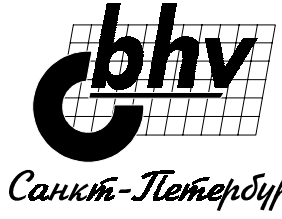
Павел Соколенко

Программирование SVGA-графики для IBM PC

- Особенности режимов SVGA
- Программирование рисунков и цветов
- Спецэффекты

Павел Соколенко

Программирование SVGA-ГРАФИКИ для IBM PC



Дюссельдорф ◆ Киев ◆ Москва ◆ Санкт-Петербург

УДК 681.3.06

В книге изложены основы программирования компьютерной графики для IBM PC на языке ассемблера. В ней рассмотрены: особенности основных видеорежимов SVGA, программирование построения рисунков и палитры цветов, работа с курсором и мышью, вывод текстовых сообщений и получение спецэффектов. Излагаемый материал иллюстрируется многочисленными примерами. В приложениях вы найдете описание графического стандарта BMP, работы с оперативной памятью компьютера, а также техники составления подпрограмм для алгоритмических языков высокого уровня.

Книга предназначена для читателей, интересующихся программированием компьютерной графики, и может быть рекомендована как начинающим, так и опытным программистам.

Для программистов

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Наталья Таркова*
Редактор	*Михаил Кокорев*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Соколенко П. Т.
Программирование SVGA-графики для IBM PC. — СПб.: БХВ-Петербург, 2001. — 432 с.

ISBN 5-94157-127-5

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 06.09.01.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 34,83.
Тираж 3000 экз. Заказ
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар, № 77.99.1.953.П.950.3.99
от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН.
199034, Санкт-Петербург, 9-я линия, 12.

Содержание

Предисловие **7**

Глава 1. Видеосистемы и стандарт VESA **9**

1.1. Видеосистемы и их стандартизация 10
1.1.1. Мониторы 10
1.1.2. Видеокарты и стандарты 11
1.1.3. Акселераторы 16
1.2. Общая характеристика стандарта VESA 21
1.2.1. Стандартизация видеорежимов 21
1.2.2. Информационные функции VBE 27
1.2.3. Основные функции VBE 1.2 32
1.2.4. Новые возможности VBE 2.0 38

Глава 2. Особенности работы в режимах VESA **43**

2.1. Проверка поддержки видеорежима 44
2.2. Обработка информации о режиме 50
2.3. Процедуры для работы с одним окном видеопамяти 52
2.4. Работа с двумя окнами видеопамяти 57
2.5. Страничная организация видеопамяти 60
2.6. Часто используемые в примерах имена 64
2.7. Раздел для начинающих 69

Глава 3. Видеорежимы packed pixel graphics **77**

3.1. Работа с отдельными точками 78
3.1.1. Команды для манипуляции с точками 78
3.1.2. Окна видеопамяти 82
3.1.3. Точки и их адреса 86
3.2. Построение геометрических фигур 90
3.2.1. Прямые линии 90
3.2.2. Прямоугольники 98
3.3. Построение рисунков 103
3.3.1. Варианты построения строк 103
3.3.2. Воспроизведение не сжатых рисунков 111
3.3.3. Воспроизведение сжатых рисунков 117
3.3.4. Заключительные замечания 125

Глава 4. Цвет на экране 129

4.1. Как получается цвет точки 129
4.2. Исходная цветовая палитра 132
4.3. Функции BIOS 137
4.4. Простая установка палитры 142
4.5. Манипуляции с палитрой цветов 148

Глава 5. Работа с текстом 157

5.1. Текстовые режимы 157
5.1.1. Русский текст на экране 158
5.1.2. Общая характеристика процесса вывода текста 161
5.1.3. Вывод текста с использованием поддержки DOS и BIOS 167
5.1.4. Непосредственная работа с видеобуфером 172
5.2. Графические режимы 178
5.2.1. Таблицы символов 178
5.2.2. Программный знакогенератор 181
5.2.3. Вывод информационных строк 187
5.2.4. Текстовый курсор в графическом режиме 193
5.2.5. Ввод символов с клавиатуры 201

Глава 6. Курсор и мышь 207

6.1. Построение рисунка курсора 207
6.1.1. Курсоры для Windows 208
6.1.2. Предварительная подготовка рисунка 210
6.1.3. Немаскируемый курсор 214
6.1.4. Маскируемый курсор 218
6.1.5. Замечания к описанным подпрограммам 223
6.2. Подготовка к работе с манипулятором "мышь" 226
6.2.1. Общее описание драйвера мыши 227
6.2.2. Предварительные действия 231
6.3. Работа в режиме опроса драйвера мыши 237
6.3.1. Управляющий алгоритм для режима опроса 237
6.3.2. Формирование кодов событий 241
6.3.3. Управление перемещением курсора 246
6.4. Работа в режиме прерываний 249
6.4.1. Функции драйвера 250
6.4.2. Примеры прерывающих подпрограмм 254

Глава 7. Цвет в коде точки 261

7.1. Кодирование цвета 261
7.1.1. Среднее количество цветов 262
7.1.2. Максимальное цветовое разрешение 263
7.1.3. 24-разрядный код точки 265
7.2. Координаты и адреса точек 269
7.3. Линии, строки и прямоугольные области 275
7.3.1. Подпрограммы для рисования линий 275
7.3.2. Подпрограммы для построения строк 282
7.3.3. Работа с прямоугольными областями 286

7.4. Рисунки, использующие палитру ... 291
7.4.1. Преобразование палитры в форматы Hi-Color ... 292
7.4.2. Преобразование палитры в форматы True Color ... 296
7.4.3. Построение рисунков с использованием палитры ... 298
7.5. Рисунки, не использующие палитру ... 303
7.5.1. Рисунки, подготовленные в стандарте BMP ... 303
7.5.2. Рисунки, подготовленные в стандарте PCX ... 308
7.5.3. Способы сжатия полноцветных рисунков ... 313
7.6. Наложение рисунков и спецэффекты ... 319

Приложение А. Рисунки в файлах BMP ... 331

А.1. Общая характеристика стандарта ... 332
А.1.1. Заголовок файла для Windows ... 332
А.1.2. Заголовок файла для OS/2 ... 333
А.1.3. Образ рисунка в файле ... 334
А.2. Общая схема обработки заголовка файла ... 336
А.2.1. Возможные отклонения от стандарта ... 337
А.2.2. Ввод спецификации и открытие файла ... 337
А.2.3. Чтение заголовка файла и палитры ... 340
А.2.4. Анализ основных полей заголовка ... 341
А.2.5. Манипуляции с палитрой ... 345
А.3. Построение рисунков, использующих палитру ... 350
А.3.1. Построение рисунка сверху вниз ... 350
А.3.2. Построение рисунка снизу вверх ... 353
А.3.3. Универсальная процедура построения рисунка ... 355

Приложение Б. Оперативная память ... 361

Б.1. Обычная память (Conventional Memory) ... 361
Б.1.1. Сегменты оперативной памяти ... 362
Б.1.2. Сегментирование текстов программ ... 364
Б.1.3. Динамическое управление памятью ... 368
Б.1.4. Использование функций DOS ... 372
Б.2. Расширенная память (Expanded Memory) ... 376
Б.2.1. Спецификация расширенной памяти ... 376
Б.2.2. Использование функций драйвера ... 380
Б.2.3. Работа с расширенной памятью ... 383
Б.3. Расширенная память (Extended Memory) ... 388

Приложение В. Оформление подпрограмм ... 393

В.1. Классификация подпрограмм ... 393
В.2. Оформление программных модулей ... 397
В.3. Параметры в стеке ... 403
В.4. Работа процедур со стеком ... 409
В.5. Учет особенностей компилятора ... 414

Список литературы ... 423

Предметный указатель ... 425

Предисловие

Основы компьютерной графики были заложены еще на больших ЭВМ, задолго до появления персональных компьютеров. Ее первые практические применения были связаны с решением задач из области автоматизации проектирования архитектурных и инженерно-технических сооружений.

Массовое распространение и непрерывное совершенствование технических характеристик персональных компьютеров и периферийного оборудования способствовало расширению круга задач, при решении которых используется графика. В свою очередь, развитие и усложнение графики стимулирует создание все более совершенного компьютерного видеооборудования. Кроме того, непрерывно расширяется круг специалистов, вовлеченных в программирование и использование графических приложений. Поэтому литература, посвященная различным аспектам видеографики, пользуется постоянным спросом и нуждается в периодическом обновлении. Однако большинство публикуемых книг являются руководствами для пользователей, а не для программистов, поскольку содержат описание техники работы с различными графическими редакторами. Автор решил попытаться восполнить этот пробел и написать руководство, содержащее описание способов программирования базовых элементов компьютерной графики и получения спецэффектов.

В свое время на русский язык было переведено и опубликовано несколько обстоятельных руководств по программированию видеосистем для IBM PC. Технический прогресс беспощаден и они устарели вместе с описанными видеосистемами. Предлагаемая книга продолжает тему программирования видеосистем, но уже современного образца, с использованием стандарта VBE, разработанного ассоциацией VESA. Этот стандарт создавался специально для того, чтобы программирование графических объектов не зависело от особенностей видеокарт, выпускаемых различными фирмами.

В соответствии с темой книги в качестве языка программирования выбран ассемблер для IBM PC. Можно до хрипоты спорить о достоинствах и недос-

татках ассемблера, но всегда остается класс задач, которые имеет смысл программировать только на этом языке, и всегда остается категория программистов, которые обязаны в совершенстве владеть ассемблером. В первую очередь именно им адресована данная книга. Вместе с тем, книга составлена так, что она может быть использована как практическое руководство для программистов, начинающих изучать ассемблер или желающих углубить свои знания о нем.

При подборе фактического материала автор стремился к тому, чтобы читатель получил достаточно полное представление об основных режимах SVGA, о том, как устроены графические задачи и какие вопросы приходится решать при их разработке. Насколько это ему удалось — судить вам, уважаемые читатели.

ГЛАВА 1

Видеосистемы и стандарт VESA

Персональный компьютер (далее ПК или PC) не был бы таковым при отсутствии внешних устройств. К ним относятся различные клавиатуры, "мыши", джойстики, принтеры, сканеры, модемы, звуковые карты, накопители на гибких, жестких, оптических и прочих дисках и, конечно же, мониторы. Пожалуй, наиболее важным из всех внешних устройств является оперативная память, поскольку без нее процессор просто не работоспособен. Вообще, внешним является любое устройство, не входящее в состав процессора (точнее микропроцессора).

Процессор не может непосредственно управлять работой внешнего устройства. Для этого нужен посредник — контроллер, который участвует в обмене данными между процессором и устройством и выполняет специфические действия, зависящие от особенностей устройства. Обычно контроллер обслуживает одно устройство. Исключением является контроллер ввода-вывода, обслуживающий все дисководы, а также порты параллельного и последовательного интерфейсов, к которым подключаются принтеры, мыши, джойстики и некоторые другие устройства.

Контроллеры могут располагаться на основной (системной, материнской) плате ПК, либо на отдельных платах (картах), вставляемых в разъемы основной платы. В отдельных случаях на карте может находиться и само внешнее устройство, например, внутренний модем.

Нас будут интересовать видеоконтроллеры, к которым подключаются мониторы. Большинство из них выполнено в виде отдельной платы, но в последние годы наметилась тенденция выпуска материнских плат с расположенными на ней ("интегрированными") видеоконтроллерами. Преимущество отдельных плат в том, что их всегда можно заменить другими с лучшими параметрами.

Для поддержки работы с любым контроллером требуется специальное программное обеспечение. Обычно оно записано на гибких или лазерных дис-

ках, прилагаемых к контроллеру, или входит в комплект операционной системы, например Windows 9X (95, 98, ME) и 2000. В любом случае при вводе нового устройства в эксплуатацию производится установка соответствующего программного обеспечения.

Данная глава содержит общий обзор современных компьютерных видеосистем и способов их программирования. Центральное место в ней занимает описание стандарта VESA, который оказал существенное влияние на развитие компьютерной графики.

1.1. Видеосистемы и их стандартизация

Компьютерный рынок динамичен по своей природе, он постоянно предлагает новое, все более совершенное видеооборудование для ПК. Пользователи, напротив, склонны соразмерять свои потребности и реальные финансовые возможности. Поэтому в эксплуатации находится множество видеосистем, изготовленных разными фирмами и в разное время. Они существенно различаются по конструктивному исполнению, техническим характеристикам и, конечно же, по особенностям программирования. На их примере можно проследить всю историю развития технических средств персональных компьютеров от первых IBM PC XT до новейших моделей на базе процессоров семейства Pentium.

1.1.1. Мониторы

Основными элементами компьютерных видеосистем являются мониторы и видеокарты. Кроме того, к ним относятся графические ускорители и платы для работы с телевизионным изображением, которые используются в зависимости от особенностей решаемых задач.

