик #GP из защищенного режима, который должен обработать прерывание, установить флаг VIP в копии EFLAGS в стеке и вернуться в V86;
- если VIP = 1 и VIF = 0 из-за выполненной в V86 команды CLI - вызывается обработчик #GP из реального режима, который должен обнулить VIF и VIP в копии EFLAGS в стеке.

Бит VIF - это флаг для облегчения поддержки команд CLI и STI в задачах V86. Если в регистре CR4 установлен бит VME, команды CLI/STI изменяют значение именно этого флага, оставляя IF нетронутым для того, чтобы операционная система могла обрабатывать прерывания и управлять другими задачами.

При вызове обработчика, находящегося в защищенном режиме, из реального режима в стек нулевого уровня привилегий помещаются GS, FS, DS, ES, SS, EFLAGS, CS, EIP и код ошибки для некоторых исключений в этом порядке, а флаги VM, TF и IF обнуляются, если вызывается шлюз прерывания.

10.9.2. Ввод-вывод в V86

В режиме V86 текущий уровень привилений, CPL, всегда равен трем. В соответствии с правилами защиты выполнение команд CLI, STI, PUSHF, POPF, INT и IRET приводит к исключению #GP, если IOPL < 3. Однако команды IN, OUT, INS, OUTS, чувствительные к IOPL в защищенном режиме, в V86 управляются битовой картой ввода-вывода, расположенной в TSS задачи. Если бит, соответствующий порту, установлен в 1, обращение к нему из V86-задачи приводит к исключению #GP; если бит сброшен - команды работы с портами ввода-вывода выполняются.

Мы не будем рассматривать пример программы, реализующей режим V86, из-за его размеров. Практически все из того, что необходимо для его создания (защищенный режим, обработка прерываний, страничная адресация и переключение задач), мы уже обсудили.

Глава 11. Программирование на в среде UNIX

Операционная система MS DOS, получившая дальнейшее развитие в виде Windows, долгое время была практически единственной ОС для персональных компьютеров на базе процессоров Intel. Но с течением времени мощность процессоров выросла настолько, что для них стало возможным работать под управлением операционных систем класса UNIX, использовавшихся обычно на более мощных компьютерах других компаний. В настоящее время существует свыше двадцати операционных систем для Intel, представляющих те или иные диалекты UNIX. Мы рассмотрим самые популярные из них:

- Linux – бесплатно распространяемая операционная система, соединяющая в себе особенности двух основных типов UNIX-систем (System V и BSD) приблизительно в равной мере. В ней много отличий и отступлений от любых стандартов, принятых для UNIX, но они более эффективны;
- FreeBSD - бесплатно распространяемая операционная система, представляющая вариант BSD UNIX. Считается наиболее стабильной из UNIX-систем для Intel;
- Solaris/x86 - коммерческая операционная система компании Sun Microsystems, представляющая вариант System V UNIX, изначально созданная для компьютеров Sun, существует в версии для Intel 80x86. Распространяется бесплатно с образовательными целями.

Несмотря на то что при программировании для UNIX обычно употребляется исключительно язык C, пользоваться ассемблером в этих системах можно, и даже очень просто. Программы в UNIX выполняются в защищенном режиме с моделью памяти flat и могут вызывать любые функции из библиотеки libc или других библиотек точно так же, как это делают программы на C. Конечно, круг задач, для которых имеет смысл использовать ассемблер в UNIX, ограничен. Если вы не занимаетесь разработкой ядра операционной системы или, например, эмулятора DOS, практически все можно сделать и на C, но иногда нужно создать что-то особенное. Написать процедуру, выполняющую что-то как можно быстрее (например, воспроизведение звука из файла в формате MP3), или программу, использующую память более эффективно (хотя это часто можно повторить на C), или программу, применяющую возможности нового процессора, поддержка которого еще не добавлена в компилятор, оказывается очень просто (если вы знаете ассемблер для UNIX).

11.1. Синтаксис AT&T

Проблема в том, что ассемблер для UNIX кардинально отличается от того, что рассматривалось в этой книге до сих пор. В то время как основные ассемблеры для MS DOS и Windows используют синтаксис, предложенный компанией Intel, изобилующий неоднозначностями, часть которых решается за счет использования поясняющих операторов типа byte ptr, word ptr или dword ptr, а часть не решается вообще (все те случаи, когда приходится указывать код команды вручную), в UNIX с самого начала используется вариант универсального синтаксиса AT&T, синтаксис SysV/386, который специально создавался с целью устранения неоднозначностей в толковании команд. Вообще говоря, существует и ассемблер для DOS/Windows, использующий AT&T-синтаксис, - это gas, входящий в набор средств разработки DJGPP, а также ассемблер, использующий Intel-синтаксис и способный создавать объектные файлы в формате ELF, применяемом в большинстве UNIX-систем, - это бесплатно распространяемый в Internet ассемблер NASM. Мы будем рассматривать только ассемблеры, непосредственно входящие в состав операционных систем, то есть те, которые вызываются стандартной командой as.

77.7.7. *Основные правила*

Итак, в ассемблере AT&T в качестве допустимых символов текста программы рассматриваются только латинские буквы, цифры и символы % (процент) $ (доллар), * (звездочка) , . (точка), , (запятая) и _ (подчеркивание). Помимо них существуют символы начала комментария, отличающиеся для разных ассемблеров и для комментария размером в целую строку или правую часть строки. Любые другие символы, кроме кавычек, двоеточия, пробела и табуляции, если они не часть комментария или не заключены в кавычки, считаются ошибочными.

Если последовательность допустимых символов строки не начинается со специального символа или цифры и не заканчивается двоеточием - это команда процессора:

```
// Остановить процессор.
                hlt
```

Когда последовательность допустимых символов начинается с символа % - это название регистра процессора:

```
// Поместить в стек содержимое регистра EAX.
                pushl  %eax
```

Если последовательность начинается с символа $ - это непосредственный операнд:

```
// Поместить в стек 0, число 10h и адрес переменной variable.
                pushl   $0
                 pushl   $0x10
                pushl   $variable
```

В том случае, когда последовательность символов начинается с точки, - это директива ассемблера:

```
.align  2
```

Если последовательность символов, с которой начинается строка, заканчивается двоеточием - это метка (внутренняя переменная ассемблера, значение которой соответствует адресу в указанной точке):

```
eternal_loop:   jmp eternal_loop
variable:       .byte 7
```

Метки, состоящие из одной цифры от 0: до 9:, используются как локальные - обращение к метке 1f соответствует обращению к ближайшей из меток 1: вперед по тексту программы; обращение к метке 4b соответствует обращению к ближайшей из меток 4: назад по тексту программы.

Одни и те же метки могут использоваться без ограничений и в качестве цели для команды перехода, и в качестве переменных.

Специальная метка . (точка) всегда равна текущему адресу (аналогично $ в ассемблерах для DOS/Windows).

Если число начинается с * - это абсолютный адрес (для команд jmp и call), в противном случае - относительный.

Если метка начинается с символа * - выполняется косвенный переход.

11.1.2. Запись команд

Названия команд, не принимающих операнды, совпадают с названиями, принятыми в синтаксисе Intel:

```
nop
```

К названиям команд, которые имеют операнды, добавляются суффиксы, отражающие размер операндов:

- b – байт;
- w– слово;
- a l - двойное слово;
- q - учетверенное слово;
- s – 32-битное число с плавающей запятой;
- a l - 64-битное число с плавающей запятой;
- t – 80-битное число с плавающей запятой.

```
// mov byte ptr variable,0
                movb    $0,variable
// fild qword ptr variable
                fildq   variable
```

Команды, принимающие операнды разных размеров, требуют указания двух суффиксов, сначала суффикса источника, а затем приемника:

```
// movsx        edx,al
                movsbl  %al,%edx
```

Команды преобразования типов имеют в AT&T названия из четырех букв — C), размер источника, T и размер приемника:

```
// cbw
                cbtw
// cwde
                cwtl
// cwd
                cwtd
// cdq
                cltd
```

Но многие ассемблеры понимают и принятые в Intel формы для этих четырех команд.

Дальние команды передачи управления (jmp, call, ret) отличаются от ближних префиксом l:

```
// call far 0007:00000000
                lcall   $7,$0
// retf 10
                lret    $10
```

Если команда имеет несколько операндов, операнд-источник всегда записывается первым, а приемник - последним, то есть в точности наоборот по сравнению с Intel-синтаксисом:

```
// mov ax,bx
                movw    %bx,%ax
// imul eax,ecx,16
                imull   $16,%ecx,%eax
```

У всех префиксов перед командой, для которой данный префикс предназначен, есть имена, как у обычных команд. Имена префиксов замены сегмента - segcs, segds, segss, segfs, seggs; имена префиксов изменения разрядности адреса и операнда - addr16 и data16:

```
                segfs
                movl    variable,%eax
                rep
                stosd
```

Кроме того, префикс замены сегмента будет включен автоматически, если используется оператор : в контексте операнда:

```
                movl    %fs:variable,%eax
```

11.1.3. Адресация

Регистровый операнд всегда начинается с символа **%**:

```
// xor edx,edx
                xorl    %eax,%eax
```

Непосредственный операнд всегда начинается с символа **$**:

```
// mov edx,offset variable
                • movl    $variable,%edx
```

Косвенная адресация использует немодифицированное имя переменной:

```
// push dword ptr variable
                  pushl    variable
```

Более сложные способы адресации удобнее рассматривать как варианты максимально сложного способа - по базе и индексированием, и сдвигом:

```
// mov eax,base_addr[ebx+edi*4] (наиболее общий случай)
                  movl     base_addr(%ebx,%edi,4),%eax
// lea eax,[eax+eax*4]
                  leal     (%eax,%eax,4),%eax
// mov ax,word ptr [bp-2]
                  movw      -2(%ebp),%ax
// mov edx,dword ptr [edi*2]
                  movl     (,%edi,2),%edx
```

11.2. Операторы ассемблера

Ассемблеры для UNIX, как и для DOS, могут вычислять значения выражений в момент компиляции, например:

```
// Поместить в EAX число 320*200.
                  movl     $320*$200,%eax
```

В этих выражениях встречаются следующие операторы.

11.2.1. Префиксные, или унарные, операторы

- (минус) – отрицательное число
- (тильда) - «логическое НЕ»

77.2.2. Инфиксные, или бинарные, операторы

Высшего приоритета:

* - умножение;

/ - целочисленное деление;

% - остаток;

< или << - сдвиг влево;

> или >> - сдвиг вправо.

Среднего приоритета:

| - побитовое «ИЛИ»;

& - побитовое «И»;

^ - побитовое «исключающее ИЛИ»;

! - побитовое «ИЛИ-НЕ» (логическая импликация).

Низшего приоритета:

\+ - сложение;

—— вычитание.

11.3. Директивы ассемблера

Все директивы ассемблера в UNIX всегда начинаются с символа . (точка). Из-за большого количества операционных систем и ассемблеров для них возникли многочисленные часто встречающиеся директивы. Рассмотрим наиболее полезные,

11.3.1. Директивы определения данных

Эти директивы эквивалентны директивам db, dw, dd, df и т. п., применяющимся в ассемблерах для DOS/Windows. Основное отличие здесь состоит в том, чтобы дать имя переменной, значение которой определяется такой директивой; в ассемблерах для UNIX обязательно надо ставить полноценную метку, заканчивающуюся двоеточием.

Байты:

```
.byte выражение...
```

Слова:

```
.word выражение... или .hword выражение... или .short  выражение...
```

Двойные слова:

```
.int выражение... или .long выражение...
```

Учетверенные слова (8-байтные переменные):

```
.quad выражение...
```

16-байтные переменные (окта-слова):

```
.octa выражение...
```

32-битные числа с плавающей запятой:

```
.float число... или .single число...
```

64-битные числа с плавающей запятой:

```
.double число...
```

80-битные числа с плавающей запятой:

```
.tfloat число...
```

Строки байтов:

```
.ascii строка...
```

Строки байтов с автоматически добавляемым нулевым символом в конце:

```
.asciz строка... или .string строка
```

Блоки повторяющихся данных:

```
.skip размер, значение или .space размер, значение // Заполняет области
                                                   // памяти указанного
                                                   // размера байтами
                                                   // с заданным значением.
```

```
.fill повтор, размер, значение    // Заполняет область памяти значениями
                                  // заданного размера (0-8 байт) указанное
                                  // число раз. По умолчанию размер
                                  // принимается равным 1, а значение - 0.
```

Неинициализированные переменные:

```
.lcomm символ, длина, выравнивание // Зарезервировать указанное число байтов
                                   // для локального символа в секции .bss.
```

77.3.2. Директивы управления символами

Присвоение значений символам:

```
.equ символ, выражение             // Присваивает символу значение выражения.
.equiv символ, выражение           // То же, что и .equ, но выдает сообщение
                                   // об ошибке, если символ определен.
.set символ, выражение             // То же, что и .equ, но можно повторять
                                   // несколько раз. Обычно, впрочем, удобнее
                                   // написать просто «символ = выражение».
```

Управление внешними символами:

```
.globl символ или .global символ  // Делает символ видимым для компоновщика,
                                  // а значит, и для других модулей
                                  // программы.
.extern символ                    // Директива .extern обычно игнорируется -
                                  // все неопределенные символы считаются
                                  // внешними.
.comm символ, длина, выравнивание // Директива эквивалентна .lcomm, но, если
                                  // символ с таким именем определен при
                                  // помощи .lcomm в другом модуле, будет
                                  // использоваться внешний символ.
```

Описание отладочных символов:

```
.def символ                       // Блок описания отладочного символа.
.endef
```

Мы не коснемся описания отладочных символов, так как их форматы сильно различаются между разнообразными операционными системами и разными форматами объектных файлов.

11.3.3. Директивы определения секций

Текст программы делится на секции - кода, данных, неинициализированных данных, отладочных символов и т. д. Секции также могут делиться на подсекции, располагающиеся непосредственно друг за другом, но это редко используется.

```
.data подсекция
```

Следующие команды будут ассемблироваться в секцию данных. Если подсекция не указана, данные ассемблируются в нулевую подсекцию.

```
.textподсекция
```

Следующие команды будут ассемблироваться в секцию кода.

```
.section имя, флаги, @тип или .section "имя", флаги
```

Общее определение новой секции:

- флаги (для ELF):
  - w или #write - разрешена запись;
  - x или #execinstr - разрешено исполнение;
  - a или #alloc - разрешено динамическое выделение памяти (.bss);
- тип (для ELF):
  - @progbits - содержит данные;
  - @nobits - не содержит данных (только занимает место).

11.3.4. Директивы управления разрядностью

```
.code16
```

Следующие команды будут ассемблироваться как 16-битные.

```
.code32
```

Отменяет действие .code16.

11.3.5. Директивы управления программным указателем

```
.align выражение, выражение, выражение
```

Выполняет выравнивание программного указателя до границы, отмеченной первым операндом. Второе выражение указывает, какими байтами заполнять пропускаемый участок (по умолчанию - ноль для секций данных и 90h для секций кода). Третье выражение задает максимальное число байтов, которые может пропустить эта директива.

В отдельных системах первое выражение - не число, кратным которому должен стать указатель, а число битов в указателе, которые должны стать нулевыми.

```
.orgд новое значение, заполнение
```

Увеличивает программный указатель до нового значения в пределах текущей секции. Пропускаемые байты заполняются указанными значениями (по умолчанию - нулями).

11.3.6. Директивы управления листингом

Запретить листинг:

```
.nolist
```

Разрешить листинг:

```
.list
```

Конец страницы:

```
.eject
```

Размер страницы (60 строк, 200 столбцов по умолчанию):

```
.psize строки, столбцы
```

Заголовок листинга:

```
.titleтекст
```

Подзаголовок:

```
.sbttl текст
```

11.3.7. Директивы управления ассемблированием

Включить текст другого файла в программу:

```
.include файл
```

Ассемблировать блок, если выполняется условие или определен либо не определен символ:

```
.if выражение
.ifdef символ
.ifndef символ или .ifnotdef символ
.else
.endif
```

Выдать сообщение об ошибке:

```
.err
```

Немедленно прекратить ассемблирование:

```
.abort
```

11.3.8. Блоки повторения

Повторить блок программы указанное число раз:

```
.rept число повторов
.endr
```

Повторить блок программы для всех указанных значений символа:

```
.irp симол, значение...
.endr
```

Повторить блок программы столько раз, сколько байтов в строке, устанавливая символ равным каждому байту по очереди:

```
.irpc символ, строка
.endr
```

Внутри блока повторения на символ можно ссылаться, начиная его с обратной косой черты (то есть как \символ). Например, такой блок

```
.irp param,1,2,3
movl    %st(0),%st(\param)
.endr
```

как и такой

```
.irpc param,123
movl    %st(0),%st(\param)
.endr
```

ассемблируется в:

```
movl    %st(0),%st(1)
movl    %st(0),%st(2)
movl    %st(0),%st(3)
```

11.3.9. Макроопределения

Начало макроопределения:

```
.macro имя, аргументы
```

Конец макроопределения:

```
.endm
```

Преждевременный выход из макроопределения:

```
.exitm
```

Внутри макроопределения обращение к параметру выполняется аналогично блокам повторения, начиная его с обратной косой черты.

Хотя стандартные директивы и включают в себя такие вещи, как блоки повторений и макроопределения, их реализация достаточно упрощена, и при программировании для UNIX на ассемблере часто применяют дополнительные препроцессоры. Долгое время было принято использовать C-препроцессор или M4, и многие ассемблеры даже могут вызывать их автоматически, но в рамках проекта GNU был создан специальный препроцессор для ассемблера - gasp. Он включает различные расширения вариантов условного ассемблирования, построения циклов, макроопределений, листингов, директив определения данных и т. д. Мы не будем заниматься реализацией таких сложных программ, которым может потребоваться gasp, даже не воспользуемся и половиной перечисленных директив, но о существовании этого препроцессора следует помнить.

11.4. Программирование с использованием libc

Все программы для UNIX, написанные на C, постоянно обращаются к различным функциям, находящимся в libc.so или других стандартных или нестандартных библиотеках. Программы и процедуры на ассемблере, естественно, могут делать то же самое. Вызов библиотечной функции выполняется обычной командой call, а передача параметров осуществляется в соответствии с C-конвенцией: параметры помещают в стек справа налево и очищают стек после вызова функции. Единственная сложность здесь состоит в том, что к началу имени вызываемой функции в некоторых системах, например FreeBSD, приписывается символ подчеркивания, в то время как в других (Linux и Solaris) имя не изменяется. Если имена в системе модифицируются, то имена процедур, написанных на ассемблере, включая main(), также должны быть изменены заранее.

Посмотрим на примере программы, выводящей традиционное сообщение **Hello world,** как это делается.

```
// helloelf.s
// Минимальная программа, выводящая сообщение "Hello world".
// Для компиляции в формат ELF.
//
// Компиляция:
// as -o helloelf.o helloelf.s
// Компоновка:
// (пути к файлу crt1.o могут отличаться на других системах)
// Solaris с SunPro C
//  ld -s -o helloelf.sol helloelf.o /opt/SUNWspro/SC4.2/lib/crt1.o -lc
// Solaris с GNU C
//  ld -s -o helloelf.gso helloelf.o
//  /opt/gnu/lib/gcc-lib/i586-cubbi-solaris2.5.1/2.7.2.3.f.1/crt1.o -lc
// Linux
//  ld -s -m elf_i386 -o helloelf.lnx /usr/lib/crt1.o /usr/lib/crti.o
//  -L/usr/lib/gcc-lib/i586-cubbi-linuxlibc1/2.7.2 helloelf.o -lc -lgcc
//  /usr/lib/crtn.o
// или gcc -o helloelf.lnx helloelf.o
                .text
// Код, находящийся в файлах crt*.o, передаст управление на процедуру main
// после настройки всех параметров.
                .globl  main
main:
// Поместить параметр (адрес строки message) в стек.
                pushl   $message
// Вызвать функцию puts (message).
                call    puts
// Очистить стек от параметров.
                popl    %ebx
// Завершить программу.
                ret

                .data
message:
                .string "Hello world\0"
```

В случае с FreeBSD придется внести всего два изменения - добавить символ подчеркивания в начало имен функций puts и main и заменить директиву .string на .ascii, так как версия ассемблера, обычно распространяемого с FreeBSD, .string не понимает.

```
// hellocof.s
// Минимальная программа, выводящая сообщение "Hello world".
// Для компиляции в вариант формата COFF, используемый во FreeBSD 2.2.

// Компиляция для FreeBSD:
// as -o hellocof.o hellocof.s
// ld -s -o hellocof.bsd /usr/lib/crt0.o hellocof.o -lc
```

```
        .text
        .globl  _main
_main:
        pushl   $message
        call    _puts
        popl    %ebx
        ret
        .data
message:
        .ascii  "Hello world\0"
```

Пользуясь этой техникой, можно создавать программы точно так же, как и на C, но выигрыш за счет того, что на ассемблере допускается соптимизировать программу на несколько процентов лучше, чем это сделает компилятор с C (при максимальной оптимизации), окажется небольшим по сравнению с потерей переносимости. Кроме того, при написании любой сколько-нибудь значительной программы целиком на ассемблере мы столкнемся с тем, что, как и в случае с Win32, нам придется создавать собственные включаемые файлы с определениями констант и структур, взятых из включаемых файлов для C. А поскольку эти ассемблеры не умеют работать со структурами данных, необходимо описывать их средствами используемого препроцессора - cpp или т4.

Лучшее применение ассемблера для UNIX (кроме собственно разработки ядра системы) все-таки остается за незначительными процедурами, требующими большой вычислительной мощности, - кодированием, архивированием, преобразованиями типа Фурье, которые не очень сложны и при необходимости могут быть легко переписаны заново на ассемблере для другого процессора или на C.

11.5. Программирование без использования libc

Может оказаться, что программа вынуждена многократно вызывать те или иные стандартные функции из libc в критическом участке, тормозящем выполнение всей программы. В этом случае стоит обратить внимание на то, что многие функции libc на самом деле всего лишь более удобный для языка C интерфейс к системным вызовам, предоставляемым самим ядром операционной системы. Такие операции, как ввод/вывод, вся работа с файловой системой, с процессами, с TCP/IP и т. п., могут выполняться путем передачи управления ядру операционной системы напрямую.

Чтобы осуществить системный вызов, надо передать его номер и параметры на точку входа ядра аналогично функции libc syscall(2). Номера системных вызовов (находятся в файле /usr/include/sys/syscall.h) и способ обращения к точке входа (дальний call по адресу 0007:00000000) стандартизированы SysV/386 ABI, но, например в Linux, используется другой механизм - прерывание 80h, следовательно, получается, что обращение к ядру ОС напрямую делает программу привязанной к конкретной системе. Часть указанных ограничений можно убрать, используя соответствующие #define, но в общем случае выигрыш в скорости

оборачивается еще большей потерей переносимости, чем само применение ассемблера в UNIX.

Посмотрим, как реализуются системные вызовы в представленных примерах:

```
// hellolnx.s
// Программа, выводящая сообщение "Hello world" на Linux без использования libc.
//
// Компиляция:
// as -o hellolnx.o hellolnx.s
// ld -s -o hellolnx hellolnx.o
//
                .text
                .globl  _start
_start:
// Системный вызов #4 "write", параметры в Linux помещают слева направо,
// в регистры %eax, %ebx, %ecx, %edx, %esi, %edi.
                movl    $4,%eax
                xorl    %ebx,%ebx
                incl    %ebx
// %ebx = 1 (идентификатор stdout)
                movl    $message,%ecx
                movl    $message_l,%edx
// Передача управления в ядро системы - прерывание с номером 80h.
                int     $0x80
// Системный вызов й1 "exit" (%eax = 1, %ebx = 0).
                xorl    %eax,%eax
                incl    %eax
                xorl    %ebx,%ebx
                int     $0x80
                hlt
                .data
message:
                .string "Hello world\012"
message_l = .-message
```

Linux является уникальным случаем по отношению к системным вызовам. В более традиционных UNIX-системах - FreeBSD и Solaris - системные вызовы реализованы согласно общему стандарту SysV/386, и различие в программах заключается лишь в том, что ассемблер, поставляемый с FreeBSD, не поддерживает некоторые команды и директивы.

```
// hellobsd.s
// Программа, выводящая сообщение "Hello world" на FreeBSD без использования libc.
//
// Компиляция:
// as -o hellobsd.o hellobsd.s
// ld -s -o hellobsd hellobsd.o
//
```

```
                .text
                .globl  _start
_start:
// Системная функция 4 "write".
// В FreeBSD номер вызова помещают в %eax, а параметры - в стек
// справа налево плюс одно двойное слово.
                pushl   $message_l
// Параметр 4 - длина буфера.
                pushl   $message
// Параметр 3 - адрес буфера.
                pushl   $1
// Параметр 2 - идентификатор устройства.
                movl    $4,%eax
// Параметр 1 - номер функции в eax.
                pushl   %eax
// В стек надо поместить любое двойное слово, но мы поместим номер вызова
// для совместимости с Solaris и другими строгими операционными системами.
// lcall $7,$0 - ассемблер для FreeBSD не поддерживает эту команду.
                .byte   0x9a
                .long   0
                .word   7
// Восстановить стек.
                addl    $16,%esp

// Системный вызов 1 "exit".
                xorl    %eax,%eax
                pushl   %eax
                incl    %eax
                pushl   %eax
// Icall $7,$0
                .byte   0x9A
                .long   0
                .word   7
                hlt
                .data
message:
                .ascii  "Hello world\012"
message_l = .-message
```

И теперь то же самое в Solaris:

```
// hellosol.s
// Программа, выводящая сообщение "Hello world" на Solaris/x86 без использования libc.
//
// Компиляция:
// as -o hellosol.o hellosol.s
// ld -s -o hellosol hellosol.o
//
                .text
                .globl  _start
```

```
_start:
// Комментарии - см. hellobsd.s.
                pushl   $message_l
                pushl   $message .
                movl    $4,%eax
                pushl   %eax
                lcall   $7,$0
                addl    $16,%esp

                xorl    %eax,%eax
                pushl   %eax
                incl    %eax
                pushl   %eax
                Icall   $7,$0
                hlt
                .data
message:
                .string "Hello world\012"
message_l = .-message
```

Конечно, создавая данные программы, мы нарушили спецификацию SysV/386 ABI несколько раз, но лишь потому, что не обращались ни к каким разделяемым библиотекам, это прошло незамеченным. Требования к полноценной программе сильно отличаются в различных операционных системах, и все они выполнены с максимально возможной тщательностью в файлах crt*.o, которые мы подключали в примере с использованием библиотечных функций. Поэтому, если вы не задаетесь целью сделать программу абсолютно минимального размера, гораздо удобнее назвать вашу процедуру main (или _main), добавляя crt*.o и -lc при компоновке.

11.6. Переносимая программа для UNIX

Как вы могли заметить, отличия между программами на ассемблере для разных UNIX-систем сводятся к разнообразным способам вызова системных функций. Кроме того, следует заметить, что номера этих функций могут тоже отличаться. Рассмотрим на примере простой программы - аналога системной команды pwd, как эти различия можно учесть.

Мы начнем создавать программу pwd без использования системных библиотек. В таком виде она может быть полезна как, например, часть мини-UNIX на одной дискете. Как мы позднее убедимся, размер нашей версии pwd станет меньше такой же программы на C (в 50-100 раз) без учета того, что любой программе на C дополнительно нужна библиотека libc.so, занимающая сотни килобайт.

Команда pwd должна Всего лишь вывести на стандартный вывод полное имя текущего каталога. Для этого можно воспользоваться различными алгоритмами:

1. Системный вызов getcwd(), который присутствует в Linux 2.2 (номер 183) и FreeBSD 3.0 (номер 326). Этой функции передается адрес буфера и его размер, а она записывает в буфер полное имя текущего каталога. Данный способ - самый эффективный, и мы в первую очередь будем использовать именно его.

2. Специальный файл /proc/self/cwd в Linux 2.2 является символической ссылкой на текущий каталог того процесса, который проверяет ее.
3. Некоторые интерпретаторы передают запускаемым программам значение текущего каталога в переменной среды PWD. Такой способ - самый ненадежный, потому что под другим shell нашу программу нельзя будет запустить.
4. Классический алгоритм, в котором программа поэтапно исследует содержимое вышележащего каталога (../, затем ../../ и т. д.) и находит запись, совпадающую с предыдущим рассмотренным каталогом (./, ../, и т. д.). Такой алгоритм мы будем использовать, если в системе не поддерживаются ни системный вызов getcwd(), ни специальный файл /proc/self/cwd.

Поскольку имена большинства системных вызовов разнятся, разместим их в отдельном файле config.i, который будет включен в программу директивой .include:

```
// Способ обращения к системному вызову (установить
// SYSCALL_linux в 1, если используется int $0x80, и
// SYSCALL_unix = 1, если используется lcall $7,$0).
SYSCALL_linux = 0
SYSCALL_unix = 1

// Максимальная длина пути (из limits.h).
MAX_PATH = 4096

// Стратегия для pwd:
// 1, если используется системный вызов getcwd(),
PWD_sys = 1
// 1, если используется файл /proc/self/cwd,
PWD_proc = 0
// 1, если используется обычный (переносимый) алгоритм,
PWD_posix = 0
// 1, если используется переменная среды PWD.
PWD_env = 0

// Номера используемых системных вызовов (из sys/syscalls.h):
// exit() - всегда 1.
SYSCALL_EXIT = 1
// write() - всегда 4.
SYSCALL_WRITE = 4
// open() - всегда 5.
SYSCALL_OPEN = 5
// close() - всегда 6.
SYSCALL_CLOSE = 6
// readlink() - 58 на FreeBSD, 85 на Linux.
SYSCALL_READLINK = 58
// readdir() - нет на FreeBSD, 89 на Linux.
SYSCALL_READDIR = 0
// getdents() - нет на FreeBSD, 141 на Linux.
SYSCALL_GETDENTS = 0
// getdirentries() - 196 на FreeBSD, нет на Linux.
SYSCALL_GETDIRENTRIES = 196
```

```
// stat() - 188 на FreeBSD, 106 на Linux.
SYSCALL_STAT = 188
// lstat() - 190 на FreeBSD, 107 на Linux.
SYSCALL_LSTAT = 190
// fstat() - 189 на FreeBSD, 108 на Linux.
SYSCALL_FSTAT = 189
// getcwd() - 326 на FreeBSD 3.0, 183 на Linux 2.2,
// нет на старых версиях.
SYSCALL_GETCWD = 326

// Размеры используемых структур.
// Размер struct dirent:
// 1024 на FreeBSD, 266 на Linux.
SIZE_DIRENT = 1024
// Смещение dirent.ino от начала структуры.
DIRENT_INO = 0
// Смещение dirent.len от начала структуры:
// 4 на FreeBSD, 8 на Linux.
DIRENT_LEN = 4
// Смещение dirent.name от начала структуры:
// 8 на FreeBSD, 10 на Linux.
DIRENT_NAME = 8

// Размер структуры stat и смещения ее элементов:
// dev, ino и nlink.
SIZE_STAT = 64
ST_DEV = 0
ST_INO = 4
ST_NLINK = 10
```

Для всех пунктов этого файла можно написать сценарий с целью автоматического конфигурирования, аналогичный GNU autoconfigure, при использовании которого модификация вручную будет не нужна. Этот сценарий, а также другие простые программы на ассемблере для UNIX доступны в Internet по адресу: http://www.lionking.org/~cubbi/serious/asmix.html.