Монитор или дисплей является одним из важнейших узлов современного персонального компьютера, предназначенным для визуализации (т. е. для представления в виде, доступном для человеческого глаза) выводимой информации. За время существования ПК было выпущено достаточно много мониторов различных типов. По степени улучшения технических характеристик их можно расположить в такой последовательности: Color Graphics Adapter (CGA), Enhanced Graphics Adapter (EGA), Video Graphics Array (VGA), Super Video Graphics Array (SVGA). Этот перечень не претендует на полноту, но каждая из названных моделей получила, в свое время, наибольшее распространение. Мониторы CGA, EGA в настоящее время безнадежно устарели и не выпускаются промышленностью, однако некоторые из них еще находятся в эксплуатации.

Для получения изображения на экране монитора требуется выполнение специальных действий, поэтому неотъемлемой частью монитора является

встроенное в него устройство управления. Это еще не контроллер, речь идет о совершенно других функциях, поэтому в англоязычной литературе для названия таких устройств используется термин `unit`. Именно благодаря встроенному устройству управления все современные мониторы (VGA и SVGA) подключаются к стандартному разъему компьютера и поддерживают стандартный аппаратный интерфейс, т. е. определенную последовательность управляющих сигналов, их полярность, уровень и форму. Поэтому мониторы выпускаются и продаются как отдельные функциональные устройства без контроллера (так же, как, например, принтеры).

Для получения изображения в современных настольных мониторах используется электронно-лучевая трубка (кинескоп). Модели таких мониторов различаются по размеру экрана, разрешающей способности, используемому кинескопу, способам настройки (простые или с цифровым управлением). Эти характеристики определяют качество изображения и стоимость изделия, но никак не влияют на интерфейс с контроллером и способы программирования.

В переносных и портативных компьютерах применяются мониторы, в которых изображение получается на матрице из жидких кристаллов. На сегодняшний день качество такого изображения хуже, чем у мониторов с обычным кинескопом. Однако поиски новых принципов получения изображения на экране идут весьма активно и, возможно, в ближайшем будущем, будут достигнуты большие успехи в этом направлении.

Как любое внешнее устройство монитор не может быть подключен к ПК без специального контроллера, являющегося посредником между ним и центральным процессором (ЦП). Эти контроллеры легко отличить от других плат, поскольку на них расположен специальный разъем для подключения монитора. При описании такого типа контроллеров используют термины "видеоконтроллер", "видеокарта", "видеоплата", реже "видеоадаптер".

1.1.2. Видеокарты и стандарты

Видеокарты воспринимают цифровую информацию, поступающую от ЦП, и вырабатывают сигналы, управляющие работой монитора. Ядром видеокарты является специализированный микропроцессор, выполняющий все необходимые функции. От него зависят такие технические характеристики, как производительность (или быстродействие), предельно допустимый объем памяти, конкретные особенности программирования. Кроме того, на любой видеокарте расположена оперативная память (`RAM`), предназначенная для хранения цифрового образа, выводимого на экран изображения. Наконец, на видеокарте находится постоянная память (`ROM`), содержащая фрагмент базовой системы ввода-вывода (`BIOS`).

Системные шины. Конструктивное исполнение видеокарт соответствует определенным техническим стандартам. Каждая из них способна взаимодейст-

вовать только с конкретным типом системной шины. Внешним признаком этого является форма разъема (гнезда) материнской платы, в которое устанавливается видеокарта. Системная шина расположена на материнской плате и представляет собой совокупность проводящих линий, по которым передаются данные, адреса и управляющие сигналы. От нее зависит такая важная характеристика, как скорость передачи данных, а следовательно, и время, затрачиваемое на построение изображения на экране.

На материнских платах компьютеров, собранных на базе процессоров Intel 80286 и Intel 80386 применялась шина ISA (Industry Standard Architecture), при использовании которой обмен данными между видеокартой и процессором производится словами или байтами. Для процессора Intel 486 была разработана новая системная шина VLB (VESA Local Bus), но ее очень скоро вытеснила шина PCI (Peripheral Component Interconnect). Обе шины позволяли передавать данные двойными словами. Большинство имеющихся в продаже современных видеокарт выполнено в стандарте PCI. С выпуском процессоров Pentium II на системной плате появилась специальная 128-разрядная шина для обмена данными с видеокартой. Она заканчивается гнездом AGP (Accelerated Graphics Port). В настоящее время видеокарты, выполненные в стандарте AGP, преобладают на компьютерном рынке.

Для каждого типа шин выпускались и продолжают выпускаться не только видеокарты, но и платы различного назначения. Поэтому на материнских платах обычно имеются разъемы для установки карт, выполненных в стандартах ISA и PCI. Например, одна из современных материнских плат фирмы Intel (ее тип AL 440 LX) содержит 2 разъема ISA, 4 разъема PCI и 1 разъем AGP. Вполне возможно, что на современном ПК с процессором Pentium вы обнаружите видеокарту, выполненную в стандарте PCI и даже в устаревшем стандарте ISA.

Описанные различия видеокарт учитываются при их программировании лишь в особых случаях, когда требуется максимальная производительность видеосистемы. Большинство прикладных задач в этом не нуждается. Поэтому для нас более важно знать тип монитора, для обслуживания которого предназначена видеокарта, поскольку от этого зависят основные особенности ее программирования и структура графических задач.

Стандарты IBM. Свой первый персональный компьютер американская фирма IBM (International Business Machines) выпустила в 1981 году. В то время основным производителем персональных компьютеров была другая американская фирма DEC (Digital Equipment Corporation). IBM не входила даже в тройку лидеров, но за короткий отрезок времени она стала "законодателем мод" в сфере производства ПК. Этому, в немалой степени, способствовало то, что IBM публиковала подробную информацию о своих новых разработках, и ее могли использовать другие фирмы, занимающиеся производством компьютеров, совместимых с IBM PC, дополнительных плат различного назначения и разработкой программного обеспечения.

Впервые за всю историю существования IBM применила в своем изделии компоненты, изготовленные другими фирмами. В частности, в IBM PC использовался микропроцессор Intel 8086. С тех пор все семейство IBM PC базируется на микропроцессорах фирмы Intel. Кроме них могут применяться совместимые микропроцессоры фирм AMD и Cyrix.

Первая модель ПК выпускалась недолго, на смену ей пришел компьютер второго поколения PC XT, аббревиатура XT расшифровывается как eXtended Technology (расширенная технология). В нем по-прежнему использовался микропроцессор Intel 8086, но пространство оперативной памяти было увеличено до 640 Кбайт. Кроме того, были разработаны новые видеокарты, предназначенные для работы с монитором CGA и позволявшие отобразить 8 цветов. Объем видеопамяти у них достигал 4 Кбайт, а разрешающая способность составляла 320×200 точек.

Следующая модель компьютера была создана на базе микропроцессора Intel 80286, она называлась PC AT, аббревиатура AT расшифровывается как Advanced Technology (прогрессивная технология). При разработке PC AT в качестве стандарта был принят монитор EGA. На видеокартах появился новый разъем для подключения монитора. Количество разных цветов точки возросло до 16-ти, а объем видеопамяти до 64 Кбайт. Такой объем видеопамяти и разрешающая способность монитора позволяли создавать на экране изображение размером 640×350 точек.

К этому времени производство видеокарт и мониторов освоили разные фирмы, и их продукция существенно различалась по техническим характеристикам. Выпускались карты, которые позволяли использовать 64 цвета и имели объем видеопамяти больше, чем 64 Кбайт. Однако для использования таких карт требовалось описание способов их программирования, которое в большинстве случаев недоступно для программистов.

Для стандартов CGA и EGA характерна сложная организация видеопамяти. Простая запись кода точки в видеопамять или чтение кода из нее невозможны, для этого требуется около десятка команд и приходится работать с портами видеокарты.

Стандарт на монитор VGA был опубликован при выпуске новой серии IBM PS (персональные системы) на базе процессора Intel 80386. Эта серия компьютеров не получила широкого распространения. Стандарт был принят, но в стремлении вырваться вперед IBM выпустила недоработанный продукт, чем и не замедлили воспользоваться конкуренты.

Стандарт VGA предусматривал новый трехрядный 15-контактный разъем для подключения монитора. Пожалуй, это наиболее продуманная часть стандарта. В разъеме были оставлены свободные (зарезервированные) контакты для будущих расширений. Поэтому его форма не изменилась до настоящего времени. При подключении современных SVGA-мониторов используются некоторые из ранее зарезервированных контактов разъема. Следует отметить,

что такой стандарт разъема распространяется только на видеокарты, предназначенные для семейства IBM PC.

Стандарт VGA был шагом вперед по количеству цветов, которое возросло с 16 до 256. Объем видеопамяти увеличился до 256 Кбайт, и упростилась ее организация. В отличие от стандартов CGA и EGA, запись и чтение кода точки теперь производились одной командой, как при работе с обычной (оперативной) памятью компьютера. Однако стандарт VGA имел следующий существенный недостаток. Видеопамять, как и обычная память, делится на сегменты размером по 64 Кбайт. Стандарт VGA не предусматривал механизм переключения сегментов, поэтому на экране можно было отобразить содержимое только одного из них. Соответственно размер максимально возможного изображения составлял 320×200 точек (320×200 = 64 000, что немного меньше, чем 64 Кбайт).

Стандарты IBM и BIOS. Для программирования конкретной видеокарты надо знать назначение ее внутренних регистров, их размерность (байты, слова и пр.), способ записи или чтения данных и расположение величин в разрядах регистров. Первый стандарт IBM регламентировал назначение, состав и способы работы с внутренними регистрами, что исключало несовместимость видеокарт. Но скоро стало очевидно, что это плохой способ решения проблемы совместимости и в стандартах EGA и VGA указанные требования распространялись только на основную часть регистров.

Для решения проблемы совместимости были стандартизированы функции BIOS. IBM выпустила описание базового набора, содержащее перечень основных функций, способ их вызова, назначение и размещение входных и выходных параметров. Так появилась группа функций BIOS с названием "Video Services". Образующие ее подпрограммы и данные не входят в основную часть BIOS, они хранятся в специальной микросхеме, расположенной на видеокарте. Поэтому, устанавливая новую видеокарту, вы одновременно устанавливаете новую реализацию функций указанной группы. У современных моделей видеокарт эта группа может занимать полный сегмент памяти, т. е. 64 Кбайт. Это свидетельствует о сложности и разнообразии выполняемых действий и о большом объеме используемых при этом данных.

Именно благодаря наличию функций "Video Services" вы можете быть уверены в том, что после смены видеокарты ваш ПК сохранит свою работоспособность. Программисты получили существенное упрощение структуры прикладных задач и их независимость от моделей видеокарт. А разработчики получили возможность изменять программную реализацию функций BIOS для учета конкретных особенностей видеокарты.

Перед выводом на экран монитора текста или графических изображений должен быть установлен соответствующий видеорежим. В частности, при первоначальной загрузке ПК BIOS устанавливает текстовый режим работы, при котором на экране можно расположить 25 строк, каждая из которых

содержит не более чем 80 символов. DOS обычно не изменяет этот режим, а прикладные задачи могут выполняться в текстовых или в графических режимах.

В группу "Video Services" обязательно входит функция, выполняющая установку видеорежима. При ее вызове указывается код видеорежима, а данные, необходимые для его установки, хранятся в области BIOS. IBM ввела стандартные значения кодов для 20-ти видеорежимов, значения которых изменяются от 0 до 13h (буква h — признак шестнадцатеричного числа). Разработчики видеокарт могут вводить новые режимы по своему усмотрению, чем они обычно и пользуются.

Стандарты были не всегда. После неудачи со стандартом VGA IBM прекратила работы по стандартизации видеооборудования. А поскольку никто этим не занимался, то наступил период "разброда и шатаний". Каждая фирма проектировала платы по своему усмотрению, не заботясь о каком-либо общем стандарте, кроме собственного. В результате было выпущено много хороших, но не совместимых друг с другом видеокарт, поддерживающих видеорежимы с более высоким, по сравнению с VGA, геометрическим и цветовым разрешением. Коды и характеристики режимов существенно различались и программы, рассчитанные на работу с одной видеокартой, не могли работать с другими или, в лучшем случае, требовали дополнительной настройки. Разумеется, что программисты нашли выход и в системных библиотеках появились модули для определения типа установленной на компьютере видеокарты и настройки программы на ее параметры, но это были полумеры, требовалось радикальное решение.

При этом следует отметить, что отсутствие стандартов имело и свою положительную сторону. Именно в это время разработчиками видеооборудования был накоплен практический опыт использования различных видеорежимов. Трудно себе представить специалистов, которые могли бы предусмотреть все возможные случаи, не опираясь на существующий опыт. Намного проще обобщить достигнутые результаты, оставить главное и отбросить ненужное.

Необходимость стандартизации понимали не только программисты, но и производители видеооборудования. Благодаря объединению их усилий и была создана специализированная ассоциация VESA, которая до настоящего времени занимается вопросами стандартизации видеооборудования (не только для IBM PC). Все ведущие производители придерживаются этих стандартов и проблема несовместимости мониторов или видеокарт в наше время не столь актуальна, но технический прогресс вынуждает, время от времени, вновь возвращаться к проблеме стандартизации, уже на более высоком уровне работы с графикой.

Подведем итог сказанному в данном разделе. В современных видеокартах используется различная элементная база, поэтому они могут существенно

различаться по своим техническим характеристикам. Но при использовании функций BIOS все они без исключения совместимы на программном уровне с видеорежимами VGA IBM и VESA. Сказанное распространяется не только на обычные видеокарты, но и на акселераторы.