Теперь, когда все возможные различия между версиями системы предусмотрены, перейдем непосредственно к программе. Упрощая задачу, не станем обрабатывать сообщения об ошибках, но код возврата в случае ошибки всегда будет ненулевым.

```
// Включение файла config.i.
.include "config.i"
// Начало программы.
.globl _start
_start:

.if PWD_posix
// Переносимая (и сложная) стратегия для pwd.

// В %ebp будет храниться указатель на первый символ части искомого пути,
// определенной к настоящему моменту.
```

```
// Она заполняется справа налево - от текущего каталога к корневому.
                movl    $pt_buffer+1024-1,%ebp
// Искомый путь инициализируется символом перевода строки.
                movl    $0x0A,(%ebp)
// В %esi будет храниться указатель на путь вида "../../../",
                movl    $up_buffer,%esi
// инициализированный как "ДО".
                movw    $0x002E,up_buffer
// Выполнить stat("/"), чтобы определить inode и dev для корневого каталога
// (по ним мы позже установим, когда он будет достигнут).
.if SYSCALL_unix
                pushl   $st_stat
                pushl   $root_path
                movl    $SYSCALL_STAT,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
                addl    $12,%esp
.endif
.if SYSCALL_linux
                movl    $st_stat,%ecx
                movl    $root_path,%ebx
                movl    $SYSCALL_STAT,%eax
                int     $0x80
.endif
                testl   %eax,%eax
                jnz     error_exit
// Сохранить dev и inode корневого каталога.
                movl    st_stat+ST_DEV,%eax
                movl    %eax,root_dev
                movl    st_stat+ST_INO,%eax
                movl    %eax,root_ino
// Главный цикл - перемещение по каталогам.
main_posix_cycle:
// Вызвать lstat() для текущего каталога типа "../../../",
// чтобы определить его inode и dev.
.if SYSCALL_unix
                pushl    $st_stat
                pushl   $up_buffer
                movl    $SYSCALL_LSTAT,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
                addl    $12,%esp
.endif
```

```
.if SYSCALL_linux
                movl    $st_stat,%ecx -
                movl    $up_buffer,%ebx
                movl    $SYSCALL_LSTAT,%eax
                int     $0x80
.endif
                testl   %eax,%eax
                jnz     error_exit
// Сохранить inode и dev каталога, который сейчас будет сканироваться.
                movl    st_stat+ST_DEV,%eax
                movl    %eax,dev
                movl    st_stat+ST_INO,%ebx
                movl    %ebx,ino
// Проверить, не совпадает ли этот каталог с корневым.
                cmpl    %eax,root_dev
                jne     posix_not_finished
                cmpl    %ebx,root_ino
                jne     posix_not_finished
// Если совпадает - вписать последний символ "/" на левом конце искомого пути
                decl    %ebp
                movb    $0x2F,(%ebp)
// и вывести его на экран.
                jmp     posix_write
// Иначе -
posix_not_finished:
// добавить "..\0" к up_buffer, чтобы перейти к вышележащему каталогу.
                movw    $0x2E2E,(%esi)
                incl    %esi
                incl    %esi
                movb    $0,(%esi)
// Открыть данный каталог только для чтения (O_RDONLY).
.if SYSCALL_unix
                pushl   $0
                pushl   $up_buffer
                movl    $SYSCALL_OPEN,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
                addl    $12,%esp
.endif
.if SYSCALL_linux
                movl    $0,%ecx
                movl    $up_buffer,%ebx
                movl    $SYSCALL_OPEN,%eax
                int     $0x80
.endif
```

```
                testl   %eax,%eax
                jl      error_exit
                movl    %eax,%edi
// Выполнить fstat над этим каталогом.
.if SYSCALL_unix
                pushl   $st_stat
                pushl   %eax
                movl    $SYSCALL_FSTAT,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
                addl    $12,%esp
.endif
.if SYSCALL_linux
                movl    $st_stat,%ecx
                movl    %eax,%ebx
                movl    $SYSCALL_FSTAT,%eax
                int     $0x80
.endif
                testl   %eax,%eax
jnz     error_exit
// Если все в порядке, добавить "/" для следующего каталога.
                movb    $0x2F, (%esi)
                incl    %esi
// Вложенный цикл: рассмотреть каждую запись в вышележащем
// каталоге и сравнить ее inode и dev с текущей.
                xorl    %ebx,%ebx
readdir_cycle:
// Вызов readdir() легче использовать, чем getdents().
.if SYSCALL_READDIR
.if SYSCALL_unix
                pushl   $1
                pushl   $dirent
                pushl   %edi
                movl    $SYSCALL_READDIR,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
                addl    $16,%esp
.endif
.if SYSCALL_linux
                movl    $1,%edx
                movl    $dirent,%ecx
                movl    %edi,%ebx
                movl    $SYSCALL_READDIR,%eax
                int     $0x80
.endif
```

```
// В $ebx будет структура dirent, над которой мы работаем в настоящий момент.
                movl    $dirent,%ebx
                decl    %eax
jne     notfound
.else
// Если нет readdir(), но есть getdents и getdirentries, алгоритм усложняется, так
// как они возвращают не одну структуру dirent, а столько, сколько поместится в буфер.
                testl   %ebx,%ebx
                jz      time_to_getdent
                xorl    %eax,%eax
                movw    DIRENT_LEN(%ebx),%ax
                addl    %eax,%ebx
// Проверить, выйдет ли %ebx за пределы буфера после добавления LEN.
                xorl    %eax,%eax
                movw    DIRENT_LEN(%ebx),%ax
                cmpl    dirent_filled,%eax
jge     time_to_getdent
                movw    DIRENT_LEN(%ebx),%ax
                testl   %eax,%eax
jnz     skip_getdent
time_to_getdent:
// Если выходит - пора делать getdents() или getdirentries() -
// в зависимости от операционной системы.
.if SYSCALL_unix
.if SYSCALL_GETDENTS
                pushl   $SIZE_DIRENT
                pushl   $dirent
                pushl   %edi
                movl    $SYSCALL_GETDENTS,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
                addl    $16,%esp
.else
                pushl   $basep
                pushl   $SIZE_DIRENT
                pushl   $dirent
                pushl   %edi
                movl    $SYSCALL_GETDIRENTRIES,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
                addl    $20,%esp
.endif
.endif
```

```
.if SYSCALL_linux
                movl    $SIZE_DIRENT,%edx
                movl    $dirent,%ecx
                movl    %edi,%ebx
                movl    $SYSCALL_GETDENTS,%eax
                int     $0x80
.endif
                movl    %eax,dirent_filled
                testl   %eax,%eax
                je      notfound
                movl    $dirent,%ebx
skip_getdent:
.endif

// Скопировать в сканируемом каталоге имя полученной записи
// в конец строки ../../../../, не смещая указатель в %esi.
                xorl    %ecx,%ecx
                pushl   %esi
                pushl   %edi
                movl    %esi,%edi
                movl    %ebx,%esi
                addl    $DIRENT_NAME,%esi
                movw    DIRENT_LEN(%ebx),%cx
                incl    %ecx
                rep
                movsb
                popl    %edi
                popl    %esi
// Выполнить lstat() для этой записи, чтобы получить ее inode и dev.
                pushl   %ebx
.if SYSCALL_unix
                pushl   $st_stat
                pushl   $up_buffer
                movl    $SYSCALL_LSTAT,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
                addl    $12,%esp
.endif
.if SYSCALL_linux
                movl    $st_stat,%ecx
                movl    $up_buffer,%ebx
                movl    $SYSCALL_LSTAT,%eax
                int     $0x80
.endif
                popl    %ebx
                testl   %eax,%eax
                jnz     readdir_cycle
```

```
// Если они не совпадают с сохраненными inode и dev для
// текущего сканируемого каталога - продолжить.
                movl    dev,%eax
                cmpl    %eax,st_stat+ST_DEV
                jne     readdir_cycle
                movl    ino,%eax
                cmpl    %eax,st_stat+ST_INO
                jne     readdir_cycle
// Вернуть up_buffer в состояние ../../../../../, соответствующее вышележащему каталогу.
                movl    $0,(%esi)
// Добавить "/" в создаваемый путь pwd (кроме случая, если это был самый первый каталог).
                cmpb    $1,first
                je      not_first
                decl    %ebp
                movb    $0x2F,(%ebp)
not_first:
                movb    $0,first
// Сдвинуть указатель в %ebp на namlen байтов влево.
                xorl    %edx,%edx
.if SYSCALL_READDIR
                movw    DIRENT_LEN(%ebx),%dx
.else
                pushl   %esi
                xorl    %ecx,%ecx
                movl    %ebx,%esi
                addl    $DIRENT_NAME,%esi
seek_zero:
                incl    %ecx
                movb    (%esi,%ecx,1),%al
                testb   %al,%al
jnz     seek_zero
                movl    %ecx,%edx
                popl    %esi
.endif
                subl    %edx,%ebp
// Скопировать имя найденного каталога в pwd, не сдвигая указатель.
                xorl    %ecx,%ecx
                pushl   %esi
                pushl   %edi
                movl    %ebp,%edi
                movl    %ebx,%esi
                addl    $DIRENT_NAME,%esi
.if SYSCALL_READDIR
                movw    DIRENT_LEN(%ebx),%cx
.else
                movl    %edx,%ecx
.endif
                rep
```

```
                movsb
                popl    %edi
                popl    %esi
// Закрыть открытый каталог.
.if SYSCALL_unix
                pushl   %edi
                movl    $SYSCALL_CLOSE,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
                addl    $8,%esp
.endif
.if SYSCALL_linux
                movl    %edi,%ebx
                movl    $SYSCALL_CLOSE,%eax
                int     $0x80
.endif
// Продолжить главный цикл, пока не будет достигнут корневой каталог.
                jmp     main_posix_cycle
notfound:
                jmp error_exit
posix_write:
// Вывести на экран найденный путь.
                movl    $pt_buffer+1024,%edx
                subl    %ebp,%edx
                movl    %ebp,%ecx
                xorl    %ebx,%ebx
                incl    %ebx
.if SYSCALL_unix
                pushl   %edx
                pushl   %ecx
                pushl   %ebx
                movl    $SYSCALL_WRITE,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
                addl    $16,%esp
.endif
.if SYSCALL_linux
                movl    $SYSCALL_WRITE,%eax
                int     $0x80
.endif
.endif
// Конец стратегии PWD_posix.

// Просто распечатать значение переменной среды PWD (плохая стратегия).
.if PWD_env
                cld
```

```
                popl    %ebx
                movl    4(%esp,%ebx,4),%edi
// Теперь адрес списка переменных в %edi.
                xorl    %eax,%eax
// Сканирование переменных в поисках той, начало которой совпадет с env_name.
env_scan_cycle:
                cmpl    $0,X%edi)
jz      error_exit
                movl    $env_name,%esi
                movl    $env_name_l,%ecx
                repe
                cmpsb
                jcxz    found_variable
                xorl    %ecx,%ecx
                decl    %ecx
                repne
                scasb
                jmp     env_scan_cycle

// Теперь %edi - адрес значения переменной PWD, заканчивающийся нулем.
found_variable:
                xorl    %edx,%edx
                decl    %edx
// Найти его длину
loop_zero_scan:
                incl    %edx
                cmpb    $0,(%edi,%edx,1)
                jne     loop_zero_scan
                movb    $0x0A,(%edi,%edx,1)
                incl    %edx
// и вывести на экран.
.if SYSCALL_unix
                pushl   %edx
                pushl   %edi
                pushl   $1
                movl    $SYSCALL_WRITE,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
                addl    $16,%esp
.endif
.if SYSCALL_linux
                movl    %edi,%ecx
                movl    $1,%ebx
                movl    $SYSCALL_WRITE,%eax
                int     $0x80
.endif
.endif
```

```
// Стратегия proc - выводится имя каталога, на который
// указывает символическая ссылка /proc/self/cwd.
.if PWD_proc
// Вызвать readlink("/proc/self/cwd").
                movl    $255,%edx
                movl    $linux_buffer,%ecx
                movl    $linux_pwd,%ebx
                movl    $SYSCALL_READLINK,%eax
// Эта стратегия осуществима только в Linux.
                int     $0x80
.endif

// Стратегия с использованием системного вызова.
.if PWD_sys
// Вызвать getcwd().
.if SYSCALL_unix
                pushl   $MAX_PATH
                pushl   $linux_buffer
                movl    $SYSCALL_GETCWD,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
            f   .word   7
                addl    $12,%esp
.endif
.if SYSCALL_linux
                movl    $MAX_PATH,%ecx
                movl    $linux_buffer,%ebx
                movl    $SYSCALL_GETCWD,%eax
                int     $0x80
.endif
                testl   %eax,%eax
                js      error_exit
                decl    %eax
// В случае с FreeBSD надо дополнительно определить длину возвращенной строки.
.if SYSCALL_unix
                xorl    %eax,%eax
                xorl    %ecx,%ecx
                decl    %ecx
                movl    $linux_buffer,%edi
                repne
                scasb
                subl    $linux_buffer,%edi
                movl    %edi,%eax
                incl    %eax
.endif
.endif

// Вывод на экран для стратегий proc и sys.
// Добавить символ новой строки в конце.
```

```
.if PWD_proc | PWD_sys
                movb    $0x0A,linux_buffer(%eax)
                incl    %eax

// Вывести на экран linux_buffer.
.if SYSCALL_unix
                pushl   %eax
                pushl   $linux_buffer
                pushl   $1
                movl    $SYSCALL_WRITE,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
                addl    $16,%esp
.endif
.if SYSCALL_linux
                movl    %eax,%edx
                xorl    %ebx,%ebx
                movl    $linux_buffer,%ecx
                incl    %ebx
                movl    $SYSCALL_WRITE,%eax
                int     $0x80
.endif
.endif

// Выход из программы без ошибок exit(0).
exit:
                xorl    %ebx,%ebx
.if SYSCALL_unix
                pushl   %ebx
                movl    $SYSCALL_EXIT,%eax
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   1
.endif
.if SYSCALL_linux
                movl    $SYSCALL_EXIT,%eax
                int     $0x80
.endif

// Выход из программы с ошибками.
.if ~(-PWD_proc)
exit_errno:
exit_ENOENT:
error_exit:
.if SYSCALL_unix
                pushl   $1
                movl    $SYSCALL_EXIT,%eax
```

```
                pushl   %eax
                .byte   0x9A
                .long   0
                .word   7
.endif
.if SYSCALL_linux
                xorl    %ebx,%ebx
                incl    %ebx
                movl    $SYSCALL_EXIT,%eax
                int     $0x80
.endif
.endif

// Область данных!
.data
.if PWD_posix
// Текущий каталог для stat().
pwd_path:
                .asciі  ".\000"
// Корневой каталог для stat().
root_path:
                .ascii  "/\000"
// Флаг, нужный для того, чтобы не ставить "/" перед первым элементом пути.
first:
                .byte   1
.endif

// Имя файла в /proc для readlink().
.if PWD_proc
linux_pwd:
                .ascii  "/proc/self/cwd\000"
.endif

// Переменная среды для стратегии PWD.
.if PWD_env
env_name:
                .ascii  "PWD="
env_name_l = . -env_name
.endif

// Указатель для системного вызова getdirentries().
.if SYSCALL_GETDIRENTRIES
basep:
                .long   0
.endif

.bss
// Буфер для stat().
.if PWD_posix
                .lcomm  st_stat,SIZE_STAT
.endif
```

```
// Буфер для результата работы getcwd() и readlink().
.if PWD_proc | PWD_sys
                .lcomm  linux_buffer,MAX_PATH
.endif

.if PWD_posix
// Буфер для вывода pwd.
                .lcomm  pt_buffer,1024
// Буфер для сканируемого каталога ( ../../../../).
                .lcomm  up_buffer,1024
.endif

// Различные переменные.
.if PWD_posix
                .lcomm  root_dev,4
                .lcomm  dev,4
                .lcomm  root_ino,4
                .lcomm  ino,4
                .lcomm  bpt,4
// Буфер для записи из каталога (dirent).
                .lcomm  dirent,SIZE_DIRENT
.if SYSCALL_GETDENTS | SYSCALL_GETDIRENTRIES
                .lcomm  dirent_filled,4
.endif
.endif
```

Для компиляции этой программы на любой системе достаточно двух команд:

```
as -o pwd.o pwd.s
ld -o pwd pwd.o
```

Полученная версия pwd занимает:

- на Linux 2.2 - 428 байт (системная версия - 19 332 байта);
- на Linux 2.0 - 788 байт (системная версия - 18 440 байт);
- а на FreeBSD 3.1 - 484 байта (системная версия - 51 916 байт);
- на FreeBSD 2.2 - 8192 байта (системная версия - 45 056 байт).

Если на FreeBSD 2.2 установлена поддержка запуска ELF-программ, можно скомпилировать работающий на ней файл pwd длиной 940 байт, но в любом случае эти размеры впечатляют. Более того, для работы нашей версии pwd не требуется никакая библиотека типа libc.so - это полностью самостоятельная статическая программа. Такие программы, написанные целиком на ассемблере, не только отличаются сверхмалыми размерами (в 20-100 раз меньше аналогов на C), но и работают быстрее, потому что обращаются напрямую к ядру системы.

Заключение

Итак, прочитав эту книгу, вы познакомились с программированием на языке ассемблера во всей широте его проявлений - от создания простых программ и процедур, вызываемых из приложений на других языках, до драйверов устройств и операционных систем. Теперь должно быть очевидно, что ассемблер не только не сдает свои позиции, но и не может их сдать - он неотъемлемо связан с компьютером, и всюду, как только мы опускаемся с уровня абстракций языков высокого уровня, рано или поздно встречаемся с ним. В то же время и абстракции, и сложные управляющие структуры, и структуры данных реализуются на языке ассемблера наиболее эффективно - не зря же Дональд Кнут, автор знаменитой книги «Искусство программирования», использовал для иллюстрации перечисленных структур и алгоритмов только ассемблер.

Ассемблер настолько многогранен, что нет никакой возможности описать в одной книге все области программирования, в которых он может быть использован. Методы защиты от копирования и противодействия отладчикам, строение различных файловых систем, программирование на уровне портов ввода-вывода таких устройств, как IDE- или SCSI-диски, и многое другое осталось в стороне, и это правильно, потому что мир ассемблера не заканчивается вместе с этой книгой, а только начинается.

Приложение 1. Таблицы символов

1. Символы ASCII

Номера строк соответствуют первой цифре в шестнадцатеричном коде символа, номера столбцов - второй, так что, например, код большой латинской буквы A - 41h (см. рис. 18).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	@	©	B	♥	♦	♣	*	*	B	o	◘	d	Ч	Г	Л	*
1	▶	◀	Г	‼	¶	§	_	↕	T	↓	→	←	∟	↔	▲	T
2		↑	"	#	S	x	&	'	(	)	*	+	,	–	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	V
4	&	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	V	Z	[	Ч	]	^	
6	`	a	Ъ	c	d	e	f	g	h	i	j	k	1	m	n	o
7	p	g	r	s	t	u	v	u	x	y	z	{	i	}	~	⌂

Рис. 18. Таблица символов ASCII

2. Управляющие символы ASCII

Таблица 23. Управляющие символы ASCII

Код	Имя	Ctrl-код	Назначение
0	NUL	"@	Пусто (конец строки)
1	SOH	^A	Начало заголовка
2	STX	^B	Начало текста
3	EOT	^C	Конец текста
4	ENQ	^D	Конец передачи
6	ACK	^F	Подтверждение
7	BEL	^G	Звонок
8	BS	^H	Шаг назад
9	HT	^I	Горизонтальная табуляция
0A	LF	^J	Перевод строки
0B	VT	"K	Вертикальная табуляция
0C	FF	^L	Перевод страницы
0D	CR	^M	Возврат каретки
0E	SO	^N	Выдвинуть
0F	SI	^O	Сдвинуть
10	DLE	^P	Оставить канал данных
11	DC1/XON	^Q	Управление устройством- 1
12	DC2	^R	Управление устройством-2
13	DC3/XOFF	^S	Управление устройством-3
14	DC4	^T	Управление устройством-4
15	NAK	^U	Отрицательное подтверждение
16	SYN	^V	Синхронизация
17	ETB	^W	Конец блока передачи
18	CAN	^X	Отмена
19	EM	^Y	Конец носителя
1A	SUB	^Z	Замена
1B	ESC	^[	Escape
1C	FS	"\	Разделитель файлов
1D	GS	^]	Разделитель групп
1E	RS	^^	Разделитель записей
1F	US	^_	Разделитель полей
20	SP		Пробел
7F	DEL	^?	Удаление

3. Кодировки второй половины ASCII

Кодировка по умолчанию для первых компьютеров - этот набор символов хранится в постоянной памяти и используется BIOS (см. рис. 19)

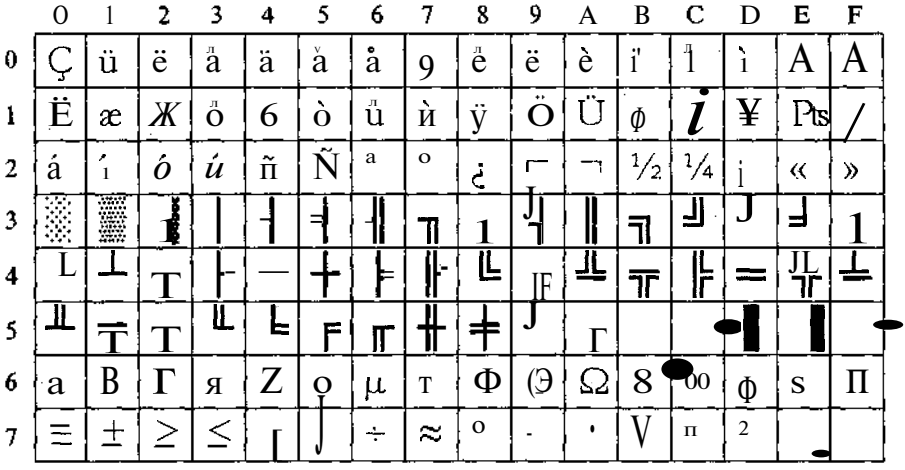

Рис. *19. Кодировка IBM cp437*

Кодировка cp866 используется DOS-приложениями как основная кодировка и компьютерной сетью Fidonet как транспортная кодировка (см. рис. 20).

Рис. 20. Кодировка IBM cp866

Кодировка KOI8-r используется как транспортная в Internet и как основная в большинстве бесплатно распространяемых операционных систем (см. рис. 21).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	─	│	┌	┐	└	┘	├	┤	┬	┴	┼	▀	▄	▌	▐	▬
1	▒	▒	▐	▗	▝	▖	▘	≈	≤	≥	⌠	⌡	°	²	·	÷
2	═	‖	╒	ё	╓	╔	╕	1	╗	╘	╙	╚	╛	╜	╝	╞
3	╟	╠	╡	Ё	╢	╖	╫	╪	╤	╧	╨	╩	╬	╦	╠	©
4	ю	а	б	ц	а	е	ф	г	х	и	й	к	л	м	н	о
5	п	я	р	с	т	у	ж	в	ь	ы	з	ш	э	щ	ч	ъ
6	Ю	А	Б	Ц	Д	Е	Ф	Г	Х	И	Й	К	Л	М	Н	О
7	П	Я	Р	С	Т	У	Ж	В	Ь	Ы	З	Ш	Э	Щ	Ч	Ъ

Рис. 21. Кодировка KOI8-r (RFC1489)

Кодировка ISO 8859-5 используется как основная в большинстве коммерческих UNIX-совместимых операционных систем (см. рис. 22).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	D	D	D	П	П	D	П	D	D	П	П	D	П	D	П	П
1	П	D	П	П	D	D	D	D	D	D	П	D	D	П	D	П
2		Ё	Ђ	Ѓ	Є	S	I	Ї	J	Љ	Њ	Ъ	Ќ	-	Ў	Ц
3	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
4	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
5	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
6	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
7	№	ё	ђ	ѓ	е	s	i	ї	j	љ	№	ћ	ќ	§	ў	ц

Рис. 22. Кодировка ISO 8859-5

Кодировка cp1251 используется как основная в графических приложениях для Microsoft Windows (см. рис. 23).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Ђ	Ѓ	‚	ѓ	„	…	†	‡	€	‰	Љ	‹	Њ	Ќ	Ћ	Џ
1	ђ	‘	’	“	”	•	–	—	™	™	љ	›	№	ќ	ћ	џ
2		Ў	ў	Ј	¤	Ґ	¦	§	Ё	©	Є	«	¬	-	®	Ї
3	°	±	І	і	ґ	µ	¶	·	ё	№	є	»	ј	Ѕ	ѕ	ї
4	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
5	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
6	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
7	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я

Рис. 23. Кодировка cp1251

4. Коды символов расширенного ASCII

Таблица 24. Расширенные ASCII-коды[1]

Клавиша	Код	Клавиша	Код	Клавиша	Код	Клавиша	Код	Клавиша	Код
F1	3Bh	Alt-R	13h	Shift-F11	87h	Alt-Tab	A5h	AIM	17h
F2	3Ch	Alt-S	1Fh	Shift-F12	88h	Ctrl-Tab	94h	Alt-J	24h
F3	3Dh	Alt-T	14h	Alt-0	81h	Alt-Del	A3h	Alt-K	25h
F4	3Eh	Alt-U	16h	AIM	82h	Alt-End	9Fh	Alt-L	26h
F5	3Fh	Alt-V	2Fh	Alt-2	83h	Alt-Home	97h	Ctrl-Right	74h
F6	40h	Alt-W	11h	Alt-3	84h	Alt-Ins	A2h	Ctrl-End	75h
F7	41h	Alt-X	2Dh	Alt-4	85h	Alt-PgUp	99h	Ctrl-Home	77h
F8	42h	Alt-Y	1Sh	Alt-5	86h	Alt-PgDn	A1h	Ctrl-PgDn	76h
F9	43h	Alt-Z	2Ch	Alt-6	87h	Alt-Enter	1Ch	Ctrl-PgUp	84h
F10	44h	Alt-\	2Bh	Alt-7	88h	Ctrl-F1	5Eh	Alt-Up	98h
F11	85h	Alt-,	33h	Alt-8	89h	Ctrl-F2	5Fh	Alt-Down	A0h
F12	86h	Alt-.	34h	Alt-9	8Ah	Ctrl-F3	60h	Alt-Left	9Bh
Alt-F1	68h	Alt-/	35h	AltC	8Bh	Ctrl-F4	61h	Alt-Right	9Dh
Alt-F2	69h	Alt-BS	OEh	Alt-=	8Ch	Ctrl-F5	62h	Alt-K/	A4h
Alt-F3	6Ah	Alt-[	1Ah	NUL	03h	Ctrl-F6	63h	Alt-K*	37h
Alt-F4	6Bh	Alt-]	1Bh	Shift-Tab	OFh	Ctrl-F7	64h	Alt-K-	4Ah
Alt-F5	6Ch	Alt-;	27h	Ins	52h	Ctrl-F8	65h	Alt-K+	4Eh
Alt-F6	6Dh	Alt-'	28h	Del	53h	Ctrl-F9	66h	Alt-Kenter	A6h
Alt-F7	6Eh	Alt-`	29h	SysRq	72h	Ctrl-F10	67h	Ctrl-K/	95h
Alt-F8	6Fh	Shift-F1	54h	Down	50h	Ctrl-F11	89h	Ctrl-K*	96h
Alt-F9	70h	Shift-F2	55h	Left	4Bh	Ctrl-F12	8Ah	Ctrl-K-	8Eh
Alt-F10	71h	Shift-F3	56h	Right	4Dh	Alt-A	1Eh	Ctrl-K+	90h
Alt-F11	8Bh	Shift-F4	57h	Up	48h	Alt-B	30h	Ctrl-K8	8Dh
Alt-F12	8Ch	Shift-F5	58h	End	4Fh	Alt-C	2Eh	Ctrl-K5	8Fh
Alt-M	32h	Shift-F6	59h	Home	47h	Alt-D	20h	Ctrl-K2	91h
Alt-N	31h	Shift-F7	5Ah	PgDn	S1h	Alt-E	12h	Ctrl-K0	92h
Alt-O	18h	Shift-F8	5Bh	PgUp	49h	Alt-F	21h	Ctrl-K.	93h
Alt-P	19h	Shift-F9	5Ch	Ctrl-Left	73h	Alt-G	22h		
Alt-Q	10h	Shift-F10	5Dh	Alt-Esc	01h	Alt-H	23h		

[1] Префикс «K» соответствует клавишам на цифровой клавиатуре.

5. Скан-коды клавиатуры

Таблица 25. Скан-коды[1]

Клавиша	Код	Клавиша	Код	Клавиша	Код	Клавиша	Код
Esc	01h	Enter	1Ch	K*	37h	Ins	52h
1 !	02h	Ctrl	1Dh	Alt	38h	Del	53h
2 @	03h	A	1Eh	SP	39h	SysRq	54h
3 #	04h	S	1Fh	Caps	3Ah	Macro	56h
4 $	05h	D	20h	F1	3Bh	F11	57h
5%	06h	F	21h	F2	3Ch	F12	58h
6 ^	07h	G	22h	F3	3Dh	PA1	5Ah
7 &	08h	H	23h	F4	3Eh	F13/LWin	5Bh
8 *	09h	J	24h	F5	3Fh	F14/RWin	5Ch
9 (	0Ah	K	25h	F6	40h	F15/Menu	5Dh
0)	0Bh	L	26h	F7	41h	F16	63h
- _	0Ch	; :	27h	F8	42h	F17	64h
= +	0Dh	' "	28h	F9	43h	F18	65h
BS	0Eh	` ~	29h	F10	44h	F19	66h
Tab	0Fh	Lshift	2Ah	Num	45h	F20	67h
Q	10h	\ |	2Bh	Scroll	46h	F21	68h
W	11h	Z	2Ch	Home	47h	F22	69h
E	12h	X	2Dh	-	48h	F23	6Ah
R	13h	C	3Eh	PgUp	49h	F24	6Bh
T	14h	V	2Fh	K-	4Ah	EraseEOF	6Dh
Y	15h	B	30h		4Bh	Copy/Play	6Fh
U	16h	N	31h	K5	4Ch	CrSel	72h
I	17h	M	32h	•®	4Dh	Delta	73h
O	18h	, <	33h	K+	4Eh	ExSel	74h
p	19h	. >	34h	End	4Fh	Clear	76h
[{	1Ah	/ ?	35h	T	50h		
] }	1Bh	RShift	36h	PgDn	51h		

[1] Префикс «K» соответствует клавишам на цифровой клавиатуре.

Таблица 26. Служебные скан-коды

Код	Функция	Код	Функция
00h	Буфер клавиатуры переполнен	FAh	ACK
AAh	Самотестирование закончено	FCh	Ошибка самотестирования
E0h	Префикс для серых клавиш	FDh	Ошибка самотестирования
E1h	Префикс для клавиш без кода отпускания	FEh	RESEND
F0h	Префикс отпускания клавиши	FFh	Ошибка клавиатуры
EEh	Эхо		

Приложение 2. Команды Intel 80x86

В этом приложении приведены скорости выполнения всех команд процессоров Intel от 8086 до Pentium II и машинные коды, которые им соответствуют.

1. Общая информация о кодах команд

1.1. Общий формат команды процессора Intel

Команда может содержать до шести полей:

1. Префиксы - от нуля до четырех однобайтных префиксов.
2. Код - один или два байта, определяющие команду.
3. ModR/M - 1 байт (если он требуется), описывающий операнды:
   биты 7-6: поле MOD - режим адресации;
   биты 5-3: поле R/O - либо указывает регистр, либо является продолжением кода команды;
   биты 2-0: поле R/M - либо указывает регистр, либо совместно с MOD - режим адресации.
4. SIB - 1 байт, если он требуется (расширение ModR/M для 32-битной адресации):
   биты 7-6: S - коэффициент масштабирования;
   биты 5-3: I - индексный регистр;
   биты 2-0: B - регистр базы.
5. Смещение - 0, 1, 2 или 4 байта.
6. Непосредственный операнд - 0, 1, 2 или 4 байта - будем использовать /ib и /iw для указания этих операндов.

1.2. Значения полей кода команды

В кодах некоторых команд мы будем встречать специальные биты и группы битов, которые обозначим w, s, d, reg, sreg и cond:

- w = 0, если команда работает с байтами;
- w = 1, если команда работает со словами или двойными словами;

- s = 0, если непосредственный операнд указан полностью;
- s = 1, если непосредственный операнд — младший байт большего операнда и должен рассматриваться как число со знаком;

- d = 0, если код источника находится в поле R/O, а приемника - в R/M;
- d = 1, если код источника находится в поле R/M, а приемника - в R/O.

Запись 10dw будет означать, что код команды - 000100dw.

Поле reg определяет используемый регистр и имеет длину 3 бита:
000 - AL/AX/EAX/ST(0)/MM0/XMM0
001 - CL/CX/ECX/ST(1)/MM1/XMM1
010 - DL/DX/EDX/ST(2)/MM2/XMM2
011 - BL/BX/EBX/ST(3)/MM3/XMM3
100 - AH/SP/ESP/ST(4)/MM4/XMM4
101 - CH/BP/EBP/ST(5)/MM5/XMM5
110 - DH/SI/ESI/ST(6)/MM6/XMM6
111 - BH/DI/EDI/ST(7)/MM7/XMM7
Запись C8r будет означать 11001reg.

Поле sreg определяет используемый сегментный регистр:
000-ES
001-CS
010 – SS
OH – DS
100-FS
101-GS

Поле cond определяет условие для команд Jcc, CMOVcc, SETcc, FCMOVcc. Его значения для разных команд:
0000 - O
0001 - NO
0010 – C/B/NAE
OOH – NC/NB/AE
0100 - E/Z
0101 - NE/NZ
0110 – BE/NA
0111 – NBE/A
1000 – S
1001 - NS
1010 - P/PE
1011 – NP/PO
1100 – L/NGE
1101 – NL/GE
1110 – LE/NG
1111 – LNE/G
Запись типа 4cc будет означать 0100cond.

1.3. Значения поля ModRM

Поле R/O (биты 5-3) содержит либо дополнительные три бита кода команды, либо код операнда, который может быть только регистром. Будем обозначать второй случай reg, а в первом записывать используемые биты.

Поля MOD (биты 7-6) и R/M (биты 3-0) определяют операнд, который может быть как регистром, так и переменной в памяти:

- MOD = 11, если используется регистровая адресация и R/M содержит код регистра reg
- MOD = 00, если используется адресация без смещения ([BX + SI] или [EDX])
- MOD = 01, если используется адресация с 8-битным смещением (variable[BX+SI])
- Q MOD = 10, если используется адресация с 16- или 32-битным смещением

Значение поля R/M различно в 16- и 32-битных режимах.

R/M в 16-битном режиме:

000 - [BX+SI]
001 - [BX+DI]
010 – [BP + SI]
011 – [BP+DI]
100 - [SI]
101 - [DI]
110 - [BP] (кроме MOD = 00 - в этом случае после ModR/M располагается 16-битное смещение, то есть используется прямая адресация)
111 – [BX]

R/M в 32-*битном* режиме:

000 - [EAX]
001 - [ECX]
010 - [EDX]
011 - [EBX]
100 - используется SIB
101 - [EBP] (кроме MOD = 00 - в этом случае используется SIB, после которого располагается 32-битное смещение)
110 – [ESI]
111 - [EDI]

1.4. Значения поля SIB

Значение поля S:

00 - не Используется
01 - умножение на 2
10 - умножение на 4
11 - умножение на 8

Значения полей I и B:
(I - регистр, используемый в качестве индекса, то есть умножающийся на S; B - регистр базы, который не умножается)

000 - EAX
001 - ECX
010 – EDX
011 - EBX

100 - для I - индекса нет
для B - ESP
101 - для I - EBP
для B - EBP, только если MOD = 01 или 10; если MOD = 00 - базы нет
110 - ESI
111 - EDI

Поля ModR/M и SIB будут записываться как /r, если поле R/O содержит код регистра, или /0 - /7, если поле R/O содержит дополнительные три бита кода команды. В других случаях поля ModR/M и SIB отсутствуют только у команд без операндов, поэтому они не будут обозначаться дополнительно.

2. Общая информация о скоростях выполнения

Скорости выполнения команд для процессоров 8086 - P5 даны в тактах (когда говорят, что тактовая частота процессора 100 MHz, это означает, что за секунду проходит 100 миллионов тактов).

Для процессоров P5 (Pentium, Pentium MMX) помимо скорости указано, может ли команда выполняться одновременно с другими, и если да, то в каком конвейере (см. раздел 9.3.2):

- UV - может выполняться одновременно, в любом конвейере;
- PU - может выполняться одновременно, в U-конвейере;
- Q PV - может выполняться одновременно, в V-конвейере;
- FX - может выполняться одновременно с командой FXCH;
- a NP - не может выполняться одновременно (для MMX - не может выполняться одновременно с командой того же типа, который указан после буквы п).

Для процессоров P6 (Pentium Pro, Pentium II) указано количество микроопераций, на которые декодируется команда. Буквой C отмечены команды со сложным строением (см. раздел 9.3.3).