1.1.3. Акселераторы

В истории вычислительной техники развитие программных и аппаратных средств тесно переплетено друг с другом. Новые технические возможности позволяют программистам сделать очередной шаг вперед и при этом у них часто появляются новые требования к аппаратуре. Основное влияние на расширение функций видеокарт оказало стремление получить на экране монитора объемное движущееся изображение, построенное с учетом перспективы и распределения света и тени. Пионерами этого направления были разработчики компьютерных игр.

Трехмерная графика. Как известно, на холсте или листе бумаги можно нарисовать только плоское изображение, а для придания ему эффекта объемности применяются специальные приемы рисования. То, что мы видим в результате, является оптической иллюзией (обманом зрения), основанной на нашем жизненном опыте восприятия окружающего мира.

Для получения изображения куба его грани надо расположить под определенными углами. У шара граней нет, поэтому для придания кругу эффекта объемности используется распределение света и тени, создаваемое с помощью штриховки или раскрашивания. Для получения эффекта расположения в пространстве нескольких объектов их размеры уменьшаются по мере удаления от точки наблюдения.

Перечисленные приемы основаны не только на нашем субъективном восприятии окружающего мира, но и на вполне объективных законах оптики. Существует возможность формального описания способов преобразования трехмерных объектов в двухмерные и программной реализации алгоритмов вычислений. Такие алгоритмы не учитывают субъективный фактор, но это второстепенный вопрос.

В отличие от листа бумаги экран монитора является плоской дискретной поверхностью, поэтому компьютерная графика имеет дело с дискретными объектами. Для их аналитического описания нужны специальные методы аппроксимации. При описании плоских (двухмерных) объектов обычно применяются методы линейной и векторной графики, а при описании трехмерных объектов — полигональной графики (слово polygon переводится как многоугольник). Чаще всего в качестве многоугольников используются треугольники.

В разработку методов трехмерной графики основной вклад внесла Reality Lab 3D, подразделение ныне не существующей компании Rendermorphis.

Результаты ее работы легли в основу графической библиотеки Direct3D, продукт фирмы Microsoft. Они же были использованы при создании языка VRML, предназначенного для описания трехмерных сценариев. Его создала Cosmo Software, а адаптировала российская фирма ParaGraph.

Процесс построения трехмерного изображения можно разделить на два этапа:

- геометрические вычисления;
- визуализация полученных результатов.

Геометрические вычисления сводятся к манипуляциям с векторами и матрицами, в результате которых получается совокупность треугольников, аппроксимирующая поверхность трехмерного объекта. Эти вычисления полностью зависят от конкретных свойств графических объектов, поэтому их выполняет центральный процессор ПК.

Процесс визуализации (rendering) заключается в том, что полученные треугольники отображаются на дискретную плоскость (поверхность экрана). При этом некоторые из них могут превратиться в линии или точки, а часть окажется на невидимой стороне объекта. Треугольники раскрашиваются по заданным образцам (их называют текстурами), а для получения эффекта объемности учитывается распределение уровней освещенности. Кроме того, могут понадобиться вычисления, связанные с изменением размеров всего изображения или отдельных его частей, коррекцией перспективы, созданием эффекта тумана и др.

Все эти действия не столь сложны, как геометрические построения, но их количество огромно, оно во много раз больше, чем количество точек на экране монитора или в рисунке, если он занимает не весь экран. Выполнение визуализации процессором ПК существенно замедляет работу с графикой.

Одним из распространенных приемов ускорения является перенос несложных, но многократно повторяемых вычислений на аппаратный уровень. Для этого создаются специальные процессоры, выполняющие нужные вычисления по микропрограммам, работающим намного быстрее программ аналогичного назначения.

Модели акселераторов. Акселератор (accelerator, ускоритель) является специализированным вычислительным устройством, предназначенным для ускорения процесса построения или преобразования графических изображений. В отличие от обычных видеокарт акселератор получает от процессора не завершенный образ изображения, а более общую и сжатую информацию, на основании которой он, а не процессор, вычисляет точечный образ рисунка и записывает его в видеопамять.

Первые модели акселераторов выпускались в виде отдельных плат, которые обрабатывали данные, поступающие от процессора, и передавали их обычной видеокарте. Но очень скоро вычислительный микропроцессор начали

располагать на плате видеокарты, а затем его и видеоконтроллер объединили в одну большую микросхему (чип). Акселераторы могут предназначаться для ускорения работы с двухмерными или трехмерными графическими объектами. В названии первых присутствует обозначение 2D, а в названии вторых — 3D, где D является первой буквой слова direction — направление.

Акселераторы могут быть рассчитаны на установку в гнездо шины PCI или в специализированный разъем AGP, который появился на материнских платах ПК после выпуска микропроцессора Pentium II. Разъем AGP имеет 128-разрядную шину данных, что существенно ускоряет процесс обмена между центральным процессором и видеокартой.

При вычислениях используется видеопамять, расположенная на плате акселератора, поэтому ее объем всегда больше того, который нужен для работы видеокарты в обычном режиме. В настоящее время в продаже имеются акселераторы AGP с объемом видеопамяти 32 и 64 Мбайт.

Современные модели акселераторов чаще всего собираются на базе специализированного графического процессора, выполненного в виде большой интегральной микросхемы (чипа), которые выпускают более десятка различных фирм. Корпорация Intel выпустила свой чип i740, но пока он не получил широкого распространения. По данным агентства Mercury Research на сентябрь 1998 года, в первую пятерку производителей графических чипов входят: ATI, S3, Cirrus Logis, Silicon Integrated System (SIS) и Trident Microsystems. Кроме основных чипов эти фирмы выпускают комплекты сопутствующих микросхем и полностью завершенную продукцию, т. е. видеокарты и акселераторы. Возглавляющая список ATI Technologies производит 27% всех графических чипов. Поэтому вероятность приобрести акселератор, собранный из ее комплектующих, весьма велика.

Акселераторы существенно различаются по цене, но возможности дешевых моделей ограничены. Обычно у них разрешающая способность не превышает 800×600 точек, а из полноцветных видеорежимов поддерживается только Hi-Color. Однако при использовании акселератора в качестве обычной видеокарты эти ограничения отсутствуют.

Функции акселераторов. Набор выполняемых функций зависит от конкретного назначения акселератора. По личному опыту вы знаете, что при рисовании и черчении приходится иметь дело с различными объектами и с разными способами их изображения. Соответственно, функции акселераторов, предназначенных для ускорения геометрических построений и для работы с художественными изображениями, различаются весьма существенно. Некую комбинацию из этих функций могут поддерживать акселераторы, предназначенные для систем автоматизации проектирования. Специфический набор функций поддерживают акселераторы, используемые для игровых приложений. В этих случаях основными требованиями являются быстрота

смены картинки и возможность создания различных спецэффектов, а точность построения самого изображения не столь существенна.

На сегодняшний день невозможно выделить некий стандартный набор функций, выполняемых акселераторами. У разработчиков отсутствует достаточный практический опыт, поэтому они просто воплощают функции, поддерживаемые графическими библиотеками Direct3D и OpenGL. К ним относятся раскрашивание треугольников по заданным образцам (наложение текстур), альфа смешение (см. раздел 7.6), создание эффекта тумана (см. раздел 7.6), вычисление распределения света и тени по методу Гуро (Gouraud Shading), коррекция перспективы и некоторые другие.

Кроме того, у акселераторов появилась функция иного назначения. Это преобразование телевизионного изображения в компьютерное и обратно. В таких случаях на видеокарте имеется разъем для подключения телевизора или видеомагнитофона. В качестве примера можно привести изделия фирм ATI и S3. Подчеркнем, что речь идет не о приеме телевизионных сигналов — для этого существуют специальные платы, а об аппаратном преобразовании сигналов из телевизионного стандарта (`NTSC`, `PAL` и т. п.) в последовательность кодов точек, записываемых в видеопамять. Такое преобразование является двухсторонним, т. е. коды хранящихся в видеопамяти точек могут преобразовываться в один из телевизионных стандартов. Такие функции расширяют возможности компьютерной обработки телевизионных изображений и делают ее более доступной для пользователей.

Программирование акселераторов. После включения ПК и загрузки операционной системы, неважно какой, акселератор работает как обычная видеокарта. Такой режим необходим для нормальной работы операционных систем и многих прикладных задач. Примером могут служить Windows и ее многочисленные приложения. Программирование акселератора как обычной видеокарты ничем не отличается от того, что описано в данной книге.

Основные (вычислительные) функции акселераторов выполняются в 32-разрядном (защищенном) режиме работы ПК. Речь идет о разрядности адресов, данные могут содержать меньшее или большее количество разрядов. Выполнение прикладных задач в защищенном режиме поддерживают, например, Windows 9X/2000/NT и OS/2. DOS является операционной системой реального (16-разрядного) режима, но существуют так называемые расширители (`DOS extenders`), которые подключаются к прикладной задаче и создают на время ее выполнения вычислительную среду, необходимую для работы в защищенном режиме. Наиболее известными из них являются DOS4GW, DOS32A, PMODE/W.

При выборе операционной системы приходится учитывать тот факт, что в настоящее время не существует стандарта на программирование функций акселераторов, хотя ассоциация VESA предпринимает активные усилия по его разработке. Первый документ VBE/AF Standard 1.0 был выпущен в августе 1996 года. В настоящее время опубликована третья редакция этого доку-

мента, но в ее первых строках подчеркивается, что она содержит черновые предложения и не более того.

Отсутствие стандартов означает, что непосредственно взаимодействующая с акселератором задача не будет переносимой. Она будет выполняться только на тех ПК, на которых установлена соответствующая модель акселератора. Существует довольно много компьютерных игр, созданных для определенных моделей акселераторов, в остальных случаях они либо вообще не работают, либо работают медленно, если вычисления выполняет процессор.

Проблема переносимости частично решается с помощью драйверов, которые продаются вместе с акселератором. Они составлены для определенной операционной системы и рассчитаны на взаимодействие с одной из распространенных графических библиотек.

Среда Windows позволяет создавать любые графические приложения. Разработчикам доступны графические библиотеки Direct3D и OpenGL, которые хорошо документированы и общедоступны. Если установлен соответствующий драйвер, то они используют возможности акселератора, в противном случае требуемые действия выполняются программно, что замедляет процесс выполнения задач, но решает проблему их переносимости.

Пакет Direct3D разработан Microsoft и является одной из частей библиотеки DirectX, входящей в состав Windows 9X, начиная с версии 98, Windows NT и 2000. Он предназначен для ускорения выполнения игровых задач в среде Windows. Первая версия пакета была выпущена в 1996 году.

Библиотека OpenGL была создана в 1993 году фирмой Silicon Graphics для компьютеров совершенно другого класса и для иной операционной системы. В 1995 году совместно с Microsoft она адаптировала ее для IBM PC. С этой библиотекой работает, например, последняя версия компилятора Фортрана для Windows.

В отличие от Direct3D, OpenGL более гибкая и многофункциональная библиотека. Изначально она создавалась для применения трехмерной графики в системах автоматизированного проектирования. Однако она не содержит средств, позволяющих работать непосредственно с видеопамятью в обход интерфейса графических устройств (GDI), который существенно замедляет выполнение прикладных задач. Для этого нужна дополнительная библиотека WinG или DirectDraw, которая является частью библиотеки DirectX.

Еще совсем недавно мысль о том, что можно создать хорошее трехмерное графическое приложение под Windows казалась совершенно нелепой любому программисту, имеющему дело с этой системой. С появлением библиотеки DirectX ситуация изменилась в лучшую сторону. Но, тем не менее, среда Windows остается весьма инерционной, и разработчики компьютерных игр продолжают и, вероятно, еще долго будут продолжать использовать для ускорения процесса игры расширитель DOS4GW и ему подобные.

1.2. Общая характеристика стандарта VESA

Video Electronics Standards Association (ассоциация стандартизации видеоэлектроники), сокращенно VESA, была основана в 1989 году. В августе того же года она опубликовала свой первый стандарт для 16-цветного видеорежима SVGA с разрешением 800×600 точек. С тех пор ассоциация выпустила множество различных стандартов, охватывающих широкий спектр видеооборудования. Одной из ее известных разработок является стандарт на системную шину VLB (VESA Local Bus) для микропроцессора Intel 486. Однако, как уже говорилось, эта шина не прижилась.

Если у вас есть доступ к сети Internet, то подробные сведения об ассоциации VESA и ее продукции можно найти на серверах www.vesa.org и ftp.vesa.org.

Нас будут интересовать стандарты VESA, регламентирующие способы программирования видеокарт. Первый завершенный стандарт появился в октябре 1991 года, он определял полный набор видеорежимов SVGA и дополнительных функций BIOS и назывался VESA BIOS Extension (VBE) version 1.2. Это функции той части BIOS, которая расположена на видеокартах и обслуживает видеосервис. Стандарт объединил предыдущие версии VBE 1.0 и VBE 1.1. Ему соответствуют практически все видеокарты, изготовленные начиная с 1992 года. Более современные видеокарты поддерживают версию VBE 2.0, которая совместима (сверху вниз) с версией VBE 1.2. Поэтому учет рекомендаций VESA при программировании работы с графикой позволяет создавать переносимые задачи, которые будут правильно работать независимо от модели видеокарты, установленной на конкретном компьютере.

1.2.1. Стандартизация видеорежимов

Понятие "видеорежим" является обобщенной характеристикой текущего состояния видеоконтроллера. Основная функция видеоконтроллера состоит в отображении содержимого видеопамяти на экране монитора. Выполнение этой функции зависит от множества величин, хранящихся во внутренних регистрах видеоконтроллера. Значения этих величин определяются при установке видеорежима. Нас интересуют те из них, которые не только влияют на работу видеокарты, но и должны учитываться в прикладных задачах.

Характеристики видеорежимов. Прежде всего, видеорежимы делятся на *текстовые* и *графические*. В зависимости от типа режима прикладная задача записывает в видеопамять или коды символов в стандарте ASCII, или коды отдельных точек графического объекта. При работе в графических режимах видеоконтроллер просто выводит на экран точки, коды которых хранятся в видеопамяти. При работе в текстовых режимах он, по кодам символов,

выбирает их изображения из специальных таблиц, а затем выводит точки изображений на экран.

Другой важной характеристикой является *разрешающая способность*. В зависимости от типа видеорежима она измеряется количеством символов или точек, которое можно разместить по горизонтали и вертикали в пределах рабочей области экрана. Количество точек является основной, а количество символов — производной единицей, т. к. оно зависит от первой величины и от размеров ячейки (знакоместа), отведенной для размещения одного символа.

Точки, расположенные по горизонтали, образуют строку, а по вертикали — столбец (в документации на BIOS используются термины row (ряд) и column (столбец). Количество точек в строке и в столбце не может быть произвольным, оно всегда кратно восьми. Максимально возможное количество точек в строке зависит от разрешающей способности монитора и его геометрических размеров. У современных мониторов минимальное расстояние между центрами смежных точек составляет от 0,28 до 0,26 мм. При размере экрана 14 дюймов по диагонали количество точек в строке не превышает 1024. У 15-дюймовых мониторов оно достигает значения 1280. Однако возможность работы в режимах с высоком разрешением зависит еще и от видеокарты, о чем будет сказано ниже.

Расстояние между соседними точками, расположенными по горизонтали и вертикали, подбирается одинаковым, для того чтобы изображение квадрата на экране выглядело как квадрат, а не как прямоугольник. Обычно количество точек по горизонтали больше, чем по вертикали, но существуют мониторы и с вертикальной ориентацией страницы.

Рабочая область никогда не заполняет всю видимую часть экрана. Во всех видеорежимах ее окружает пространство, которое в документации называется overscan или border (граница, кайма). Поэтому в разных режимах геометрические размеры рабочей области могут не совпадать.

Важной характеристикой видеорежимов является *количество цветов*, которое можно одновременно отобразить на экране. Во всех графических режимах цвет получается в результате совмещения в одной точке экрана трех базовых цветов (красного, зеленого и синего) разной интенсивности. В зависимости от видеорежима коды базовых цветов располагаются либо в специальных регистрах видеокарты, либо в видеопамяти, т. е. непосредственно в коде точки. Первую категорию режимов принято называть packed pixel graphics (упакованная точечная графика), а вторую — direct color (непосредственный цвет). Вторая категория, в свою очередь, делится на режимы Hi-Color и True Color. В любом случае от видеорежима зависят размер кода точки и размеры кодов базовых цветов.

Видеорежимы VESA. Разработчикам стандарта VESA предстояло, в первую очередь, ограничить разнообразие применявшихся на практике видеорежи-

мов, связав с каждым из них конкретный код и набор характеристик. В двух первых версиях стандарта было описано 8 графических режимов `packed pixel graphics` и 5 текстовых режимов высокого разрешения. Графические видеорежимы `direct color` были введены в третьей версии стандарта. Коды режимов VESA и их характеристики перечислены в табл. 1.1.

Таблица 1.1. *Видеорежимы VESA*

Код режима	Количество точек в строке	Количество строк по вертикали	Размер точки в битах	Размер строки в байтах	Количество цветов
VESA редакции 1.0 и 1.1					
100h	640	400	8	640	256
101h	640	480	8	640	256
102h	800	600	4	100	16
103h	800	600	8	800	256
104h	1024	768	4	128	16
105h	1024	768	8	1024	256
106h	1280	1024	4	160	16
107h	1280	1024	8	1280	256
108h	80	60	–	160	16
109h	132	25	–	264	16
10Ah	132	43	–	264	16
10Bh	132	50	–	264	16
10Ch	132	60	–	264	16
VESA редакции 1.2					
10Dh	320	200	15	640	32K
10Eh	320	200	16	640	64K
10Fh	320	200	32/24	1280/960	16M
110h	640	480	15	1280	32K
111h	640	480	16	1280	64K
112h	640	480	32/24	2560/1920	16M
113h	800	600	15	1600	32K
114h	800	600	16	1600	64K

Таблица 1.1 *(окончание)*

Код режима	Количество точек в строке	Количество строк по вертикали	Размер точки в битах	Размер строки в байтах	Количество цветов
VESA редакции 1.2 *(прод.)*					
115h	800	600	32/24	3200/2400	16M
116h	1024	768	15	2048	32K
117h	1024	768	16	2048	64K
118h	1024	768	32/24	4096/3072	16M
119h	1280	1024	15	2560	32K
11Ah	1280	1024	16	2560	64K
11Bh	1280	1024	32/24	5120/3840	16M

В первом столбце табл. 1.1 перечислены коды видеорежимов. Это шестнадцатеричные числа, поэтому в их записи могут встречаться не только цифры, но и латинские буквы от `A` до `F`, а в конце кода обязательно указывается латинская буква `h` (100h = 256). В остальных столбцах таблицы приведены десятичные числа. В последнем столбце, для сокращения записи количества цветов, использованы буквы `K` и `M`. Они обозначают степени числа два, наиболее близкие по значению к тысяче `(K=1024)` и миллиону `(M=1048576)`. Соответственно, количество цветов может быть следующим: `32K = 32768, 64K = 65536, 16M = 16777216`.

В графических режимах размер строки указан в точках, а в текстовых — в виде количества символов. В текстовых режимах ширина символов постоянна и составляет 8 точек, а высота — 8 или 16 точек.

Классификация режимов. Перечисленные в табл. 1.1 видеорежимы делятся на следующие пять групп:

- текстовые режимы;
- 16-цветные режимы (`EGA`);
- `packed pixel graphics` (256 цветов);
- `Hi-Color` (`direct color` 32K или 64K цветов);
- `True Color` (`direct color` 16M цветов).

Этим группам соответствуют 4 разные модели видеопамяти, поскольку разновидности режимов `direct color` используют одну модель. В описании стандарта понятие "модель видеопамяти" четко не объясняется, но речь идет

о том, как видеоконтроллер интерпретирует содержимое байтов видеопамяти. Для программиста важно знать не модель, а способ доступа к видеопамяти и что при этом записывается в ее байты.

Три из четырех моделей допускают непосредственную работу с видеопамятью, т. е. запись и чтение содержимого ее байтов и слов с помощью обычных команд ассемблера. Исключением являются режимы `EGA`, в этом случае для чтения или записи необходима работа с внутренними регистрами видеокарты. Эти режимы морально устарели, кроме того, они хорошо описаны в литературе, поэтому в данной книге не рассматриваются.

Текстовые режимы VESA просто расширяют возможности аналогичных режимов IBM и позволяют использовать стандартные процедуры `BIOS`, предназначенные для работы с текстом. Программирование в текстовых режимах описано в первой части главы 5.

Режимы `packed pixel graphics` отличаются от режима VGA IBM тем, что введено сегментирование видеопамяти, все пространство которой делится на окна размером по 64 Кбайт. Своевременное переключение окон позволяет работать с большим пространством видеопамяти, которое требуется для поддержки видеорежимов с высоким разрешением. Напомним, что в режиме VGA IBM разрешение составляет 320×200 точек (сравните с табл. 1.1).

В режимах `packed pixel graphics` между кодами точки и цвета нет однозначного соответствия, поскольку они расположены в разных устройствах видеокарты. Коды точек хранятся в видеопамяти, а коды их цветов — в специальных регистрах видеокарты. Изменяя содержимое этих регистров, можно изменить все цвета, использованные в изображении, без обращения к видеопамяти, т. е. не изменяя кодов точек образа рисунка. Количество регистров (256) определяет размер кодов точек — 1 байт (8 разрядов). В главе 3 описано программирование рисования и построения графических объектов в этих режимах, а в главе 4 — работа с цветом.

В режимах `direct color` базовые цвета расположены непосредственно в коде точки, который может содержать 2, 3 или 4 байта. На момент написания данной книги трехбайтовый код был обнаружен только у одного семейства акселераторов фирмы ATI. Размещение базовых цветов в коде точки значительно расширяет возможности работы с цветом и позволяет создавать различные спецэффекты, которые широко распространены в современной графике. Программирование в режимах `direct color` описано в главе 7. В этих режимах увеличивается размер кода точки, а следовательно, и пространство видеопамяти, необходимое для хранения содержимого рабочей области экрана. Например, 1 Мбайт видеопамяти достаточно для работы во всех режимах `packed pixel graphics`, кроме `107h`, с разрешением 1280×1024 точки. В то время как для поддержки режима `112h` с разрешением всего 640×480 точек требуется 2 Мбайт, а для режима `11Bh` с разрешением 1280×1024 необходимо 6 Мбайт видеопамяти. Поэтому возможность использования режимов

с высоким разрешением зависит не только от монитора, но и от объема видеопамяти.

Выбор конкретного режима зависит от программиста и от особенностей задачи, которую ему предстоит решать. В режимах packed pixel graphics достигается максимальная производительность видеосистемы, но ограничены возможности манипуляций с цветом. В режимах direct color увеличиваются затраты оперативной и видеопамяти и замедляется процесс построения графических объектов, но существенно расширяются возможности работы с цветом. Вариантов много, есть из чего выбирать.

Коды режимов VESA и OEM. Код используется не только при установке видеорежима, но и во многих других случаях. Поэтому, со времен IBM, он хранится в специальном байте оперативной памяти, расположенном в области данных BIOS с абсолютным адресом 0449h. Старший разряд этого байта имеет специальное назначение, поэтому для записи кода видеорежима остается 7 разрядов и он может изменяться от 0 до 7Fh. Первые 20 значений (от 0 до 13h) отведены для кодов режимов IBM. Использование остальных значений кодов ничем не регламентировано.

Значения кодов видеорежимов, соответствующих стандарту VESA, изменяются от 100h до 11Bh. Такие числа не могут быть записаны в байт и, тем более, в его семь разрядов. Поэтому разработчики видеокарт по своему усмотрению заменяют 9-разрядные коды VESA 7-разрядными кодами OEM. Original Equipment Manufacturer (OEM) в дословном переводе означает "изготовитель оригинального оборудования", в нашем случае — изготовитель видеокарты, которая может собираться из микросхем и деталей других фирм. После установки режимов VESA в байте с адресом 0449h хранятся коды OEM, а в состав BIOS входит специальная таблица их соответствия режимам VESA. Коды OEM уникальны для каждой модели и могут не совпадать даже у видеокарт одного семейства.

По своему усмотрению разработчики видеокарт могут вводить дополнительные режимы, отличающиеся по характеристикам от режимов VESA и IBM. Например, у акселераторов фирм ATI и S3 добавлены режимы с разрешением 320×400, 400×300 и 512×384 точек. Можно предположить, что их удобно использовать при работе с кадрами телевизионных изображений. В соответствии с требованиями VESA коды и характеристики дополнительных режимов должны быть указаны в информационных блоках, хранящихся в области BIOS. Структура этих блоков и способ доступа к ним описаны в следующем разделе данной главы.

Набор режимов, введенный в VBE 1.2, VESA больше никогда не изменяла. В настоящее время выпускаются 20-дюймовые мониторы и видеокарты с объемом памяти 4 Мбайт и более. Это позволяет вводить новые видеорежимы с разрешением 1600×1200 точек. Такие видеорежимы поддерживают, например, видеокарты фирм S3 (ViRGE) и Matrox, в обоих случаях объем видеопамя-

ти составляет 4 Мбайт. При этом код режима packed pixel graphics с разрешением 1600×1200 точек в одном случае 120h, а в другом — 11Ch. Кроме того, карты Matrox поддерживают режимы Hi-Color с указанным разрешением.

Из всего сказанного следует, что работу с новой для вас видеокартой надо начинать с получения исчерпывающей информации о кодах поддерживаемых видеорежимов и их характеристиках. Для этого имеет смысл составить простую программу, которая будет выводить на экран (а лучше в файл) все нужные данные. Способ определения этих данных не сложен, он описан в следующем разделе данной главы.

1.2.2. Информационные функции VBE

Авторы стандарта VESA стремились не только облегчить работу программистов, но и не ограничивать разработчиков в выборе способов улучшения характеристик видеокарт. Компромиссным решением было включение в состав BIOS специальных информационных блоков с основными данными о видеокарте. В частности, они содержат список и характеристики всех поддерживаемых видеорежимов. В данном разделе описана структура основных информационных блоков и способ доступа к ним прикладных задач.