Во всех случаях даны минимально возможные скорости - если шина данных не заблокирована, операнды выровнены по границам двойных слов, операнды находятся в кэше данных, команды по адресу для перехода расположены в кэше кода, переходы угаданы процессором правильно, в момент выполнения команды не происходит заполнения кэша, страницы находятся в TLB (иначе для P5 следует прибавить 13-28 тактов), в момент выполнения команды нет исключений, не происходят AGI и т. д.

Операнды обозначаются следующим образом:

- im - непосредственный операнд;
- i8, i16, i32 - непосредственный операнд указанного размера;
- ac - EAX, AX, AL;
- r - любой регистр общего назначения;
- r8 - AH, AL, BH, BL, DH, DL, CH, CL;
- a r16 - AX, BX, CX, DX, BP, SP, SI, DI;

- r32 - EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI;
- sr - сегментный регистр;
- m – операнд в памяти;
- mm - регистр MMX;
- xmm - регистр SSE;
- s0 - регистр ST(0);
- si - регистр ST(i).

Для команд условных переходов приводятся два значения скорости выполнения - если переход произошел и если нет.

Другие используемые сокращения:

- PM - защищенный режим;
- RM - реальный режим;
- VM - режим V86;
- TS - переключение задачи;
- CG - шлюз вызова;
- TG - шлюз задачи;
- /in - увеличение привилегий;
- /out - уменьшение привилегий;
- /NT - переход во вложенную задачу.

Время переключения задачи:

- Pentium: TS = 85 при переключении в 32-битный и 16-битный TSS и 71 при переключении в V86 TSS;
- 80486: TS = 199 при переключении в 32-битный и 16-битный TSS и 177 при переключении в V86 TSS;
- 80386: TS = 307-314 при переключении в 32-битный и 16-битный TSS и 224-231 при переключении в V86;
- 80286: TS = 280.

3. Префиксы

Все префиксы выполняются за 1 такт и имеют размер 1 байт:

0F0h: LOCK
OF2h: REPNE/REPNZ
OF3h: REP/REPE/REPZ
2Eh: CS:
36h: SS:
3Eh: DS:
26h: ES:
64h: FS:
65h: GS:
66h: OS
67h: AS

4. Команды процессоров Intel 8088 - Pentium III

Таблица 27. Команды

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
AAA	37	8	8	3	4	3	3 NP	1m
AAD i8	D5 ib	60	15	14	19	14	10 NP	3m
AAM i8	D4 ib	83	19	16	17	15	18 NP	4m
MS	3F	8	7	3	4	3	3 NP	1m
ADC ac,im	14w im	4	4	3	2	1	1 PU	2m
ADC r,im	80sw /2 im	4	4	3	2	1	1 PU	2m
ADC m,im	80sw /2 im	23+ea	16	7	7	3	3 PU	4m
ADC r,r	10dw /r	3	3	2	2	1	1 PU	2m
ADC m,r	10dw /r	24+ea	10	7	7	3	3 PU	4m
ADC r,m	10dw /r	13+ea	10	7	6	2	2 PU	3m
ADD ac,im	04w im	4	4	3	2	1	1 UV	1m
ADD r,im	80sw /0 im	4	4	3	2	1	1 UV	1m
ADD m,im	80sw /0 im	23+ea	16	7	7	3	3 UV	4m
ADD r,r	00dw /r	3	3	2	2	1	1 UV	1m
ADD m,r	00dw /r	24+ea	10	7	7	3	3 UV	4m
ADD r,m	00dw /r	13+ea	10	7	6	2	2 UV	2m
AND ac,im	24w im	4	4	3	2	1	1 UV	1m
AND r,im	80sw /4 im	4	4	3	2	1	1 UV	1m
AND m,im	80sw /4 im	23+ea	16	7	7	3	3 UV	4m
AND r,r	20dw /r	3	3	2	2	1	1 UV	1m
AND m,r	20dw /r	24+ea	10	7	7	3	3 UV	4m
AND r,m	20dw /r	13+ea	10	7	6	2	2 UV	2m
ARPL r,r	63 /r			10	20	9	7 NP	C
ARPL m,r	63 /r			11	21	9	7 NP	C.
BOUND r,m	62 /r		35	13	10	7	8 NP	C
BSFr16,r16	OF BC /r				10+3n	6..42	6..34 NP	2m
BSF r32,r32	OF BC /r				10+3n	6..42	6..42 NP	2m
BSF r16,m16	OF BC /r				10+3n	6..43	6..35 NP	3m
BSF r32,m32	OF BC /r				10+3n	6..43	6..43 NP	3m
BSR r16,r16	OF BD /r				10+3n	6..103	7..39 NP	2m
BSR r32,r32	OF BD /r				10+3n	7..104	7..71 NP	2m
BSR r16,m16	0F BD /r				10+3n	6..103	7..40 NP	3m
BSR r32,m32	OF BD /r				10+3n	7..104	7..72 NP	3m
BSWAP r32	0F C8r					1	1 NP	2m
BT r,r	OF A3 /r				3	3	4 NP	1m
BT m,r	OF A3 /r				12	8	9 NP	C
BT r,i8	OF BA /4 ib				3	3	4 NP	1m
BT m,i8	OF BA /4 ib				6	2	4 NP	2m
BTC r,r	OF BB /r				6	6	7 NP	1m
BTC m,r	OF BB /r				13	13	13 NP	C
BTC r,i8	OF BA /7 ib				6	6	7 NP	1m

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
BTC m,i8	OF BA /7 ib				8	8	8 NP	4m
BTR r,r	OF B3 /r				6	6	7 NP	1m
BTR m,r	OF B3 /r				13	13	13 NP	C
BTR r,i8	OF BA /6 ib				6	6	7 NP	1m
BTR m,i8	OF BA /6 ib				8	8	8 NP	4m
BTS r,r	OF AB /r				6	6	7 NP	1m
BTS m,r	OF AB /r				13	13	13 NP	C
BTS r,i8	OF BA /5 ib				6	6	7 NP	1m
BTS т, i8	OF BA /5 ib				8	8	8 NP	4m
CALL near im	E8 im	23	14	7	7	3	1 PV	4m
CALL near r	FF /2	20	13	7	7	5	2 NP	C
CALL near т	FF /2	29+ea	19	11	10	5	2 NP	C
CALL far im (RM)	9A im	36	23	13	17	18	4 NP	C
CALL far im (PM)	9A im			26	34	20	4..13 NP	C
CALL im (CG)	9A im			41	52	35	22 NP	C
CALL im (CG/in)	9A im			82	86	69	44 NP	C
CALL im (TS)	9A im			177	TS	37+TS	21+TS NP	C
CALL im (TG)	9A im			177	TS	27+TS	22+TS NP	C
CALL far т (RM)	FF /3	53+ea	38	16	22	17	4 NP	C
CALL far т (PM)	FF /3			29	38	20	5..14 NP	C
CALL т CG	FF /3			44	56	35	22 NP	C
CALL т CG/in	FF /3			83	90	69	44 NP	C
CALL т TS	FF /3				TS	37+TS	21+TS NP	C
CALL т TG	FF /3				TS	37+TS	22+TS NP	C
CBW	98	2	2	2	3	3	3 NP	1т
CDQ	99				2	3	2 NP	1т
CLC	F8	2	2	2	2	2	2 NP	1т
CLD	FC	2	2	2	2	2	2 NP	4т
CLI	-- FA	2	2	3	3	5	7 NP	C
CLTS	OF 06			2	5	7	10 NP	C
CMC	F5	2	2	2	2	2	2 NP	1т
CMOVcc r,r	OF 4cc /r							2т
CMOVcc r,m	OF 4cc /r							3т
CMP ac,im	3Cw im	4	4	3	2	1	1 UV	1т
CMP r,im	80sw /7 im	4	4	3	2	1	1 UV	1т
CMP m,im	80sw /7 im	14+ea	10	6	5	2	2 UV	2т

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
CMP r,r	38dw /r	3	3	2	2	1	1 UV	1m
CMP m,r	38dw /r	13+ea	10	7	5	2	2 UV	2m
CMP r,m	38dw /r	10+ea	10	6	6	2	2 UV	2m
CMPSB	A6	30	22	8	10	8	5 NP	C
REP* CMPSB	F2/3A6	9+30n	5+22n	5+9n	5+9n	7+7n	9+4n NP	C
CMPSW/ CMPSD	A7	30	22	8	10	8	5 NP	C
REP* CMPSW/D	F2/3A7	9+30n	5+22n	5+9n	5+9n	7+7n	9+4n NP	C
CMPXCHG r,r	OF BOw /r					6	5NP	C
CMPXCHG m,r	OF BOw /r					7..10	6 NP	C
CMPXCHG8B m64	0F C7 /1						10 NP	C
CPUID	0F A2					14	13 NP	C
CWD	99	5	4	2	2	3	2 NP	1т
CWDE	98				3	3	3 NP	1т
DAA	27	4	4	3	4	2	3 NP	1т
DAS	2F	4	4	3	4	2	3 NP	1т
DEC r16/32	48r	3	3	2	2	1	1 UV	1т
DEC r	FEw /1	3	3	2	2	1	1 UV	1т
DEC m	FEw /1	23+ea	15	7	6	3	3 UV	4т
DIV r8	F6w /6	80..90	29	14	14	16	17 NP	3m
DIV r16	F6w /6	144..162	38	22	22	24	25 NP	4т
DIV r32	F6w /6				38	40	41 NP	4т
DIV m8	F6w /6	86..96+ea	35	17	17	16	17 NP	4т
DIV m16	F6w /6	150..168+ea	44	25	25	24	25 NP	4т
DIV m32	F6w /6				41	40	41 NP	4т
EMMS	OF 77						NP	C
ENTER И6,0	C8 iw ib		15	11	10	14	11 NP	C
ENTER И6,1	C8 iw ib		25	15	12	17	15 NP	C
ENTER i16,i8	C8 iw ib	(m=n-1)	22+16m	12+4m	15+4m	17+3n	15+2n NP	C
F2XM1	D9 F0	310..630		310..630	211..476	140.279	13..57 NP	C
FABS	D9 E1	10..17		10..17	22	3	1 FX	1т
FADD m32	D8 /0	90..120+ea		90..120	24..32	8..20	3/1 FX	2т
FADD m64	DC /0	95..125+ea		95..125	29..37	8..20	3/1 FX	2т
FADD s0,si	D8 C0r	70..100		70..100	23..34	8..20	3/1 FX	1т
FADD si,s0	DC C0r	70..100		70..100	23..34	8..20	3/1 FX	1т
FADDP si,s0	DE C0r	75..105		75..105	23..31	8..20	3/1 FX	1т
FBLD m80	DF /4	290..310+ea		290..310	266..275	70..103	48..58 NP	C

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
FBSTP m80	DF /6	520..540+ea		520..540	512..534	172..176	148..154 NP	C
FCHS	D9 E0	10..17		10..17	24..25	6	1 FX	3m
FCLEX	9B DB E2	2..8		2..8	11	7	9 NP	3m
FCMOVB s0,si	DA C0r							2m
FCMOVE s0,si	DA C8r							2m
FCMOVBE s0,si	DA D0r							2m
FVMOVU s0,si	DA D8r							2m
FCMOVNB s0,si	DB C0r							2m
FCMOVNE s0,si	DB C8r							2m
FCMOVNBE s0,si	D8 D0r							2m
FCMOVNU s0,si	DB D8r							2m
FCOM m32	D8 /2	60..70+ea		60..70	26	4	4/1 FX	2m
FCOM m64	DC /2	65..75+ea		65..75	31	4	4/1 FX	2m
FCOM si	D8 D0r	40..50		40..50	24	4	4/1 FX	1m
FCOMP m32	D8 /3	60..70+ea		60..70	26	4	4/1 FX	2m
FCOMP m64	DC /3	65..75+ea		65..75	31	4	4/1 FX	2m
FCOMP si	D8 D8r	42..52		42..52	26	4	4/1 FX	1m
FCOMPP	DE D9	45..55		45..55	26	5	4/1 FX	1m
FCOMI s0,si	DB F0r							1m
FCOMIP so,si	DF F0r							1m
FCOS	D9 FF				123..772	257..354	18..124 NP	C
FDECSTP	D9 F6	6..12		6..12	22	3	1 NP	1m
FDIV m32	D8 /6	215..225+ea		215..225	89	73	39 FX	2m
FDIV m64	DC /6	220..230+ea		220..230	94	73	39 FX	2m
FDIV s0,si	D8 F0r	193..203		193..203	88..91	73	39 FX	1m
FDIV si,s0	DC F8r	193..203		193..203	88..91	73	39 FX	1m
FDIVP si,s0	DE F8r	197..207		197..207	91	73	39 FX	1m
FDIVR m32	D8 /7	216..226+ea		216..226	89	73	39 FX	2m
FDIVR m64	DC /7	221..231+ea		221..231	94	73	39 FX	2m
FDIVR s0,si	D8 F8r	194..204		194..204	88..91	73	39 FX	1m
FDIVR si,s0	DC F0r	194..204		194..204	88..91	73	39 FX	1m
FDIVRP si,s0	DE F0r	198..208		198..208	91	73	39 FX	1m
FDISI	9B DB E1	2..8		FNOP				
FENI	9B DB EO	2..8		FNOP				
FFREE si	DD C0r	9..16		9..16	18	3	1 NP	1m

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 **80387**	80486	P5	P6
FIADD m32	DA /0	102..137+ea		102..137	71..85	20..35	7/4 NP	C
FIADD т64	DE/0	108..143+ea		108..143	57..72	19..32	7/4 NP	C
FICOM m16	DE /2	72..86+ea		72..86	71..75	16..20	8/4 NP	C
FICOM т32	DA /2	78..91+ea		78..91	56..63	15..17	8/4 NP	C
FICOMP m16	DE /3	74..88+ea		74..88	71..75	16..20	8/4 NP	C
FICOMP т32	DA /3	80..93+ea		80..93	56..63	15..17	8/4 NP	C
FIDIV m16	DA /6	224..238+ea		224..238	136..140	85..89	42 NP	C
FIDIV т32	DE /6	230..243+ea		230..243	120..127	84..86	42 NP	C
FIDIVR т16	DA Я	225..239+ea		225..239	135..141	85..89	42 NP	C
FIDIVR m32	DE /7	231..245+ea		231..245	121..128	84..86	42 NP	C
FILD m16	DF/0	46..54+ea		46..54	61..65	13..16	3/1 NP	4т
FILD m32	DB /0	52..60+ea		52..60	45..52	9..12	3/1 NP	4т
FILD m64	DF /5	60..68+ea		60..68	56..67	10..18	3/1 NP	4т
FIMUL m16	DA /1	124..138+ea		124..138	76..87	23..27	7/4 NP	C
FIMUL m32	DE /1	130..144+ea		130..144	61..82	22..24	7/4 NP	C
FINCSTP	D9 F7	6..12		6..12	21	3	1 NP	1т
FINIT	9B DB E3	2..8		2..8	33	17	16 NP	C
FIST m16	DF /2	80..90+ea		80..90	82..95	29..34	6 NP	4т
FIST m32	DB /2	82..92+ea		82..92	79..93	28..34	6 NP	4т
FISTP m16	DF/3	82..92+ea		82..92	82..95	29..34	6 NP	4т
FISTP m32	DB /3	84..94+ea		84..94	79..93	28..34	6 NP	4т
FISTP m64	DF /7	94..105+ea		94..105	80..97	28..34	6 NP	4т
FISUB m16	DE /4	102..137+ea		102..137	71..85	20..35	7/4 NP	C
FISUB m32	DA /4	108..143+ea		108..143	57..82	19..32	7/4 NP	C
FISUBR m16	DE /5	102..137+ea		102..137	71..85	20..35	7/4 NP	C
FISUBR m32	DA /5	108..143+ea		108..143	57..82	19..32	7/4 NP	C
FLD m32	D9 /0	38..56+ea		38..56	20	4	1 FX	1т
FLD m64	DD /0	40..60+ea		40..60	25	3	1 FX	1m
FLD m80	DB /5	53..65+ea		53..65	44	3	3 NP	4т
FLD si	D9 C0r	17..22		17..22	14	4	1 FX	1т
FLD1	D9 E8	15..21		15..21	24	4	2 NP	2т
FLDL2T	D9 E9	16..22		16..22	40	8	5/3 NP	2т
FLDL2E	D9 EA	15..21		15..21	40	8	5/3 NP	2т
FLDPI	D9 EB	16..22		16..22	40	8	5/3 NP	2т
FLDLG2	D9 EC	18..24		18..24	41	8	5/3 NP	2т
FLDLN2	D9 ED	17..23		17..23	41	8	5/3 NP	2т
FLDZ	D9 EE	11..17		11..17	20	4	2 NP	1т
FLDCW m16	D9 /5	7..14+ea		7..14	19	4	7 NP	3m
FLDENV т 14	D9 /4	35..45+ea		35..45	71	44	37 NP	C
FLDENV m14 (PM)	D9 /4				71	34	32..33 NP	C