Вызов функций VBE. На любой видеокарте имеется микросхема пассивной, т. е. доступной только для чтения (ROM), памяти, в которой хранится фрагмент BIOS, содержащий структуры данных и подпрограммы, предназначенные для поддержки работы видеосистемы. В частности, к ним относятся функции, обращение к которым (вызов которых) происходит через прерывание int 10h (Video Services).

В состав группы Video Services обязательно входит набор функций для поддержки стандартных режимов IBM. Он необходим, по крайней мере, для нормального выполнения процесса загрузки ПК. Дополнением к нему являются функции VBE, описанные в данном и двух следующих разделах.

Перед обращением к BIOS код запрашиваемой функции помещается в регистр ax. Он состоит из кодов группы и функции в группе. Код группы VBE равен 4Fh, он указывается в старшем байте регистра ax. Код функции для версии VBE 1.2 может изменяться от 0 до 8, он указывается в младшем байте регистра ax. Таким образом, содержимое регистра ax при вызове функций VBE 1.2 может изменяться от 4F00h до 4F08h.

Функции могут иметь входные и выходные параметры, которые передаются в регистрах общего назначения или в сегментных регистрах. Входные параметры нужны для нормального выполнения конкретной функции, а выходные содержат ее возвращаемый результат. Назначение и размещение параметров в регистрах будет описано для каждой функции.

Если запрошенная задачей функция поддерживается BIOS, то в регистр al возвращается код 4Fh.

Важно

Это признак того, что функция могла быть выполнена. При успешном выполнении дополнительно очищается байт ah. В противном случае он содержит код ошибки. Таким образом, код 4Fh в регистре ax является признаком успешного выполнения запроса.

Примеры вызова информационных функций VBE и использования возвращаемых ими данных описаны в главе 2.

Запрос общих данных. Для получения общих данных о видеокарте предназначена функция 4F00h Get SuperVGA Information. Входным параметром является адрес массива размером 256 байтов, при исполнении запроса в него записываются данные о видеокарте. Полный адрес этого массива указывается в регистрах es:di. Форма записи es:di общепринята, она означает, что в регистре es находится сегмент памяти, а в регистре di — расположение (смещение) массива в этом сегменте. При исполнении запроса только первые 20 байтов массива заполняются следующими данными:

- 00 4 байта — VESASignature;
- 04 2 байта — VESAVersion;
- 06 4 байта — OEMStringPtr;
- 0Ah 4 байта — Capabilities;
- 0Eh 4 байта — VideoModePtr;
- 12h 2 байта — TotalMemory.

В первом столбце приведенного списка указаны смещения полей относительно начала массива, адрес которого находится в регистрах es:di.

Поле VESASignature содержит ASCII-коды четырех букв, образующих слово "VESA". Вот эти коды — 56h, 45h, 53h, 41h.

Поле VESAVersion занимает 2 байта, содержащих номер версии и ее редакцию, например 0102 для VBE 1.2 или 0200 для VBE 2.0.

В поле OEMStringPtr находится полный адрес (из области BIOS) начала строки текста, содержащей наименование изготовителя видеокарты. Коды символов соответствуют стандарту ASCII, а строка заканчивается пустым байтом (формат строки ASCIIZ). Полный адрес занимает два слова, в первом из них хранится смещение, а во втором — код сегмента памяти.

Поле Capabilities состоит из 32-х независимых разрядов (битов), в которых указываются специфические особенности видеокарты. Авторы стандарта явно перестарались, даже в новейшей версии VBE 3.0 описано назначение только пяти младших разрядов.

У рядовой видеокарты все 32 разряда поля Capabilities очищены.

Установка нулевого разряда означает возможность увеличения количества разрядов регистров DAC до 8 (см. описание функции 4F08h).

Установка первого разряда означает, что видеоконтроллер не совместим с режимом IBM VGA.

Установка второго разряда означает необходимость синхронизации момента изменения содержимого регистров DAC с обратным ходом луча (см. описание функции 4F09h).

Третий и четвертый разряды описаны только в VBE 3.0, они устанавливаются в тех случаях, когда видеоконтроллер или внешние устройства поддерживают работу со стереоскопическими сигналами.

В поле VideoModePtr находится адрес начала списка видеорежимов, поддерживаемых картой. Первое слово поля содержит смещение, а второе сегмент. Список расположен в области BIOS, код каждого режима занимает одно слово. Признаком конца списка является код 0FFFFh.

В поле TotalMemory указан установленный на видеокарте объем памяти, выраженный в блоках размером 64 Кбайт. 1 Мбайт соответствует 16 блокам (10h). На устаревших моделях видеокарт это поле может быть очищено.

В каких случаях полезна описанная функция? Например, если графическая задача рассчитана на выполнение в защищенном (32-разрядном) режиме работы микропроцессора, то обязательно надо проверять номер версии VBE. Видеокарта может работать в таком режиме, если на ней реализованы функции VBE 2.0, но не VBE 1.2.

Анализ состояния разрядов поля Capabilities и списка поддерживаемых режимов при выполнении задачи едва ли целесообразен. Поддержка выбранного режима обязательно проверяется в задаче, но делается это более надежным способом, чем просмотр списка, поскольку присутствие режима в списке еще не означает его поддержку. Подробно это обсуждается в главе 2.

Целесообразно составить простую программу, которая сохраняет в файле или распечатывает результаты выполнения запроса 4F00h, включая список видеорежимов, и учитывать эти результаты при программировании.

Запрос характеристик видеорежимов. В начале выполнения любой графической задачи обязательно вызывается другая информационная функция, которая возвращает данные, необходимые для настройки на работу с конкретной видеокартой. Здесь описана структура информационного блока, а процессу настройки посвящена специальная глава 2.

Функция 4F01h Get SuperVGA Mode Information позволяет получить информацию о любом из поддерживаемых видеорежимов независимо от того, установлен он или нет. Ее целесообразно вызывать до попытки установить режим, т. к. полученные данные позволяют определить, поддерживает видеокарта работу в нужном режиме или нет, и выполнить ряд подготовительных действий.

Перед вызовом в регистрах es:di указывается адрес массива размером в 256 байтов (как и для функции 4F00h), а в регистр cx помещается код инте-

ресующего вас режима. Если видеокарта содержит VBE, то при возврате в регистре ax записан код 4Fh. Если режим не поддерживается, то признак ошибки не вырабатывается, просто очищаются все поля табл. 1.2.

Результатом исполнения запроса является структура данных, приведенная в табл. 1.2. В первом столбце таблицы указаны смещения полей от начала массива, адрес которого находится в регистрах es:di. Второй столбец содержит размеры полей в байтах. Для примера в трех последних столбцах показаны значения величин, формируемых видеокартой VIRGE /DX /GS семейства S3 при запросе характеристик режимов 101h (640×480, 256 цветов), 110h (640×480, 32K) и 112h (640×480, 16M цветов). В шести случаях последние столбцы слиты в один это означает, что данные в соответствующих полях зависят не от режима, а от характеристик видеокарты.

Таблица 1.2. *Информация, возвращаемая по запросу 4F01h*

Адрес поля	Размер поля	Что хранится в поле	Режим 101h	Режим 110h	Режим 112h
00	2	Атрибуты режима	009Bh	009Bh	009Bh
02	1	Атрибуты окна A	07		
03	1	Атрибуты окна B	00		
04	2	Размер ячейки окна в Кбайт	0040h		
06	2	Размер окна в Кбайт	0040h		
08	2	Код видеосегмента окна A	A000h	A000h	A000h
0Ah	2	Код видеосегмента окна B	A000h	A000h	A000h
0Ch	4	Адрес подпрограммы BIOS	556Ch C000h		
10h	2	Размер строки в байтах	0280h	0500h	0A00h
12h	2	Размер строки в точках	0280h	0280h	0280h
14h	2	Количество строк на экране	01E0h	01E0h	01E0h
16h	1	Ширина символа (текст)	08	08	08
17h	1	Высота символа (текст)	10h	10h	10h
18h	1	Количество планов памяти	01	01	01
19h	1	Количество бит на точку	08	0Fh	20h
1Ah	1	Количество банков видеопамяти	01	01	01
1Bh	1	Модель видеопамяти	04	06	06
1Ch	1	Размер банка в Кбайт	00	00	00

Таблица 1.2 *(окончание)*

Адрес поля	Размер поля	Что хранится в поле	Режим 101h	Режим 110h	Режим 112h
1Dh	1	Номер последней страницы	0Bh	05	02
1Eh	1	Резервный байт	01	01	01
1Fh	1	Размер маски красного цвета	00	05	08
20h	1	Позиция маски красного цвета	00	0Ah	10h
21h	1	Размер маски зеленого цвета	00	05	08
22h	1	Позиция маски зеленого цвета	00	05	08
23h	1	Размер маски синего цвета	00	05	08
24h	1	Позиция маски синего цвета	00	00	00
25h	1	Размер резервного поля	00	01	08
26h	1	Позиция резервного поля	00	0Fh	18
27h	1	Флаги для режимов direct color	00	00	00
Следующие 3 поля заполняются при VBE 2.0 и выше					
28h	4	Адрес начала видеопамяти	0000 F800		
2Ch	4	Адрес свободного пространства	00	00	00
30h	2	Размер свободного пространства	00	00	00

Часть величин, перечисленных в табл. 1.2, уже обсуждалась выше, назначение остальных будет описано по мере изложения материала, в тех случаях, когда они используются при программировании. Здесь мы ограничимся одним замечанием и опишем байт атрибутов режима.

Замечание

Начиная с версии VBE 1.2, появились поля, в которых указываются количество банков видеопамяти (1Ah) и размер банка в килобайтах (1Ch). У всех исследованных видеокарт указан один банк, а его размер равен нулю. Непонятно о каких банках идет речь, тем более, если их размер, выраженный в килобайтах, помещается в пределах одного байта. Поэтому эти поля лучше не использовать до выяснения их назначения.

Атрибуты видеорежима. Нулевое слово информационного массива, возвращаемого по запросу 4F01h, содержит характеристики видеорежима, которые называются атрибутами. Каждый разряд этого слова имеет конкретное назначение, кроме первого, который зарезервирован. Стандарты VBE 1.2 и 2.0 описывают только назначение разрядов младшего байта этого слова (его ад-

рес в массиве 0), а старший байт зарезервирован. В табл. 1.3 показано, что обозначает 1 в каждом из разрядов.

Таблица 1.3. *Назначение разрядов кода атрибутов режимов*

Бит	Код	Что обозначает установка разряда
0	1	Видеокарта поддерживает режим
1	–	Резервный разряд (состояние безразлично)
2	1	BIOS поддерживает вывод на экран
3	1	Используется цветной монитор
4	1	Режим графический
5	1	Режим не совместим с VGA
6	1	Невозможна работа с окнами видеопамяти
7	1	Доступно все пространство видеопамяти (VBE 2.0)

При программировании интерес представляет только состояние некоторых разрядов байта атрибутов. Например, текстовые и графические режимы имеют разные коды и дополнительная проверка состояния бита 4 не имеет смысла. При работе в текстовых режимах надо обязательно проверить состояние бита 2.

Основной интерес представляет состояние бита 0: если он очищен, то видеокарта не поддерживает выбранный режим. При программировании для защищенного режима надо проверять состояние бита 7. Если он установлен, то возможна непосредственная работа со всем пространством видеопамяти без переключения окон (см. раздел 1.2.4).

1.2.3. Основные функции VBE 1.2

BIOS, соответствующая стандарту VBE 1.2, кроме двух информационных, поддерживает еще семь функций, которые описаны в данном разделе.

Установка и чтение режима. Любая графическая задача устанавливает тот видеорежим, на работу с которым она рассчитана. Для этой цели в состав VBE включена специальная функция. Перед ее вызовом целесообразно выполнить функцию 4F01h и проверить возможность работы в выбранном вами режиме, способ проверки описан в главе 2.

Функция 4F02h Set SuperVGA Video Mode устанавливает видеорежим VESA, его код перед вызовом функции помещается в регистр bx. Обычно при установке режимов видеопамять очищается и экран оказывается черным. Если

в регистре bx установить старший (15-й) разряд, то видеопамять не очищается. Сохранение содержимого видеопамяти может быть полезным (и применяется) в некоторых специальных случаях, но не забывайте, что при смене видеорежима картинка на экране изменяется до неузнаваемости.

Начиная с VBE 2.0, используется 14-й разряд регистра bx. Он должен быть очищен, если задача выполняется в реальном (16-разрядном) режиме работы микропроцессора, и установлен, если задача выполняется в защищенном (32-разрядном) режиме. При установке 14-го разряда возможность работы с окнами обычно исключается, поскольку доступно все пространство видеопамяти. Подробнее об этом сказано в разделе 1.2.4.

Функция универсальна в том отношении, что позволяет устанавливать не только режимы VESA, но и режимы IBM. Коды режимов VESA имеют значения от 100h и выше. Обнаружив в регистре bx код с меньшим значением, функция 4F02h вызывает стандартную процедуру BIOS, предназначенную для установки режимов IBM (обычно ее использует функция 0 прерывания int 10h).