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
FMUL m32	D8 /1	110..125+ea		110..125	27..35	11	3/1 FX	2m
FMUL т64	DC /1	154..168+ea		154..168	32..57	14	3/1 FX	2m
FMUL s0,si	D8 C8r	90..145+ea		90..145	29..57	16	3/1 FX	1m
FMUL si,s0	DC C8r	90..145+ea		90..145	29..57	16	3/1 FX	1m
FMULP si,s0	DE C8r	94..148+ea		94..148	29..57	16	3/1 FX	1m
FNCLEX	DB E2	2..8		2..8	11	7	9/9 NP	3m
FNDISI	DB E1	2..8		FNOP				
FNENI	DB EO	2..8		FNOP				
FNINIT	DB E3	2..8		2..8	33	17	12 NP	C
FNOP	D9 D0	10..16		10..16	12	3	1 NP	1m
FNSAVE т (RM)	DD /6	197..207+ea		197..207	375..376	154	127..151 NP	C
FNSAVE т (PM)	DD /6				375..376	143	124 NP	C
FNSETPM	DB E4			2..8	FNOP			
FNSTCW m16	D9 /7	12..18		12..18	15	3	2 NP	3m
FNSTENV m16	D9 /6	40..50+ea		40..50	103..104	56..67	48..50 NP	C
FSTSW m16	9B DD /7	12..18		12..18	15	3	2 NP	3т
FSTSW AX	9B DF EO			10..16	13	3	2 NP	3т
FPATAN	D9 F3	250..800		250..800	314..487	218..303	19..134 NP	C
FPREM	D9 F8	15..190		15..190	74..155	70..138	16..64 NP	C
FPREM1	D9F5				95..185	72..167	20..70 NP	C
FPTAN	D9 F2	30..540		30..540	191..497	200..273	17..173 NP	C
FRNDINT	D9 FC	16..50		16..50	66..80	21..30	9..20 NP	C
FRSTOR т (RM)	DD /4	197..207+ea		197..207	308	131	75..95 NP	C
FRSTOR т (PM)	DD /4				308	120	70 NP	C
FSAVE т (RM)	9B DD /6	197..207+ea		197..207	375..376	154	127..151 NP	C
FSAVE т (PM)	9B DD /6				375..376	143	124 NP	C
FSCALE	D9 FD	32..38		32..38	67..86	30..32	20..31 NP	C
FSETPM	9B DB E4			2..8	FNOP			
FSIN	D9 FE				122..771	257..354	16..126 NP	C
FSINCOS	D9 FB				194..809	292..365	17..137 NP	C
FSQRT	D9 FA	180..186		180..186	122..129	83..87	70 NP	1т
FST m32	D9 /2	84..90+ea		84..90	44	7	2 NP	2т
FST m64	DD /2	96..104+ea		96..104	45	8	2 NP	2т
FST si	DD D0r	15..22		15..22	11	3	1 NP	1т

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
FSTP m32	D9 /3	86..92+ea		86..92	44	7	2 NP	2m
FSTP m64	DD /3	98..106+ea		98..106	45	8	2 NP	2m
FSTP т80	DB /7	52..58+ea		52..58	53	6	3 NP	C
FSTP si	DD D8r	17..24		17..24	12	3	1 NP	1m
FSTCW m16	9B D9 /7	12..18		12..18	15	3	2 NP	3m
FSTENV т	9B D9 /6	40..50+ea		40..БО	103..104	56..67	48..50 NP	C
FSTSW m16	9B DD /7	12..18		12..18	15	3	2 NP	3m
FSTSW AX	9B DF EO			10..16	13	3	2 NP	3m
FSUB m32	D8 /4	90..120+ea		90..120	24..32	8..20	3/1 FX	2m
FSUB m64	DC /4	95..125+ea		95..125	28..36	8..20	3/1 FX	2m
FSUB s0,si	D8 E0r	70..100		70..100	26..37	8..20	3/1 FX	1m
FSUB si,s0	DC E8r	70..100		70..100	26..37	8..20	3/1 FX	1m
FSUBP si,s0	DE E8r	75..105		75..105	26..34	8..20	3/1 FX	1m
FSUBR m32	D8 /5	90..120+ea		90..120	24..32	8..20	3/1 FX	2m
FSUBR m64	DC /5	95..125+ea		95..125	28..36	8..20	3/1 FX	2m
FSUBR s0,si	D8 E8r	70..100		70..100	26..37	8..20	3/1 FX	1m
FSUBR si,s0	DC E0r	70..100		70..100	26..37	8..20	3/1 FX	1m
FSUBRP si,s0	DE E0r	75..105		75..105	26..34	8..20	3/1 FX	1m
FTST	D9 E4	38..48		38..48	28	4	4/1 FX	1m
FUCOM si	DD E0r				24	4	4/1 FX	1m
FUCOMP si	DD E8r				26	4	4/1 FX	1m
FUCOMPP	DA E9				26	5	4/1 FX	2m
FUCOMI s0,si	DB E8r							1m
FUCOMIP s0,si	DF E8r							1m
FWAIT	9B	4		3	6	1..3	1..3 NP	2m
FXAM	D9 E5	12..23		12..23	30..38	8	21 NP	1m
FXCH si	D9 C8r	10..15		10..15	18	4	1 PV	1m
FXRSTOR т	OF AE /1							C
FXTRACT	D9 F4	27..55		27..55	79..76	16..20	13 NP	C
FXSAVE т	OF AE /0							C
FYL2X	D9 F1	900..1100		900..1100	120..538	196..329	22..111 NP	C
FYL2XP1	D9 F9	700..1000		700..1000	257..547	171..326	22..103 NP	C
HLT	F4	2	2	2	5	4	4 NP	C
IDIV r8	F6w /7	101..112	44..52	17	19	19	22 NP	3m
IDIV r16	F6w /7	165..184	53..61	25	27	27	30 NP	4т
IDIV r32	F6w /7				43	43	46 NP	4т
IDIV m8	F6w /7	107..118+ea	50..58	20	22	20	22 NP	4т
IDIV m16	F6w /7	171..190+ea	59..67	28	30	28	30 NP	4т

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
IDIV т32	F6w /7				46	44	46 NP	4m
IMUL r8	F6w /5	80..98	25..28	13	9..14	13..18	11 NP	1m
IMUL r16	F6w /5	128..154	34..37	21	9..22	13..26	11 NP	3m
IMUL r32	F6w /5				9..38	13..42	10 NP	3m
IMUL m8	F6w /5	86..104+ea	32..34	16	12..17	13..18	11 NP	2m
IMUL m16	F6w /5	134..160+ea	40..43	24	12..25	13..26	11 NP	4m
IMUL r32	F6w /5				12..41	13..42	10 NP	4m
IMUL r,r,i8	6B /r ib		22	21	9..14	13..18	10 NP	1m
IMUL r16,r16,i16	69 /r im		29	21	9..22	13..26	10 NP	1m
IMUL r32,r32,i32	69 /r im				9..38	13..42	10 NP	1m
IMUL r,r,i8	6B /r ib		25	24	12..17	13..18	10 NP	2m
IMUL r16,m16,i16	69 /r im		32	24	12..25	13..26	10 NP	2m
IMUL r32,m32,i32	69 /r im				12..41	13..42	10 NP	2m
IMUL r16, r16	OF AF /r				9..22	13.18	10 NP	1m
IMUL r32,r32	OF AF /r				9..38	13..42	10 NP	1m
IMUL r16, m16	OF AF /r				12..25	13..18	10 NP	2m
IMUL r32, m32	OF AF /r				12..41	13..42	10 NP	2m
IN ac,i8 (RM)	E4w ib	14	10	5	12	14	7 NP	C
IN ac,i8 (CPL≤IOPL)	E4w ib				6	9	4 NP	C
IN ac,i8 (CPL>IOPL)	E4w ib				26	29	21 NP	C
IN ac,i8 (V86)	E4w ib				26	27	19 NP	C
IN ac,DX (RM)	ECw	12	8	5	13	14	7 NP	C
IN ac,DX (CPL≤IOPL)	ECw				7	8	4 NP	C
IN ac,DX (CPL>IOPL)	ECw				27	28	21 NP	C
IN ac,DX (V86)	ECw				27	27	19 NP	C
INC r	FEw /0	3	3	2	2	1	1 UV	1т
INC r16/32	40r	3	3	2	2	1	1 UV	1т
INC т	FEw /0	23+ea	15	7	6	3	3 UV	4m
INS* (RM)	6Cw		14	5	15	17	9 NP	C

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
INS* (CPL≤IOPL)	6Cw				9	10	6 NP	C
INS* (CPL>IOPL)	6Cw				29	32	24 NP	C
INS* (V86)	6Cw				29	30	22 NP	C
INT i8 (RM)	CD ib	71	47	23	37	30	16 NP	3m
INT i8 (PM)	CD ib			40	59	44	31 NP	C
INT i8 (PM/in)	CD ib			78	99	71	48 NP	C
INT i8 (V86)	CD ib			78	119	82	60 NP	C
INT i8 (TG)	CD ib			167	TS	37+TS	23+TS NP	C
INT3 (RM)	CC	72	45	23	33	26	16 NP	C
INT3 (PM)	CC			40	59	44	30 NP	C
INT3 (PM/in)	CC			78	99	71	47 NP	C
INT3 (V86)	CC			78	119	82	59 NP	C
INT3 (TG)	CC			167	TS	37+TS	22+TS NP	C
INTO (OF=0)	CE	4	4	3	3	3	4 NP	C
INTO (RM)	CE	73	48	24	35	28	13 NP	C
INTO (PM)	CE				59	46	30 NP	C
INTO (PM/in)	CE				99	73	47 NP	C
INTO (V86)	CE				118	84	59 NP	C
INTO (TG)	CE				TS	29+TS	22+TS NP	C
INVD	OF 08					4	15 NP	C
INVLPG m	OF 01 /7					12	25 NP	C
IRET/IRETD (RM)	CF	44	28	17	22	15	7 NP	C
IRET/IRETD (PM)	CF			31	38	15	10..19 NP	C
IRET/IRETD (PM/out)	CF			55	82	36	27 NP	C
RET/IRETD (PM/NT)	CF			169	TS	32+TS	10+TS	C
Jcc i8 (не вып.)	70c ib	4	4	3	3	!	1 PV	1т
Jcc i8 (вып.)	70c ib	16	13	7	7	3	1 PV	1т
Jcc im (не вып.)	OF 80c ib				3	1	1 PV	1т
Jcc im (вып.)	OF 80c ib				3	1	1 PV	1т
JCXZ i8 (не вып.)	E3 ib	6	5	4	5	5	5 NP	2т
JCXZ i8 (вып.)	E3 ib	18	16	8	9	8	6 NP	2т
JMP near i8	EB ib	15	13	7	7	3	1 PV	1т
JMP near i16/32	"E9 ib	15	13	7	7	3	1 PV	1т

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
JMP near r	FF /4	11	11	7	7	5	2 NP	2m
JMP near m	FF /4	18+ea	17	11	10	5	2 NP	2m
JMP far im (RM)	EA im	15	13	11	12	17	3 NP	C
JMP far im (PM)	EA im				27	19	23 NP	C
JMP far im (CG)	EA im			38	45	32	18 NP	C
JMP far im (TS)	EA im			175	TS	42+TS	19+TS NP	C
JMP far im (TG)	EA im			180	TS	43+TS	20+TS NP	C
JMP far m (RM)	FF /5	24+ea	26	15	12	13	4 NP	C
JMP far m (PM)	FF /5				31	18	23 NP	C
JMP far m (CG)	FF /5			41	49	31	18 NP	C
JMP far m (TS)	FF /5			178	5+TS	41+TS	19+TS NP	C
JMP far m (TG)	FF /5			183	5+TS	42+TS	20+TS NP	C
LAHF	9F	4	2	2	2	3	2 NP	1т
LAR r,r	OF 02 /r			14	15	11	8 NP	C
LAR r,m	OF 02 /r			16	16	11	8 NP	C
LDS r,m (RM)	C5 /r	24+ea	18	7	7	6	4 NP	C
LDS r,m (PM)	C5 /r				22	12	4..13 NP	C
LEA r,m	8D /r	2+ea	6	3	2	1..2	1 UV	1т
LEAVE	C9		8	5	4	5	3 NP	3m
LES r,m (RM)	C4 /r	24+ea	18	7	7	6	4 NP	C
LES r,m (PM)	C4 /r				22	12	4..13 NP	C
LFS r,m (RM)	OF B4 /r				7	6	4 NP	C
LFS r,m (PM)	0F B4 /r				22	12	4..13 NP	C
LGDT m	OF 01 /2			11	11	11	6 NP	C
LGS r,m (RM)	OF B5 /r				7	6	4 NP	C
LGS r,m (PM)	OF B5 /r				22	12	4..13 NP	C
LIDT m	OF 01 /3			12	11	11	6 NP	C
LLDT r	OF 00 /2			17	20	11	9 NP	C
LLDT m	OF 00 /2			19	24	11	9 NP	C
LMSW r	OF 01 /6			3	10	13	8 NP	C
LMSW m	OF 01 /6			6	13	13	8 NP	C
LOCK	F0	2	2	0	0	1	1 NP	C

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
LODSB/ LODSW/ LODSD	ACw	16	10	5	5	5	2 NP	2m
LOOP i8 (не вып.)	E2 ib	5	5	4	11	6	5 NP	4m
LOOP i8 (вып.)	E2 ib	17	15	8	11	7	6 NP	4m
LOOPE i8 (не вып.)	E1 ib	6	5	4	11	6	7 NP	4m
LOOPE i8 (вып.)	E1 ib	18	16	8	11	9	8 NP	4m
LOOPNE i8 (не вып.)	E0 ib	5	5	4	11	6	7 NP	4m
LOOPNE i8 (вып.)	E0 ib	19	16	8	11	9	8 NP	4m
LSL r,r	OF 03 /r			14	20 или 25	10	8 NP	C
LSL r,m	OF 03 /r			16	21 или 26	10	8 NP	C
LSS r,m (RM)	OF B2 /r				7	6	4 NP	C
LSS r,m (PM)	0F B2 /r				22	12	8..17 NP	C
LTR r	OF 00 /3			17	23	20	10 NP	C
LTR m	OF 00 /3			19	27	20	10 NP	C
MOV r,r	88dw /r	2	2	2	2	1	1.UV	1т
MOV m,ac	AOdw im	14	9	3	2	1	1 UV	2т
MOV m,r	88dw /r	13+ea	9	3	2	1	1 UV	2т
MOV ac,m	AOdw im	14	8	5	4	1	1 UV	1т
MOV r,m	88dw /r	12+ea	12	5	4	1	1 UV	1m
MOV m,im	C6w /0 im	14+ea	12..13	3	2	1	1 UV	2т
MOV r8,i8	B0r ib	4	3..4	2	2	1	1 UV	1т
MOV r16/32,i16/32	B8r ib	4	3..4	2	2	1	1 UV	1т
MOV sr,r	8E /r	2	2	2	2	3	2..11 NP	4т
MOV sr,r (PM)	8E /r			17	18	9	2..11 NP	4т
MOV sr,m	8E /r	12+ea	9	5	5	3	3..12 NP	4т
MOV sr,m (PM)	8E /r			19	19	9	3..12 NP	4т
MOV r,sr	8C /r	2	2	2	2	3	1 NP	3m
MOV m,sr	8C /r	13+ea	11	3	2	3	1 NP	1т
MOV CR0,r	OF 22 /r				10	16	22 NP	C
MOV CR2,r	OF 22 /r				4	4	10 NP	C

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
MOV CR3,r	OF 22 /r				5	4	21 NP	C
MOV CR4,r	OF 22 /r						14 NP	C
MOV r,CRx	OF 20 /r				6	4	4 NP	C
MOV DRO-3,r	OF 23 /r				22	11	11 NP	C
MOV DR4-5,r	OF 23 /r						12 NP	C
MOV DR6-7,r	OF 23 /r				16	11	11 NP	C
MOV r,DRO-3	OF 21 /r				22	10	2 NP	C
MOV r,DR4-5	OF 21 /r						12 NP	C
MOV r,DR6-7	OF 21 /r				22	10	11 NP	C
MOV TR6,r	OF 26 /6			12	4			
MOV TR7,r	OF 26 /7			12	4			
MOV TR3,r	OF 26 /3				6			
MOV r,TR6	OF 24 /6			12	4			
MOV r,TR7	OF 24 /7			12	4			
MOV r,TR3	OF 24 /3				3			
MOVD mm,rm32	OF 6E /r						PU	1т
MOVD r32,mm	OF 6E /r						PU	1т
MOVD m32,mm	OF 6E /r						PU	2т
MOVQ m64,mm	OF 7F /r						PU	2т
MOVQ mm,m64	OF 6F /r						PU	1т
MOVQ mm,mm	OF *F /r						PU	1т
MOVS*	A4w	18	5	7	7	4	4 NP	C
MOVSX r,r	OF BEw /r				3	3	3 NP	1т
MOVSX r,m	OF BEw /r				6	3	3 NP	1т
MOVZX r,r	OF B6w /r				3	3	3 NP	1т
MOVZX r,m	OF B6w /r				6	3	3 NP	1т
MUL r8	F6w /4	70..77	26..28	13	9..14	13..18	11 NP	1т
MUL r16	F6w /4	118..133	35..37	21	9..22	13..26	11 NP	3m
MUL r32	F6w /4				9..38	13..42	10 NP	3m
MUL m8	F6w /4	76..83+ea	32..34	16	12..17	13..18	11 NP	2т
MUL m16	F6w /4	124..139+ea	41..43	24	12..25	13..26	11 NP	4т
MUL m32	F6w /4				12..41	13..42	10 NP	4т
NEG r	F6w /3	3	3	2	2	1	1 NP	1т
NEG m	F6w /3	24+ea	13	7	6	3	3 NP	4т
NOP	90	3	3	3	3	1	1 UV	1т