Как уже говорилось ранее, при установке режима код VESA заменяется кодом OEM, который записывается в байт с абсолютным адресом 449h, относящимся к области данных BIOS.

Специальная функции для завершения работы в режиме VESA не предусмотрена — она просто не нужна. В этом случае либо устанавливается новый видеорежим (стандартный или VESA), либо завершается выполнение задачи и DOS выполняет все необходимые действия по восстановлению исходного (текстового) видеорежима.

Функция 4F03h Get Current Video Mode позволяет определить код установленного (текущего) режима VESA, который возвращается в регистре bx. Функция введена по той причине, что в байте 449h хранится код OEM. Исполнение запроса 4F03 сводится к чтению кода OEM из указанного байта и преобразованию его в код VESA по таблице соответствия, которая обязательно входит в состав BIOS.

Сохранение и восстановление состояния. Имеется в виду сохранение текущего содержимого регистров цвета видеокарты (DAC) и некоторых величин, хранящихся в области данных BIOS. Напомним, что регистры DAC используются только при работе в режимах packed pixel graphics.

Функция 4F04h Save/Restore Video State выполняет копирование вышеназванных величин в указанный массив или, наоборот, из указанного массива в регистры DAC и в область данных BIOS. Кроме того, она позволяет определить размер массива, необходимый для размещения сохраняемых величин. Перед вызовом функции заполняется несколько регистров, каких — это зависит от запрашиваемого действия.

Обязательно заполняются регистры cx и dx. В регистре cx используются 4 младших разряда, установка каждого из которых определяет группу сохраняемых или восстанавливаемых величин:

- бит 0 — характеристики оборудования из области данных BIOS;
- бит 1 — характеристики видеорежима из области данных BIOS;
- бит 2 — содержимое регистров DAC;
- бит 3 — содержимое регистров.

В регистре dx указывается код запрашиваемого действия:

- 0 — определить размер буфера для размещения сохраняемых величин;
- 1 — сохранить состояние;
- 2 — восстановить ранее сохраненное состояние.

Если регистр dx очищен, то регистры bx и es не заполняются. После исполнения запроса в bx находится количество байтов памяти, которое надо выделить для сохранения указанных в cx групп величин. Теперь можно выбрать расположение массива в памяти и запросить сохранение состояния.

Если в dx задан код 1 или 2, то в паре регистров es:bx указывается полный адрес массива, в котором надо сохранить или из которого нужно восстановить ранее сохраненные величины.

В соответствии со стандартом VGA IBM функция 1Ch прерывания int 10h имеет аналогичное название и выполняет аналогичные действия. В описании VBE дана ссылка на эту функцию и разработчики видеокарт приняли ее как установку к действию. Анализ восстановленного текста BIOS у нескольких видеокарт с версиями VBE 1.2 и VBE 2.0 показал, что выполнение функции 4F04h сводится к проверке кода, указанного в dx. Если он больше чем 2, то запрос отвергается, в противном случае происходит обращение к функции 1Ch. Никакие другие действия не выполняются.

В этой связи возникает вопрос: каким действиям соответствует установка бита 3 регистра cx? Дело в том, что функция 1Ch сохраняет только фрагменты из области данных BIOS и содержимое регистров DAC. Никакие другие величины не сохраняются и не восстанавливаются.

Возможно, что разработчики видеокарт не придали указанному факту должного значения по той причине, что функция 4F04h редко используется при программировании, без нее можно обойтись.

Переключение окон видеопамяти. Окна видеопамяти используются при работе микропроцессора в реальном режиме. В этом случае командам доступно пространство адресов размером не более 64 Кбайт. У современных карт объем видеопамяти намного (в десятки раз) превосходит указанную величину. Поэтому видеопамять делится на одинаковые сегменты размером 64 Кбайт, которые принято называть окнами.

Для доступа к видеопамяти выделяется видеобуфер (или видеосегмент). Это пространство адресов размером 64 Кбайт, в графических режимах его адрес (код) обычно, но не обязательно, равен `A000h`, а в текстовых `B800h`. Код видеосегмента это не более чем признак. Обнаружив его, видеоконтроллер записывает данные в текущее окно видеопамяти или считывает их оттуда.

Текущий номер окна хранится в одном из регистров видеоконтроллера и является частью абсолютного адреса видеопамяти. `BIOS` поддерживает работу с этим регистром.

Функция `4F05h CPU Video Memory Control` читает или изменяет номер текущего окна видеопамяти. Наличие этой функции позволяет задачам работать со всем пространством видеопамяти.

Перед вызовом функции `4F05h` в регистре `bx` указываются номер окна и запрашиваемое действие. Нуль в регистре `bh` (старший байт регистра `bx`) означает установку нового окна с номером, указанным в `dx`. Единица в регистре `bh` означает чтение номера текущего окна, он возвращается в регистре `dx`. В главе 2 подробно описана техника переключения окон и формат, в котором указываются их номера.

Стандартом VESA предусмотрена возможность существования у видеокарты двух окон (`A` и `B`). Нуль в регистре `bl` (младшем байте регистра `bx`) соответствует окну `A`, а единица — `B`. Почти все исследованные автором видеокарты поддерживали работу только с окном `A`. Исключением являются видеокарты фирмы ATI Technologies, у которых окно `A` доступно только для записи данных в видеопамять, а окно `B` — только для чтения.

Программная реализация функции `4F05h` такова, что основные действия выполняет отдельная процедура (подпрограмма), к которой задача может обращаться непосредственно, т. е. минуя прерывание `int 10h`. Функция `4F01h` размещает адрес этой подпрограммы в выходном массиве в двойном слове со смещением `0Ch` (см. табл. 1.2). Стандарт VESA рекомендует прямое обращение к подпрограмме вместо использования функции `4F05h`.

Перемещение по видеопамяти. Сразу после установки видеорежима на экране отображается содержимое младшей части памяти. Будем называть ее рабочей или отображаемой областью. Размер рабочей области зависит от характеристик режима и равен произведению размера строки в байтах на количество строк, помещающихся на экране.

В процессе выполнения задачи может возникнуть необходимость перемещения рабочей области в другой участок видеопамяти. Например, для просмотра отдельных частей большого рисунка или текста, который полностью не помещается на экране. В англоязычной литературе в таких случаях используется специальный термин `scrolling` — прокрутка, перемещение, просмотр. Механизм прокрутки используют многие приложения для Windows, он реализуется в виде горизонтального и вертикального лифтов.

Для реализации механизма прокрутки, в первую очередь должна существовать возможность размещения в видеопамяти большого изображения. Высота изображения (количество строк) может быть произвольной, лишь бы хватило видеопамяти. Но ширина ограничена величиной, которая в документации называется `logical scan line`. В процессе отображения видеопамяти контроллер, отсчитав указанное в ней количество байтов, начинает выводить следующую строку на экране. Если ширина рисунка больше чем `scan line`, то продолжение текущей строки окажется на экране в следующей строке и изображение будет искажено. Чтобы это не произошло, надо установить значение `scan line` равным ширине рисунка.

В табл. 1.2 значение `scan line` расположено в поле `10h`, оно равно произведению количества точек в строке (поле `12h` табл. 1.2) на размер кода точки в байтах. В поле `19h` табл. 1.2 указано количество разрядов в коде точки. Количество байтов определяется делением количества разрядов на 8.

Функция `4F06h Get/Set Logical Scan Line Length` позволяет прочитать или изменить логический размер строки, т. е. определить или изменить адрес видеопамяти, начиная с которого контроллер выводит новую строку на экран.

Перед обращением к `BIOS` в регистр `bl` помещается код подфункции: 0 для установки (записи) и 1 для чтения логического размера строки, который указывается или возвращается в регистре `cx` в виде количества точек.

В обоих случаях после исполнения функции в регистрах находятся следующие величины: `bx` — количество байтов в строке, `cx` — количество точек в строке, `dx` — максимально возможное количество строк указанного размера. Процедура `BIOS` вычисляет содержимое `dx` путем деления объема памяти, установленной на видеокарте, на размер строки в байтах. На практике содержимое `dx` используется крайне редко.

Начиная с `VBE 2.0`, введены еще два варианта запроса функции `4F06h`. Код 2 в регистре `bl` отличается от кода 0 только тем, что в регистре `cx` указывается размер строки в байтах, а не в точках. Код 3 позволяет определить максимально возможный логический размер строки для установленного видеорежима. Процедура `BIOS` вычисляет его исходя из характеристик режима и установленного на видеокарте объема памяти.

Проверка трех видеокарт, соответствующих `VBE 2.0`, показала, что `BIOS` видеокарты `VIRGE /DX /GX` содержит ошибку и при указании кода 3 в регистре `bl` функция `4F06h` выдает совершенную чушь. При кодах 0, 1 и 2 функция выполняется правильно.

Предположим, что при работе в видеорежиме с разрешением 640×480 точек установлен логический размер строки 736 точек. В таком случае при обработке каждой строки видеоконтроллер выводит на экран первые 640 точек, а остальные 96 просто пропускает. На экране будет видна левая верхняя часть

изображения размером 640×480 точек. Для того чтобы увидеть его остальную часть, надо переместить рабочую область видеопамяти.

Функция 4F07h Get/Set Display Start устанавливает или читает координаты левого верхнего угла видимой области видеопамяти, выраженные в виде номеров строки и столбца.

Перед обращением к BIOS в регистре bl указывается 0 для установки новых значений координат или 1 — для чтения ранее установленных (текущих) координат. Для установки в регистре cx указывается номер столбца, а в регистре dx — номер строки (номера строк и столбцов начинаются с нуля). При чтении в cx и dx возвращаются текущие значения указанных величин.

Начиная с VBE 2.0, появилась возможность синхронизировать установку новых значений координат рабочей области с моментом обратного хода луча. Для этого в регистре bl указывается код 80h.

Таким образом, функция 4F06h позволяет создать условия для прокрутки в горизонтальном направлении, а функция 4F07h выполняет указанную прокрутку.

Регистры цвета видеокарты. На видеокартах имеется 256 регистров DAC, в которых хранятся коды базовых цветов. Они применяются только при работе в видеорежимах packed pixel graphics и не используются в режимах direct color. Базовых цветов три — красный, зеленый и синий. В соответствии со стандартом IBM VGA код базового цвета занимает 6 двоичных разрядов. У некоторых современных видеокарт, например Matrox, появилась возможность увеличения размера кода до 8 разрядов. В таком случае в поле Capabilities будет установлен нулевой разряд (см. описание функции 4F00h).

Функция 4F08h Get/Set DAC Palette Control предназначена для определения или изменения размера кода базовых цветов, хранящихся в регистрах цвета. Для установки нового размера регистр bl очищается (код 0), а в регистр bh записывается нужный размер кода базового цвета в битах. Для чтения установленного размера базовых цветов в регистр bl записывается код 1, а текущий размер возвращается в регистре bh.

Если видеокарта не позволяет изменить размер кода, то при попытке установки указанная в bh величина заменяется на 6.

Таков полный перечень функций VBE 1.2. Возможно, вы обратили внимание на отсутствие в нем функций, выполняющих запись кодов точек в видеопамять или их чтение. Такие функции просто не нужны, поскольку возможны непосредственные запись и чтение содержимого байтов, слов или двойных слов видеопамяти, т. е. в ней могут находиться операнды команд.

В расположенной на видеокарте BIOS сохранились функции записи и чтения кодов точек во всех графических стандартах IBM. Их вызов осуществляется

через прерывание int 10h, коды запросов 0Ch для записи точки и 0Dh для чтения. Эти функции нужны только в тех случаях, когда при работе с видеопамятью используются внутренние регистры видеокарты. Уже с появлением режима VGA IBM необходимость в их использовании при программировании отпала.

В заключение следует отметить, что последующие версии VBE совместимы сверху вниз с VBE 1.2, поэтому описанные функции выполняют все современные видеокарты. Вероятность массового выпуска видеокарт, рассчитанных только на 32-разрядный режим работы, маловероятна из соображений совместимости.

1.2.4. Новые возможности VBE 2.0

Уже на момент публикации VBE 1.2 выпускались микропроцессоры, поддерживающие работу в защищенном (32-разрядном) режиме и существовало соответствующее программное обеспечение. Поэтому возникла необходимость в создании следующей версии, учитывающей особенности защищенного режима. Она была опубликована в ноябре 1994 года и получила название VBE 2.0. Интересно, что в этой версии были добавлены только две новые функции и внесены некоторые дополнения в ранее существовавшие, о них говорилось в предыдущем разделе.

Линейное пространство адресов. Защищенный режим отличается от реального тем, что не только данные, но и адреса содержат 32 разряда. Соответственно, размер адресуемого в командах пространства составляет 4 294 967 296 байтов или 4 Гбайт (4 биллиона байтов). У современных ПК реальный объем оперативной памяти намного меньше указанной величины, любой адрес оперативной памяти не только помещается в 32-разрядном регистре, но и не заполняет весь регистр, часть старших разрядов остается свободной.

Оперативная память занимает младшую часть всего пространства адресов, поэтому BIOS и область ввода-вывода перенесены в его старшую часть. Самые старшие адреса отведены для BIOS, например, при перезагрузке ПК происходит обращение к адресу FFFFFFF0h.