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
NOT r	F6w /2	3	3	2	2	1	1 NP	1m
NOT m	F6w /2	24+ea	13	7	6	3	3 NP	4m
OR ac,im	0Cw im	4	4	3	2	1	1 UV	1m
OR r,im	80sw /1 im	4	4	3	2	1	1 UV	1m
OR m,im	80sw /1 im	23+ea	16	7	7	3	3 UV	4m
OR r,r	08dw /r	3	3	2	2	1	1 UV	1m
OR m,r	08dw /r	24+ea	10	7	7	3	3 UV	4m
OR r,m	08dw /r	13+ea	10	7	6	2	2 UV	2m
OUT i8,ac (RM)	E6w ib	14	9	3	10	16	12 NP	C
OUT i8,ac (CPL?IOPL)	E6w ib				4	11	9 NP	C
OUT i8,ac (CPL≤IOPL)	E6w ib				24	31	26 NP	C
OUT i8,ac (V86)	E6w ib				24	29	24 NP	C
OUT DX,ax (RM)	EEw	12	7	3	11	16	12 NP	C
OUT DX,ac (CPL≤IOPL)	EEw				5	10	9 NP	C
OUT DX,ac (CPL>IOPL)	EEw				25	30	26 NP	C
OUT DX,ac (V86)	EEw				25	29	24 NP	C
OUTS* (RM)	6Ew		14	5	14	17	13 NP	C
OUTS* (CPL≤IOPL)	6Ew				8	10	10 NP	C
OUTS* (CPL>IOPL)	6Ew				28	32	27 NP	C
OUTS* (V86)	6Ew				28	30	25 NP	C
PACKSSWB mm,mm	OF 63 /r						UV NP1	1т
PACKSSWB mm,m64	OF 63 /r						PU NP1	2т
PACKSSDW mm,mm	OF 6B /r						UV NP1	1т
PACKSSDW mm,m64	OF 6B /r						PU NP1	2т
PACKUSWB mm,mm	OF 67 /r						UV NP1	1т
PACKUSWB mm,m64	OF 67 /r						PU NP1	2т

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
PADDB mm,mm	OF FC /r						UV	1m
PADDB mm,m64	OF FC /r						PU	2m
PADDW mm,mm	OF FD /r						UV	1m
PADDW mm,m64	OF FD /r						PU	2m
PADDD mm,mm	OF FE /r						UV	1m
PADDD mm,m64	OF FE /r						PU	2m
PADDSB mm,mm	OF EC /r						UV	1m
PADDSB mm,m64	OF EC /r						PU	2m
PADDSW mm,mm	OF ED /r						UV	1m
PADDSW mm,m64	OF ED /r						PU	2m
PADDUSB mm,mm	OF DC /r						UV	1m
PADDUSB mm,m64	OF DC /r						PU	2m
PADDUSW mm,mm	OF DD /r						UV	1m
PADDUSW mm,m64	OF DD /r						PU	2m
PAND mm,mm	OF DB /r						UV	1m
PAND mm,m64	OF DB /r						PU	2m
PANDN mm,mm	OF DF /r						UV	1m
PANDN mm,m64	OF DF /r						PU	2m
PCMPEQB mm,mm	OF 74 /r						UV	1m
PCMPEQB mm,m64	OF 74 /r						PU	2m
PCM PEQW mm,mm	OF 75 /r						UV	1m

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
PCMPEQW mm,m64	OF 75 /r						PU	2m
PCMPEQD mm,mm	OF 76 /r						UV	1m
PCMPEQD mm,m64	OF 75 /r						PU	2m
PCMPGTB mm,mm	OF 64 /r						UV	1m
PCMPGTB mm,m64	OF 64 /r						PU	2m
PCMPGTW mm,mm	OF 65 /r						UV	1m
PCMPGTW mm,m64	OF 65 /r						PU	2m
PCMPGTD mm,mm	OF 66 /r						UV	1m
PCMPGTD mm,m64	OF 66 /r						PU	2m
PMADDWD mm,mm	OF F5 /r						UV NP2	1m
PMADDWD mm,m64	OF F5 /r						PU NP2	2m
PMULHW mm,mm	OF E5 /r						UV NP2	1m
PMULHW mm,m64	OF E5 /r						PU NP2	2m
PMULLW mm,mm	0F D5 /r						UV NP2	1m
PMULLW mm,m64	OF D5 /r						PU NP2	2m
POP r	58r	12	10	5	4	1	1 UV	2m
POP m	8F/0	25+ea	20	5	5	6	2 NP	C
POP DS (RM)	1F	12	8	5	7	3	3 NP	C
POP DS (PM)	1F			20	21	9	3..12 NP	C
POP ES (RM)	7	12	8	5	7	3	3 NP	C
POP ES (PM)	7			20	21	9	3..12 NP	C
POP SS (RM)	17	12	8	5	7	3	3 NP	C
POP SS (PM)	17			20	21	9	8..17 NP	C
POP FS (RM)	0F A1	12	8	5	7	3	3 NP	C
POP FS (PM)	0F A1				21	9	3..12 NP	C
POP GS (RM)	0F A9	12	8	5	7	3	3 NP	C
POP GS (PM)	0F A9				21	9	3..12 NP	C

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
POPA/POPAD	61		51	19	24	9	5 NP	C
POPF/POPFD (RM)	9D	12	8	5	5	9	4 NP	C
POPF/POPFD (PM)	9D				5	6	14 NP	C
POR mm,mm	OF EB /r						UV	1m
POR mm,m64	OF EB /r						PU	2m
PSLLW mm,mm	OF F1 /r						UV NP1	1m
PSLLW mm,m64	OF F1 /r						PU NP1	1m
PSLLW mm,i8	OF 71 /6 ib						UV NP1	2m
PSLLD mm,mm	OF F2 /r						UV NP1	1m
PSLLD mm,m64	OF F2 /r						PU NP1	1m
PSLLD mm,i8	OF 72 /6 ib						UV NP1	2m
PSLLQ mm,mm	OF F3 /r						UV NP1	1m
PSLLQ mm,m64	OF F3 /r						PU NP1	1m
PSLLQ mm,i8	OF 73 /6 ib						UV NP1	2m
PSRAW mm,mm	OF E1 /r						UV NP1	1m
PSRAW mm,m64	OF E1 /r						PU NP1	1m
PSRAW mm,i8	OF 71 /4 ib						UV NP1	2m
PSRAD mm,mm	OF E2 /r						UV NP1	1m
PSRAD mm,m64	OF E2 /r						PU NP1	1m
PSRAD mm,i8	OF 72 /4 ib						UV NP1	2m
PSRLW mm,mm	OF D1 /r						UV NP1	1m
PSRLW mm,m64	OF D1 /r						PU NP1	1m
PSRLW mm,i8	OF 71 /2 ib						UV NP1	2m
PSRLD mm,mm	OF D2 /r						UV NP1	1m
PSRLD mm,m64	OF D2 /r						PU NP1	1m

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
PSRLD mm,i8	OF 72 /2 ib						UV NP1	2m
PSRLQ mm,mm	OF D3 /r						UV NP1	1m
PSRLQ mm,m64	OF D3 /r						PU NP1	1m
PSRLQ mm,i8	OF 73 /2 ib						UV NP1	2m
PSUBB mm,mm	OF F8 /r						UV	1m
PSUBB mm,m64	OF F8 /r						PU	2m
PSUBW mm,mm	OF F9 /r						UV	1m
PSUBW mm,m64	OF F9 /r						PU	2m
PSUBD mm,mm	OF FA /r						UV	1m
PSUBD mm,m64	OF FA /r						PU	2m
PSUBSB mm,mm	OF E8 /r						UV	1m
PSUBSB mm,m64	OF E8 /r						PU	2m
PSUBSW mm,mm	OF E9 /r						UV	1m
PSUBSW mm,m64	OF E9 /r						PU	2m
PSUBUSB mm,mm	OF D8 /r						UV	1m
PSUBUSB mm,m64	OF D8 /r						PU	2m
PSUBUSW mm,mm	OF D9 /r						UV	1m
PSUBUSW mm,m64	OF D9 /r						PU	2m
PUNPCKHBW mm,mm	OF 68 /r						UV NP1	1m
PUNPCKHBW mm,m64	OF 68 /r						PU NP1	2m
PUNPCKHWD mm,mm	OF 69 /r						UV NP1	1m

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
PUNPCKHWD mm,m64	OF 69 /r						PU NP1	2m
PUNPCKHDQ mm,mm	OF 6A /r						UV NP1	1m
PUNPCKHDQ mm,m64	OF 6A /r						PU NP1	2m
PUNPCKLBW mm,mm	OF 60 /r						UV NP1	1m
PUNPCKLBW mm,m64	OF 60 /r						PU NP1	2m
PUNPCKLWD mm,mm	OF 61 /r						UV NP1	1m
PUNPCKLWD mm,m64	OF 61 /r						PU NP1	2m
PUNPCKLDQ mm,mm	OF 62 /r						UV NP1	1m
PUNPCKLDQ mm,m64	OF 62 /r						PU NP1	2m
PUSH r	50r	15	10	3	2	1	1 UV	3m
PUSH т	FF /6	24+ea	16	5	5	4	2 NP	4m
PUSH i8	6A ib			3	2	1	1 NP	3m
PUSH i16/32	68 im			3	2	1	1 NP	3m
PUSH CS	0E	14	9	3	2	3	1 NP	4m
PUSH DS	1E	14	9	3	2	3	1 NP	4m
PUSH ES	6	14	9	3	2	3	1 NP	4m
PUSH SS	16	14	9	3	2	3	1 NP	4m
PUSH FS	OF AO				2	3	1 NP	4m
PUSH GS	0F A8				2	3	1 NP	4m
PUSHA/ PUSHAD	60		36	17	18	11	5 NP	C
PUSHF/ PUSHFD (RM)	9C	14	9	3	4	4	9 NP	C
PUSHF/ PUSHFD (PM)	9C			3	4	3	3 NP	C
PXOR mm,mm	OF EF /r						UV	1m
PXOR mm,m64	OF EF /r						PU	2m
RCL r,1	D0w /2	2	2	2	9	3	1 PU	2m
RCL m,1	D0w /2	23+ea	15	7	10	4	3 PU	4m
RCLr,CL	D2w /2	8+4n	5+n	5+n	9	8..30	7..24 NP	C

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
RCL m,CL	D2w /2	28+ea+4n	17+n	8+n	10	9..31	9..26 NP	C
RCL r,i8	COw /2 ib		5+n	5+n	9	8..30	8..25 NP	C
RCL m, i8	COw /2 ib		17+n	8+n	10	9..31	10..27 NP	C
RCR r,1	D0w /3	2	2	2	9	3	1 PU	2m
RCR m,1	D0w /3	23+ea	15	7	10	4	3 PU	4m
RCR r,CL	D2w /3	8+4n	5+n	5+n	9	8..30	7..24 NP	C
RCR m,CL	D2w /3	28+ea+4n	17+n	8+n	10	9..31	9..26 NP	C
RCR r,i8	COw /3 ib		5+n	5+n	9	8..30	8..25 NP	C
RCR m,i8	COw /3 ib		17+n	8+n	10	9..31	10..27 NP	C
RDMSR	OF 32						20..24 NP	C
RDPMC	OF 33							C
RDTSC	OF 31						20..24 NP	C
RETN	C3	20	16	11	10	5	2 NP	4т
RETN i16	C2 iw	24	18	11	10	5	3 NP	C
RETF (RM)	CB	34	22	15	18	13	4 NP	4т
RETF (PM)	CB			25	32	18	4..13 NP	C
RETF (PM/out)	CB			55	62	33	23 NP	C
RETF i16 (RM)	CA iw	33	25	15	18	14	4 NP	C
RETF i16 (PM)	CA iw			25	32	17	4..13 NP	C
RETF i16 (PM/out)	CA iw			55	68	33	23 NP	C
ROL r,1	D0w /0	2	2	2	3	3	1 PU	1т
ROL m,1	D0w /0	23+ea	15	7	7	4	3 PU	4т
ROL r,CL	D2w /0	8+4n	5+n	5+n	3	3	4 NP	1т
ROL m,CL	D2w /0	28+ea+4n	17+n	8+n	7	4	4 NP	4т
ROL r,i8	COw /0 ib		5+n	5+n	3	2	1 PU	1т
ROL m,i8	COw /0 ib		17+n	8+n	7	4	3 PU	4т
ROR r,1	D0w /1	2	2	2	3	3	1 PU	1т
ROR m,1	D0w /1	23+ea	15	7	7	4	3 PU	4т
ROR r,CL	D2w /1	8+4n	5+n	5+n	3	3	4 NP	1т
ROR m,CL	D2w /1	28+ea+4n	17+n	8+n	7	4	4 NP	4т
ROR r,i8	COw /1 ib		5+n	5+n	3	2	1 PU	1т
ROR m,i8	COw /1 ib		17+n	8+n	7	4	3 PU	4т
RSM	0F AA						83 NP	C
SAHF	9E	я	3	2	3	2	2 NP	1т
SAL r,1	D0w /4	2	2	2	3	3	1 PU	1т
SAL m,1	D0w /4	23+ea	15	7	7	4	3 PU	4т
SAL r,CL	D2w /4	8+4n	5+n	5+n	3	3	4 NP	1т
SAL m,CL	D2w /4	28+ea+4n	17+n	8+n	7	4	4 NP	4т
SAL r,i8	COw /4 ib		5+n	5+n	3	2	1 PU	1т

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
SAL m, i8	COw /4 ib		17+n	8+n	7	4	3 PU	4m
SALC	D6	7	?	3	3	3	3 NP	7
SAR r,1	D0w /7	2	2	2	3	3	1 PU	1m
SAR m,1	D0w /7	23+ea	15	7	7	4	3 PU	4m
SAR r.CL	D2w /7	8+4n	5+n	5+n	3	3	4 NP	1m
SAR m,CL	D2w /7	28+ea+4n	17+n	8+n	7	4	4 NP	4m
SAR r,i8	COw /7 ib		5+n	5+n	3	2	1 PU	1m
SAR m,i8	COw /7 ib		17+n	8+n	7	4	3 PU	4m
SBB ac,im	1Cwim	4	4	3	2	1	1 PU	2m
SBB r,im	80sw /3 im	4	4	3	2	1	1 PU	2m
SBB m,im	80sw /3 im	23+ea	16	7	7	3	3 PU	4m
SBB r,r	18dw /r	3	3	2	2	1	1 PU	2m
SBB m,r	18dw /r	24+ea	10	7	7	3	3 PU	4m
SBB r,m	18dw /r	13+ea	10	7	6	2	2 PU	3m
SCASB/ SCASW/ SCASD	AEw	18	15	7	7	6	4 NP	3m
REP* SCAS*	F2/3 AEw	9+15n	5+15n	5+8n	5+8n	7+5n	8+5n NP	C
SETcc r8 (вып.)	0F 9cc				4	4	1 NP	1m
SETcc r8 (не вып.)	0F 9cc				4	3	3 NP	3m
SETcc m8 (вып.)	0F 9cc				5	3	1 NP	1m
SETcc m8 (не вып.)	0F 9cc				5	4	3 NP	3m
SGDT m	OF 01 /0			11	9	10	4 NP	4m
SHL r,1	D0w /4	2	2	2	3	3	1 PU	1m
SHL m,1	D0w /4	23+ea	15	7	7	4	3 PU	4m
SHL r,CL	D2w /4	8+4n	5+n	5+n	3	3	4 NP	1m
SHL m,CL	D2w /4	28+ea+4n	17+n	8+n	7	4	4 NP	4m
SHL r,i8	COw /4 ib		5+n	5+n	3	2	1 PU	1m
SHL m,i8	COw /4 ib		17+n	8+n	7	4	3 PU	4m
SHR r,1	D0w /5	2	2	2	3	3	1 PU	1m
SHR m,1	D0w /5	23+ea	15	7	7	4	3 PU	4m
SHR r,CL	D2w /5	8+4n	5+n	5+n	3	3	4 NP	1m
SHR m,CL	D2w /5	28+ea+4n	17+n	8+n	7	4	4 NP	4m
SHR r,i8	COw /5 ib		5+n	5+n	3	2	1 PU	1m
SHR m,i8	COw /5 ib		17+n	8+n	7	4	3 PU	4m
SHLD r,r,im	0F A4				3	2	4 NP	2m
SHLD r,r,CL	0F A5				3	3	4 NP	2m