Видеопамять, как и оперативная, образует линейное пространство адресов, которое в документации VESA называется FFB (Flat Frame Buffer) или LFB (Linear Frame Buffer). В защищенном режиме возможен произвольный доступ к любым адресам видеопамяти без использования механизма переключения окон. Физический (абсолютный) адрес начала LFB хранится в поле 28h (см. табл. 1.2), он не зависит от видеорежима. В качестве примера в табл. 1.2 приведено значение F8000000h.

Содержимое следующих двух полей (2Ch и 30h) зависит от видеорежима. Эти поля заполнены не у всех видеокарт, а в VBE 3.0 вообще объявлены резерв-

ными потому, что их содержимое можно легко вычислить. В поле 2Ch должна храниться сумма адреса начала LFB и размера рабочей области памяти. В поле 30h должна находиться разность между общим объемом видеопамяти и размером ее рабочей области.

Замечание

Напомним, что размер рабочей области вычисляется как произведение величин, указанных в полях 10h (размер строки в байтах) и 14h (количество строк) (см. табл. 1.2).

Физический адрес нельзя использовать для работы с видеопамятью, он должен быть предварительно преобразован в линейный адрес. Способ такого преобразования и сегментный регистр, указываемый при работе с LFB, зависят от используемого задачей распределения памяти. Например, если применяется простая линейная (гладкая — flat) модель, то физический адрес просто уменьшается на базовый адрес области данных, а для доступа к любым адресам используется сегментный регистр DS.

Для того чтобы видеоконтроллер поддерживал работу с LFB при установке видеорежима (функция 4F02), в регистре bx кроме указания кода режима надо установить 14-й разряд, например, bx = 4101h для установки режима 101h и разрешения работы с LFB.

В большинстве случаев после разрешения работы с LFB исключается возможность работы с окнами видеопамяти. При этом запросы функции 4F05h отвергаются и в регистре ah возвращается код ошибки 3. Но в литературе встречаются сведения о существовании видеокарт, без уточнения их названия, одновременно допускающих оба способа работы с видеопамятью.

Следует заметить, что поддержка LFB является самым существенным нововведением VBE 2.0. При работе с LFB исключается необходимость контроля адресов видеопамяти в задачах для определения моментов, когда надо изменять текущее окно. Это значительно упрощает и ускоряет манипуляции с графическими объектами.

Работа с регистрами палитры. В режимах packed pixel graphics коды цветов точек хранятся в регистрах палитры (их 256). Эти регистры недоступны обычным командам. Для записи или чтения их содержимого необходимо обращение к внутренним регистрам видеокарты.

В VBE 1.2 отсутствует специальная функция, выполняющая чтение или изменение содержимого регистров DAC палитры. Прикладные задачи могут использовать для этой цели стандартные функции IBM VGA, примеры работы с ними описаны в главе 4 данной книги.

В VBE 2.0 такая функция введена. В отличие от функций IBM VGA она использует другой формат палитры, позволяет изменять содержимое блока регистров DAC во время обратного хода луча и поддерживает работу с дополнительным набором регистров палитры, если таковой имеется.

Функция 4F09h Get/Set Block DAC Registers позволяет сохранить или изменить текущую палитру цветов (содержимое блока регистров DAC). Код выполняемого действия указывается в регистре bl. При сохранении блока регистров bl=1, а при записи bl=0. В регистре cx задается количество сохраняемых или изменяемых регистров (размер блока), а в регистре dx — номер первого сохраняемого регистра (0 — 255).

Для размещения палитры в оперативной памяти надо выделить массив размером 4*N байтов, где N — размер блока, указанный в регистре cx (напомним, что он не может быть больше чем 256). Содержимое каждого регистра занимает 4 подряд расположенных байта, в первых трех находятся коды синего, зеленого и красного цветов, а четвертый очищен. Полный адрес массива задается в регистрах es:di (es — сегмент, di — смещение). Такое расположение базовых цветов принято в палитре формата BMP (см. приложение А данной книги). Отметим, что в формате BMP код базового цвета занимает 8 разрядов, а у большинства видеокарт он составляет 6 разрядов. Поэтому при использовании данной функции может потребоваться преобразование хранящихся в файле кодов базовых цветов. Подробнее об этом сказано в главе 4 данной книги.

Некоторые модели видеокарт содержат дополнительную группу регистров палитры. Для работы с дополнительной группой в регистре bl указываются коды 2 или 3 (вместо 0 или 1). Если дополнительная палитра отсутствует, то при возврате из BIOS в регистре ah находится код ошибки 2.

В некоторых моделях видеокарт содержимое регистров палитры можно изменять только во время обратного хода луча, в противном случае на экране появляются помехи ("снег"). Признаком такой особенности видеокарты является установка бита 2 в поле Capabilities (см. описание функции 4F00h). В таком случае вместо кода 0 в регистре bl указывается 80h.

Интерфейс защищенного режима. Большинство функций BIOS, в том числе и функций VBE, независимо от версии, рассчитано на выполнение в реальном (16-разрядном) режиме работы микропроцессора. Если задача выполняется в защищенном (32-разрядном) режиме, то для обращения к функциям BIOS необходим временный переход в реальный режим работы микропроцессора. Это увеличивает количество вспомогательных действий при вызове функций BIOS и замедляет процесс их выполнения. Замедление становится ощутимым, если функции вызываются часто.

Разработчики VBE 2.0 предусмотрели возможность непосредственного вызова процедур, дублирующих функции 4F05h, 4F07h и 4F09h, но рассчитанных на выполнение в защищенном режиме. Прикладная задача определяет адреса точек входа в указанные процедуры следующим способом.

Функция 4F0Ah Return VBE 2.0 Protected Mode Interface возвращает адрес массива, содержащего указанные выше процедуры и некоторые дополни-

тельные данные. Перед ее вызовом надо очистить регистр bl. После исполнения запроса в регистры возвращаются следующие величины:

- es — сегмент массива, расположенного в области BIOS в формате для реального режима (чаще всего код C000h);
- di — адрес (смещение) начала массива в этом сегменте;
- cx — размер массива в байтах.

Первые три слова массива es:[di+0], es:[di+2] и es:[di+4] содержат адреса (смещения относительно начала массива) точек входа в процедуры, дублирующие функции 4F05h, 4F07h и 4F09h для защищенного режима. Процедуры полностью перемещаемые, они могут выполняться как непосредственно в ROM BIOS, так и в оперативной памяти, разумеется, после предварительного копирования, для чего и нужен размер массива, возвращаемый в регистре cx.

Указанные процедуры должны вызываться как ближние, т. е. без смены сегментного регистра (см. приложение В данной книги). Если задача использует простую линейную модель памяти, то доступ к области BIOS происходит без смены сегментного регистра и нет необходимости копировать процедуры в оперативную память. Если же пространство адресов сегментировано, то процедуры надо скопировать в сегмент кодов. В таком случае их вызов будет происходить без смены сегментного регистра.

Фактически процедуры не являются полными аналогами функций, выполняемых в реальном режиме. Имеются следующие различия:

- Аналог 4F05h поддерживает работу только с одним окном A.
- Аналог 4F07h лишь устанавливает новое начало отображаемого участка видеопамяти, причем вместо номера строки и столбца при вызове указывается полный (32-разрядный) адрес начала отображаемой области. Его старшая часть помещается в регистр dx, а младшая — в cx.
- Аналог 4F09h поддерживает только основной набор регистров DAC.

Кроме перечисленных функций описываемый массив может содержать перечень номеров портов видеокарты и адресов, которые задача может использовать для ввода и вывода данных. Если такой список присутствует, то его смещение относительно начала массива указано в слове es:[di+6]. Если это слово очищено, то списка в массиве нет. Для большинства программистов этот список не представляет интереса, поскольку не известно назначение указанных в нем портов и адресов. Стандарт VBE 2.0 оговаривает только способ их хранения в таблице, но не назначение.

Замечание

Целесообразность введения функции 4F0Ah не очевидна, поэтому в версии VBE 3.0 она не относится к числу обязательных.

Программирование работы в защищенном режиме в данной книге не рассматривается. Если вас интересует этот вопрос, то советуем прочитать

статьи Андрианова С. А. в журналах "Мир ПК" [1, 2], в них приведены простые примеры работы с описанными функциями. При наличии доступа к сети Internet подшивки этих журналов можно найти на www.opensystems.ru.

Заключительные замечания. Несмотря на небольшое число функций (11), их состав оказался вполне достаточным. Авторы версии VBE 3.0, которая опубликована в сентябре 1998 года, не ввели ни одной новой функции, а только расширили возможности существующих с учетом новейших достижений разработчиков видеокарт.

В разные годы ассоциация VESA выпустила несколько небольших документов с описанием функций специального назначения. Формально они не относятся к VBE и их описание отсутствует во всех стандартах. Даже авторы VBE 3.0 ограничились их перечислением и весьма лаконичным комментарием. Вот перечень этих функций без комментариев:

- 4F10h — Power Management Extension (PM) для стандарта DPMS;
- 4F11h — Flat Panel Interface Extension (FP);
- 4F13h — Audio Interface Extension (AI);
- 4F14h — OEM Extension, вводимые по усмотрению разработчиков;
- 4F15h — Display Data Channel (DDC).

Функции с кодами 10h, 14h и 15h были обнаружены автором при исследовании BIOS видеокарт с версиями VBE 1.2 и VBE 2.0. Если вы умеете восстанавливать исходные тексты BIOS, то можно проверить, какие функции поддерживает интересующая вас видеокарта.

Общая характеристика стандарта VBE закончена и мы переходим к описанию программирования работы с графикой в видеорежимах VESA.

ГЛАВА 2

Особенности работы в режимах VESA

Стандарт VESA создавался для того, чтобы графические задачи могли самостоятельно, или при минимальном вмешательстве оператора, настроиться на работу с установленной на ПК видеокартой. В этой главе описано, как производится такая настройка.

Любой стандарт оставляет некоторую свободу действий производителям оборудования, поэтому существуют модели видеокарт, которые формально соответствуют требованиям VESA, а фактически их программирование все же имеет специфические особенности. Тем не менее, возможна единая схема, в которую укладывается работа с большинством наиболее распространенных видеокарт. Мы рассмотрим элементы этой схемы работы с видеокартами, а обнаруженные автором отклонения от нее будут специально оговариваться.

Независимо от видеорежима VESA, который используется в задаче, перед началом работы с графикой должны быть выполнены определенные действия, обеспечивающие в дальнейшем ее корректную работу, универсальность и независимость от модели видеокарты. Вот перечень этих действий:

- проверить, поддерживает `BIOS` требуемый видеорежим или нет;
- проверить, достаточно видеопамяти для выбранного режима или нет;
- получить и сохранить в области данных характеристики режима;
- прочитать и сохранить исходный видеорежим (не обязательно);
- установить требуемый видеорежим VESA;
- вычислить константу для коррекции номеров окон видеопамяти;
- настроить подпрограммы для работы с видеоокнами;
- определить размер и расположение полей базовых цветов.

При выполнения перечисленных действий используются функции `VBE`, описанные в предыдущей главе.

2.1. Проверка поддержки видеорежима

Для нормального выполнения любой прикладной задачи должны быть созданы соответствующие условия. Поэтому при разработке задач, как правило, предусматриваются вспомогательные действия, направленные на проверку и создание таких условий. В этом отношении графические задачи не являются исключением.

Предварительные действия, выполняемые в графических задачах, можно разделить на две категории по признаку их зависимости от видеорежима. В первую очередь обычно выполняются те из них, которые не зависят от видеорежима, используемого в задаче. К ним относятся проверки операционной среды (версии DOS), наличие необходимого пространства оперативной памяти, изменение значений векторов прерываний, формирование многократно используемых величин и т. п.

От номера версии DOS зависит набор функций, выполняемых по запросам прикладных задач. Базовый набор функций, предназначенных для работы с файловой системой, был сформирован в версии 3.0 и с тех пор существенно не изменялся. На большинстве современных компьютеров используются версии не ниже 6.0, поэтому вопрос о необходимости проверки версии DOS решает программист с учетом особенностей создаваемой задачи.

Графические задачи обычно нуждаются в большом пространстве оперативной памяти, соизмеримым с объемом видеопамяти, необходимым для поддержания используемого видеорежима. Поэтому в процессе подготовительных действий обязательно производится определение доступного для задачи пространства оперативной памяти, его резервирование и распределение для внутреннего использования. Как это делается, описано в приложении Б данной книги.

В процессе выполнения задачи могут использоваться специальные таблицы. Если это общесистемные таблицы, то надо определить их расположение в оперативной памяти или в области `BIOS`. Если же таблицы являются собственностью задачи, то их надо разместить в доступном пространстве памяти. Вспомогательные действия, выполняемые при работе с таблицами шрифтов, содержащими изображения букв, цифр и других символов, описаны в главе 5.

В тех случаях, когда задача должна реагировать на прерывания от внешних устройств, при выполнении подготовительных действий надо создать условия для вызова прерывающих подпрограмм. В главах 5 и 6 описана настройка подпрограмм, реагирующих на прерывания, поступающие от системного таймера и манипулятора "мышь".