Таблица 27. Команды (продолжение)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
SHLD m,r,im	0F A4				7	3	4 NP	4m
SHLD m,r,CL	0F A5				7	4	5 NP	4m
SHRD r,r,im	OF AC				3	?	4 NP	2m
SHRD r,r,CL	OF AD				3	3	4 NP	2m
SHRD m,r,im	OF AC				7	3	4 NP	4m
SHRD m,r,CL	OF AD				7	4	5 NP	4m
SIDT m	OF 01 /1			12	9	10	4 NP	C
SLDT r16	OF 00 /0			2	2	2	2 NP	4m
SLDT m16	OF 00 /0			3	2	3	2 NP	C
SMSW r16	OF 01 /4			2	2	2	4 NP	4m
SMSW m16	OF 01 /4			3	3	3	4 NP	C
STC	F9	2	2	2	2	2	2 NP	1m
STD	FD	2	2	2	2	2	2 NP	4m
STI	FB	2	2	2	3	5	7 NP	C
STOSB/ STOSW/ STOSD	AAw	11	10	3	4	5	3 NP	3m
REP STOS*	F3 AAw	9+14n	6+9n	4+3n	5+5n	7+4n	3+n NP	C
STR r16	OF 00 /1			2	2	2	2 NP	4m
STR m16	OF 00 /1			3	2	3	2 NP	C
SUB ac,im	2Cw im	4	4	3	2	1	1 UV	1m
SUB r,im	80sw /5 im	4	4	3	2	1	1 UV	1m
SUB m,im	80sw /5 im	23+ea	16	7	7	3	3 UV	4m
SUB r,r	28dw /r	3	3	2	2	1	1 UV	1m
SUB m,r	28dw /r	24+ea	10	7	7	3	3 UV	4m
SUB r,m	28dw /r	13+ea	10	7	6	2	2 UV	2m
SYSENTER	OF 34							C
SYSEXIT	OF 35							C
TEST ac,im	A8w im	4	4	3	2	1	1 UV	1m
TEST r,im	F6w /4 im	5	4	3	2	1	1 UV	Im
TEST m,im	F6w /4 im	11+ea	10	6	5	2	2 UV	2m
TEST r,r	84w /r	3	3	2	2	1	1 UV	1m
TEST m,r	84w /r	13+ea	10	6	5	2	2 UV	2m
TEST r,m	84w /r	13+ea	10	6	5	2	2 UV	2m
UD2	OF OB							C
VERR r16	OF 00 /4			14	10	11	7 NP	C
VERR m16	OF 00 /4			16	11	11	7 NP	C
VERW r16	OF 00 /4			14	15	11	7 NP	C
VERW m16	OF 00 /4			16	16	11	7 NP	C
WAIT	9B	4	6	3	6	1..3	1..3 NP	2m

Таблица 27. Команды (окончание)

Команда	Код	8087	80186	80286 80287	80386 80387	80486	P5	P6
WBINVD	OF 09					5	2000+ NP	C
WRMSR	OF 30						30..45 NP	C
XADD r,r	0F COw /r					3	3 NP	4m
XADD m,r	OF COw /r					4	4 NP	C
XCHG ac,r / r,ac	90r	3	3	3	3	3	2 NP	3m
XCHG r,r	86w /r	4	4	3	3	3	3 NP	3m
XCHG r,m / m,r	86w /r	25+ea	17	5	5	5	3 NP	C
XLAT	D7	11	11	5	5	4	4 NP	2m
XOR ac,im	34w im	4	4	3	2	1	1 UV	1m
XOR r,im	80sw /6 im	4	4	3	2	1	1 UV	1m
XOR m,im	80sw /6 im	23+ea	16	7	7	3	3 UV	4m
XOR r,r	30dw /r	3	3	2	2	1	1 UV	1m
XOR m,r	30dw /r	24+ea	10	7	7	3	3 UV	4m
XOR r,m	30dw /r	13+ea	10	7	6	2	2 UV	2m

Таблица 28. Коды команд расширения SSE

Команда	Код
MOVAPS xmm,xmm/m128	OF 28 /r
MOVAPS xmm/m128,xmm	OF 29 /r
MOVUPS xmm,xmm/m128	0F 10 /r
MOVUPS xmm,xmm/m128	OF 11 /r
MOVLPS xmm,m64	0F 12 /r
MOVLPS m64.xmm	OF 13 /r
MOVHLPS xmm,xmm	0F 12 /r
MOVHPS xmm,m64	OF 16 /r
MOVLHPS xmm,xmm	0F 16 /r
MOVHPS m64,xmm	0F 17 /r
MOVSS xmm,xmm/m32	F3 0F 10 /r
MOVSS xmm/m32,xmm	F3 OF 11 /r
ADDPS xmm,xmm/m128	OF 58 /r
ADDSS xmm,xmm/m32	F3 OF 58 /r
SUBPS xmm,xmm/m128	OF 5C /r
SUBSS xmm,xmm/m128	F3 OF 5C /r
MULPS xmm,xmm/m128	OF 59 /r
MULSS xmm,xmm/m32	F3 OF 59 /r
DIVPS xmm,xmm/m128	0F 5E /r
DIVSS xmm,xmm/m32	F3 OF 5E /r

Таблица 28. Коды команд расширения SSE (окончание)

Команда	Код
RSQRTPS xmm,xmm/m128	OF 52 /r
RSQRTSS xmm,xmm/m32	F3 OF 52 /r
SQRTPS xmm,xmm/m128	OF 51 /r
SQRTSS xmm,xmm/m32	F3 OF 51 /r
RCPPS xmm,xmm/m128	OF 53 /r
RCPSS xmm,xmm/m32	F3 OF 53 /r
MAXPS xmm,xmm/m128	OF 5F /r
MAXSS xmm,xmm/m32	F3 OF 5F /r
MAXPS xmm,xmm/m128	OF 5F /r
MAXSS xmm,xmm/m128	F3 OF 5F /r
MINPS xmm,xmm/m128	OF 5D /r
MINSS xmm,xmm/m128	F3 OF 5D /r
CMPPS xmm,xmm/m128,i8	OF C2 /r i
CMPSS xmm,xmm/m128,i8	F3 OF C2 /r i
COMISS xmm,xmm/m32	OF 2F /r
UCOMISS xmm,xmm/m32	OF 2E /r
CVTPI2PS xmm,mm/m64	OF 2A /r
CVTSI2SS xmm,r/m32	F3 OF 2A /r
CVTPS2PI mm,xmm/m64	OF 2D /r
CVTSS2SI r32,xmm/m32	F3 OF 2D /r
CVTTPS2PI mm,xmm/m64	OF 2C /r
CVTTSS2SI r32,xmm/m32	F3 OF 2C /r
ANDPS xmm,xmm/m128	OF 54 /r
ANDNPS xmm,xmm/m128	OF 55 /r
ORPS xmm,xmm/m128	OF 56 /r

Используемые сокращения

ABI	Application Binary Interface	Интерфейс для приложений на низком уровне
AGI	Address Generation Interlock	Задержка для генерации адреса
AMIS	Alternative Multiplex Interrupt Specification	Спецификация альтернативного мультиплексорного прерывания
API	Application Program Interface	Интерфейс между приложением и программой
ASCII	American Standard Code for Information Interchange	Американский стандартный код для обмена информацией
AT&T	American Telephone and Telegraph	Американский телефон и телеграф (компания, которой принадлежала торговая марка UNIX)
BCD	Binary Coded Decimal	Двоично-десятичный формат
BIOS	Basic Input/Output System	Основная система ввода-вывода
BIT	Binary Digit	Двоичная цифра
BPB	BIOS Parameter Block	Блок параметров BIOS (для блочных устройств)
BRM	Big Real Mode	Большой реальный режим (то же, что и нереальный режим)
BSD	Berkeley System Distribution	Один из основных видов UNIX-систем
BSS	Block, Started by Symbol	Участок программы, содержащий неинициализированные данные
CMOS	Complementary Metal Oxide Semiconductor	Комплементарные металлооксидные пары
COFF	Common Object File Format	Общий формат объектных файлов
CPL	Current Privilege Level	Текущий уровень привилегий
CRT	Cathode Ray Tube	Электронно-лучевая трубка
DAC	Digital to Analog Converter	Цифро-аналоговый преобразователь
DDK	Drivers Development Kit	Набор для создания драйверов
DLL	Dynamically Linked Library	Динамическая библиотека
DMA	Direct Memory Access	Прямой доступ к памяти
DOS	Disk Operating System	Дисковая операционная система
DPL	Descriptor Privilege Level	Уровень привилегий дескриптора

DPMI	DOS Protected Mode Interface	Интерфейс для защищенного режима в DOS
DSP	Digital Signal Processor	Процессор для оцифрованного звука в звуковых картах
DTA	Disk Transfer Area	Область передачи дисковых данных (в DOS)
ELF	Executable and Linking Format	Формат исполняемых и компонуемых файлов
EMS	Expanded Memory Specification	Спецификация доступа к дополнительной памяти
EPB	Execution Program Block	Блок информации об исполняемой программе
FAT	File Allocation Table	Таблица распределения файлов
FCR	FIFO Control Register	Регистр управления FIFO
FIFO	First In First Out	Первый вошел - первый вышел (очередь)
FM	Frequency Modulation	Частотный синтез
FPU	Floating Point Unit	Блок для работы с числами с плавающей запятой
GDT	Global Descriptor Table	Глобальная таблица дескрипторов
HCI	Human Computer Interface	Интерфейс между пользователем и программой
HMA	High Memory Area	Верхняя область памяти (64 Кб после первого мегабайта)
IBM	International Business Machines	Название компании
ICW	Initialization Control Word	Управляющее слово инициализации
IDE	Integrated Drive Electronics	Один из интерфейсов для жестких дисков
IDT	Interrupt Descriptor Table	Таблица дескрипторов обработчиков прерываний
IER	Interrupt Enable Register	Регистр разрешения прерываний
IOCTL	Input/Output Control	Управление вводом-выводом
IOPL	Input/Output Privilege Level	Уровень привилегий ввода-вывода
IRQ	Interrupt Request	Запрос на прерывание (от внешнего устройства)
ISP	Interrupt Sharing Protocol	Протокол разделения прерываний
LCR	Line Control Register	Регистр управления линией
LDT	Local Descriptor Table	Таблица локальных дескрипторов
LE	Linear Executable	Линейный исполняемый формат
LFN	Long File Name	Длинное имя файла
LSR	Line Status Register	Регистр состояния линии

MASM	Macro Assembler	Ассемблер компании Microsoft
MCR	Modem Control Register	Регистр управления модемом
MMX	Multimedia Extension	Расширение для работы с мультимедийными приложениями
MSR	Modem State Register	Регистр состояния модема
MSR	Machine Specific Register	Машинно-специфичный регистр
NE	New Executable	Новый исполняемый формат
NPX	Numerical Processor Extension	Расширение для работы с числами с плавающей запятой
OCW	Operation Control Word	Управляющее слово (для контроллера прерываний)
PE	Portable Executable	Переносимый исполняемый формат
POST	Power On Self Test	Самотестирование при включении
PSP	Program Segment Prefix	Префикс программного сегмента
RBR	Reciever Buffer Register	Регистр буфера приемника
RFC	Request For Comments	Запрос для комментария (форма публикации документов в Internet, включая стандарты)
RFM	Real Flat Mode	Реальный flat-режим (то же, что и нереальный режим)
RTC	Real Time Clock	Часы реального времени
RPL	Requestor Privilege Level	Запрашиваемый уровень привилегий
RPN	Reverse Polish Notation	Обратная польская запись (для арифметических выражений)
SCSI	Small Computer System Interface	Один из интерфейсов для жестких дисков
SUN	Stanford University Networks	Названиекомпании
SVGA	SuperVGA	Любой видеоадаптер, способный на режимы, большие 13h
TASM	Turbo Assembler	Ассемблер компании Borland
THR	Transmitter Holding Register	Регистр хранения передатчика
TSR	Terminate and Stay Resident	Завершиться и остаться резидентным
TSS	Task State Segment	Сегмент состояния задачи
UMB	Upper Memory Block	Блок верхней памяти (между границами 640 Кб и 1 Мб)
VBE	VESA BIOS Extension	СпецификацияVESA для расширения BIOS
VCPI	Virtual Control Program Interface	Один из интерфейсов к защищенному режиму для DOS
VESA	Video Electronics Standard Association	Ассоциация по стандартизации видео в электронике

VGA	Video Graphics Array	Основной тип видеоадаптеров
VxD	Virtual X Device	Виртуальное устройство X (общее название виртуальных драйверов в Windows 95)
WASM	Watcom Assembler	Ассемблер компании Watcom
XMS	Extended Memory Specification	Спецификация доступа к расширенной памяти

Глоссарий

А

Активационная запись (activation record) - область стека, заполняемая при вызове процедуры.

Ассемблер (assembly language) - язык программирования низкого уровня.

Ассемблер (assembler)— компилятор с языка ассемблера.

Б

Байт (byte) - тип данных, имеющий размер 8 бит; минимальная адресуемая единица памяти.

Бит (bit) - минимальная единица измерения информации.

В

Всплывающая программа (popup program) - резидентная программа, активизирующаяся по нажатию определенной «горячей» клавиши.

Г

Горячая клавиша (hotkey) - клавиша или комбинация клавиш, используемая не для ввода символов, а для вызова программ и подобных необычных действий.

Д

Двойное слово (double word) - тип данных, имеющий размер 32 бита.

Дескриптор (descriptor) - восьмибайтная структура, хранящаяся в одной из таблиц GDT, LDT или IDT и описывающая сегмент или шлюз.

Директива (directive) - команда ассемблеру, которая не соответствует командам процессора.

Драйвер (driver) - служебная программа, выполняющая функции посредника между операционной системой и внешним устройством.

З

Задача (task) - программа, модуль или другой участок кода программы, который можно запустить, выполнять, отложить и завершить.

Защищенный режим (protected mode) - режим процессора, в котором действуют механизмы защиты, сегментная адресация с дескрипторами и селекторами и страничная адресация.

И

Идентификатор (handle или identifier) - число (если handle) или переменная другого типа, используемая для идентификации того или иного ресурса.

Исключение (exception) - событие, при котором выполнение программы прекращается и управление передается обработчику исключения.

K

Код (code) - исполняемая часть программы (обычная программа состоит из кода, данных и стека).

Компилятор (compiler) - программа, преобразующая текст, написанный на понятном человеку языке программирования, в исполняемый файл.

Конвейер (pipe) - последовательность блоков процессора, которая задействуется при выполнении команды.

Конвенция (convention) - договоренность о передаче параметров между процедурами.

Конечный автомат (finite state machine) - программа, которая может переключаться между различными состояниями и выполнять в разных состояниях разные действия.

Кэш (cache) - быстрая память, использующаяся для буферизации обращений к основной памяти.

Л

Лимит (limit) - поле дескриптора (равно размеру сегмента минус 1).

Линейный адрес (linear address) - адрес, получаемый сложением смещения и базы сегмента.

Ловушка (trap) - исключение, происходящее после вызвавшей его команды.

M

Метка (label) - идентификатор, связанный с адресом в программе.

H

Нить (thread) - процесс, данные и код которого совпадают с данными и кодом других процессов.

Нереальный режим (unreal mode) - реальный режим с границами сегментов по 4 Гб.

O

Операнд (operand) - параметр, передаваемый команде процессора.

Описатель носителя (media descriptor) - байт, используемый DOS для идентификации типа носителя (обычно не используется).

Останов (abort) — исключение, происходящее асинхронно.

Отложенное вычисление (lazy evaluation) - вычисление, которое выполняется, только если реально требуется его результат.

Очередь предвыборки (prefetch queue) - буфер, из которого команды передаются на расшифровку и выполнение.

Ошибка (fault) - исключение, происходящее перед вызвавшей его командой.

П

Пиксел (pixel) - минимальный элемент растрового изображения.

Повторная входимость (reentrancy) - возможность запуска процедуры из обработчика прерывания, прервавшего выполнение этой же процедуры.

Подчиненный сегмент (conforming segment) — сегмент, на который можно передавать управление программам с более низким уровнем привилегий.

Прерывание (interrupt) - сигнал от внешнего устройства, приводящий к прерыванию выполнения текущей программы и передаче управления специальной программе-обработчику (см. ловушка),

Р

Разворачивание циклов (loop unrolling) - превращение циклов, выполняющихся известное число раз, в линейный участок кода.

Реальный режим (real mode) - режим, в котором процессор ведет себя идентично 8086 - адресация не выше одного мегабайта памяти, размер всех сегментов ограничен и равен 64 Кб, только 16-битный режим.

Резидентная программа (resident program) - программа, остающаяся в памяти после возврата управления в DOS.

С

Сегмент (segment) - элемент сегментной адресации в памяти или участок программы для DOS/Windows.

Секция (section) - участок программы для UNIX.

Селектор (selector) - число, хранящееся в сегментном регистре.

Скан-код (scan-code) - любой код, посылаемый клавиатурой.

Слово (word) - тип данных, имеющий размер 16 бит.

Смещение (offset) - относительный адрес, отсчитываемый от начала сегмента.

Стековый кадр (stack frame) - область стека, занимаемая параметрами процедуры, активационнои записью и локальными переменными или только локальными переменными.

Страничная адресация (pagination) - механизм адресации, в котором линейное адресное пространство разделяется на страницы, которые могут располагаться Б разных областях памяти или вообще отсутствовать.

Т

Таблица переходов (jumptable) - массив адресов процедур для косвенного перехода на процедуру *с* известным номером.

Ш

Шлюз (gate) - структура данных, позволяющая осуществлять передачу управления между разными уровнями привилегий в защищенном режиме.

Алфавитный указатель

Д

З

И

К