В процессе настройки могут быть выявлены условия, препятствующие выполнению задачи или требующие вмешательства оператора, например не соответствующая версия DOS, недостаточный объем оперативной памяти

и пр. В таком случае на экран выводятся аварийные сообщения или поддерживается диалог с оператором, если он может что-то изменить. Вывод сообщений проще программировать в текстовом режиме, который установлен DOS перед вызовом задачи. Поэтому переход в графический видеорежим целесообразно производить после всех описанных проверок.

Когда невозможна установка видеорежима? Установке требуемого видеорежима могут препятствовать три причины:

- режимы VESA вообще не поддерживаются;
- не поддерживается конкретный видеорежим;
- недостаточно видеопамяти для работы в этом режиме.

Возникновение первой причины может означать, что на компьютере установлена очень старая видеокарта. Кроме того, если задача была вызвана не из DOS, а из Windows 3.X, 9X, ME или 2000, то при определенных условиях она может не получить доступ к функциям `BIOS`. Если же задача была вызвана из DOS и качество видеокарты не вызывает сомнений, то имеет смысл проверить состояние системного программного обеспечения, установленного на компьютере.

Возникновение второй причины означает, что предельно допустимый объем видеопамяти, поддерживаемый видеокартой, не позволяет работать в выбранном видеорежиме.

Возникновение третьей причины означает, что установленный на видеокарте объем памяти меньше предельно допустимого и того, который нужен для работы в запрашиваемом видеорежиме.

Объем видеопамяти, необходимый для работы в конкретном режиме, можно подсчитать на основании данных, приведенных в табл. 1.1. Если окажется, что видеокарта имеет достаточный объем памяти, то имеет смысл проверить состояние системного программного обеспечения, установленного на компьютере. Разумеется, вы должны быть уверены в корректности самой задачи.

Размещение массива Info. Рассмотрим, как производится проверка возможности работы в выбранном видеорежиме. Для проверки соответствия видеокарты стандарту VESA и получения общей информации о ней предназначена функция `4F00h` прерывания `int 10h` (см. главу 1). Перед ее вызовом в паре регистров `es:di` надо указать адрес буфера размером в 256 байтов для размещения полученных данных. Указание буфера обязательно, независимо от того, будет задача использовать эти данные или нет.

Буфер такого же размера потребуется и при следующей проверке для размещения основных данных, используемых в процессе настройки задачи. Выделять специально для него постоянное место в памяти, например, в разделе данных задачи, не целесообразно, поскольку полученная информация

нужна только при выполнении подготовительных действий. В задаче наверняка должен быть предусмотрен буфер большего размера для чтения файлов, содержащих рисунки. В приложении Б данной книги описано, как создается такой буфер. При выполнении подготовительных действий этот буфер свободен и в его начало можно поместить служебную информацию. Предположим, что буфер существует и сегмент, в котором он расположен, хранится в переменной `Info`, описанной в разделе данных задачи.

Проверка существования VBE. В примере 2.1 приведен текст фрагмента программы, проверяющего соответствие видеокарты стандарту VESA.

Пример 2.1. Проверка поддержки стандарта VESA

```
test_1:push   es          ; сохранение содержимого es
mov     es, Info          ; значение сегмента буфера Info
xor     di, di            ; адрес начала буфера
mov     ax, 4F00h         ; код запрашиваемой функции
int     10h               ; обращение к BIOS
cmp     ax, 4Fh           ; стандарт VESA поддерживается ?
jz      test_2            ; -> да, продолжение проверок
pop     es                ; нет, выполнение задачи не возможно
                          ; вывод аварийного сообщения
```

Выполнение примера 2.1 начинается с сохранения в стеке содержимого регистра `es` (если он еще не использовался и его содержимое не имеет значения, то команды `push es` и `pop es` можно исключить из текста примера). Далее в регистр `es` записывается значение сегмента, содержащего буфер `Info`, а `di` очищается для размещения данных с начала буфера. После этого в регистр `ax` записывается код запрашиваемой функции и происходит обращение к прерыванию `BIOS int 10h`. Если видеокарта соответствует стандарту VESA, то при возврате из `BIOS` в `ax` находится код `4Fh`, это и проверяет команда `cmp ax, 4Fh`. Если результат проверки положительный, то следующая команда `jz` выполнит переход на метку `test_2`, которая обозначает начало примера 2.2.

Если результат проверки отрицательный (код отличается от `4Fh`), то дальнейшее выполнение задачи не возможно. На экран надо вывести аварийное сообщение типа "`Видеокарта не поддерживает режимы VESA`" и прекратить выполнение задачи. Как можно подготовить текст сообщения и вывести его на экран, описано в главе 5, посвященной работе с текстом.

В случае успешного выполнения запроса, в буфере `Info` находится общая информация о видеокарте (см. главу 1). Как правило, ее использует только Windows при выборе драйверов для работы с конкретной картой. Нам драйверы подбирать не надо, поэтому интерес может представлять только объем видеопамяти, указанный в слове массива `Info` со смещением `12h`. Эта вели-

чина выражена в блоках размером по 64 Кбайт, поэтому 1 Мбайт соответствует код `10h`. Объем видеопамяти в случае его использования при выполнении задачи надо сохранить в области данных, т. к. уже на следующем шаге содержимое массива `Info` изменится.

Проверка поддержки видеорежима. Если видеокарта соответствует стандарту VESA, то надо проверить, поддерживается выбранный вами видеорежим или нет, и одновременно прочитать в массив `Info` информацию о нем. В примере 2.2 показано, как можно выполнить эти действия.

Пример 2.2. Чтение информации о режиме и проверка его поддержки

```
test_2:mov    ax, 4F01h   ; код запрашиваемой функции
       mov    cx, NewMode ; код нужного видеорежима
       int    10h     ; обращение к BIOS
       pop    es      ; восстановление содержимого es
       cmp    ax, 4Fh; нужный режим поддерживается?
       jz     test_3 ; -> да, продолжение проверок
                     ; нет, указанный видеорежим не поддерживается
```

Предполагается, что пример 2.2 выполняется после примера 2.1, поэтому пара регистров `es:di` содержит адрес буфера `Info` для записи информации о режиме и в стеке сохранено содержимое регистра `es`. Выполнение примера начинается с указания кодов запрашиваемой функции и нужного видеорежима, после чего происходит обращение к `BIOS` для выполнения запроса. После возвращения в задачу восстанавливается сохраненное в стеке исходное содержимое регистра `es`. Если видеокарта рассчитана на поддержку выбранного видеорежима, то в `ax` будет находиться код `4Fh`, это и проверяет предпоследняя команда примера. Если условие выполнено, то произойдет переход на метку `test_3` (начало примера 2.3) для продолжения проверок.

Если код в регистре `ax` отличается от `4Fh`, то видеокарта не поддерживает требуемый видеорежим. В зависимости от конкретных особенностей задачи, ее выполнение может быть либо прервано, либо предпринята попытка перейти на работу в другом режиме, требующем меньший объем видеопамяти. Если вы уверены в корректности задачи и в том, что видеокарта поддерживает нужный режим, то проверьте системное программное обеспечение, установленное на компьютере. В практике автора был случай, когда драйвер манипулятора "мышь" реагировал на выполнение функции `4F01h`, к которой он не имел никакого отношения. В остальном работа этого драйвера не вызывала никаких нареканий. Ошибка была выявлена при работе задачи, в которой проверялось исполнение запросов функций VESA, и устранена путем замены драйвера.

Заключительная проверка. Если в регистре `ax` находится код `4Fh`, то остается последний штрих — проверить достаточность реально существующего

(а не предельно допустимого) объема видеопамяти для поддержки видеорежима. При сборе информации о запрошенном режиме BIOS выполняет нужные для проверки вычисления и сравнения, результат которых помещается в нулевой разряд нулевого байта массива Info. Задаче остается только проверить состояние этого разряда.

После выполнения примера 2.2 было восстановлено исходное содержимое регистра es и доступ к массиву Info с использованием этого регистра уже невозможен. Для дальнейшей работы с данными о режиме надо выделить другой сегментный регистр, например, fs или gs, который не используется функциями BIOS.

Фрагмент программы, выполняющий заключительную проверку, приведен в примере 2.3. Предполагается, что он выполняется после примеров 2.1 и 2.2, поэтому регистр di очищен и указывает начало буфера, содержащего данные о режиме. Для доступа к этим данным используется сегментный регистр fs, поэтому в него записывается значение переменной Info.

Пример 2.3. Заключительная проверка поддержки видеорежима

```
test_3: mov   fs, Info            ; сегмент с данными о режиме
        test  byte ptr fs:[di], 1 ; объем видеопамяти достаточен ?
        jne   stmd                ; -> да, конец проверок
        ; Недостаточно видеопамяти для поддержки нужного режима
```

В команде test запись byte ptr явно указывает, что операндом является байт. Тип операнда указывается в тех случаях, когда Макроассемблер не может его определить по записи команды. Если нулевой разряд байта установлен, то объем памяти достаточен и команда jne передаст управление на метку stmd, указанную перед первой командой примера 2.4.

Если нулевой разряд очищен, то объем видеопамяти не достаточен для поддержки выбранного режима. Что делать в подобных случаях, говорилось при описании примера 2.2.

Установка видеорежима. После успешного выполнения трех описанных проверок можно либо сразу установить рабочий видеорежим, а затем продолжить подготовительные действия, либо, наоборот, сначала завершить всю подготовку и лишь после этого устанавливать рабочий видеорежим. Только из соображений удобства описания, мы сначала рассмотрим установку видеорежима, а затем вернемся к подготовительным действиям.

Установку режимов VESA осуществляет функция 4F02h. Если при завершении задачи должен быть восстановлен исходный видеорежим, то его значение сохраняется в разделе данных. Следующий фрагмент программы иллюстрирует способ сохранения исходного и установку нового видеорежима.

Пример 2.4. Сохранение исходного видеорежима и установка нового

```
stmd:   mov   ax, 0F00h      ; код функции "Чтение видеорежима"
        int   10h            ; BIOS читает текущий видеорежим
        mov   OldMode, al    ; сохранение кода видеорежима
        mov   bx, NewMode    ; код одного из режимов VESA
        mov   ax, 4F02h      ; код запрашиваемой функции BIOS
        int   10h            ; BIOS исполняет запрос
        cmp   ax, 4Fh        ; режим установлен ?
        jz    succ           ; -> да, на продолжение программы
        ; Ошибка при установке видеорежима
```

В тексте примера 2.4 использованы имена `OldMode` и `NewMode`. Первое из них может быть только именем байта, расположенного в области данных программы. Как создаются такие имена, описано в последнем разделе данной главы. `NewMode` может быть именем расположенного в области данных слова, или константы, которой заранее присвоено конкретное значение, скажем, `NewMode = 110h`. Кроме того, код режима может быть указан в команде явно, например, `mov bx, 110h`. Если при выполнении задачи видеорежим устанавливается только один раз, то выбор способа указания `NewMode` произволен. Тем не менее использование переменных является более универсальным и предпочтительным приемом.

Первые три команды примера 2.4 считывают текущий видеорежим и сохраняют его в байте `OldMode`. Следующие три команды устанавливают новый режим, в котором будет работать задача. После второго возврата из `BIOS` анализируется содержимое регистра `ax`. Если в нем записан код `4Fh`, то нужный режим установлен и происходит переход на начало примера 2.5 (метка `succ`). Отличие кода от `4Fh` означает чрезвычайную ситуацию, поскольку предварительно были выполнены проверки, показавшие, что видеокарта поддерживает нужный режим. Если ваша задача заведомо корректна, то остается только проверять общее состояние компьютера и программного обеспечения.

В примере 2.4 для определения значения исходного видеорежима издается запрос `0Fh` прерывания `int 10h`. При его исполнении `BIOS` просто считывает в регистр `al` содержимое байта `0000:0449`, относящегося к области данных `BIOS`. Выполнить эти действия можно непосредственно в задаче без обращения к `BIOS`. В таком случае исключаются примерно 30 команд, которые `BIOS` выполняет при расшифровке запроса, сохранении и восстановлении содержимого регистров.

Коды VESA и OEM. Размер кода режимов VESA превышает размер байта, поэтому при установке этих режимов в байт `0000:0449` записывается так называемый код OEM, т. е. код, выбираемый по усмотрению разработчиков видеокарты. В `ROM BIOS` имеются две таблицы соответствия для преобразова-

ния кодов видеорежимов из VESA в OEM и обратно. Никаких соглашений относительно значений кодов OEM не существует, кроме того, что его размер не превышает семи разрядов, а значения отличаются от кодов видеорежимов IBM. Функция `VBE 4F03h` читает код текущего видеорежима из байта `0000:0449` и преобразует его в код режима VESA по таблице соответствия, хранящейся в `ROM BIOS`. Если в указанном байте находился код одного из видеорежимов IBM, то его значение не изменяется. В этом отношении функция `4F03h` более универсальна, чем функция `0Fh`.

Стандартная функция установки видеорежимов IBM (функция `00` прерывания `int 10h`) установит режим VESA, если при обращении к ней в регистре `al` указать соответствующий код OEM. Так что коды OEM не совсем бесполезны, они, например, используются в драйверах для Windows.

Если вы владеете техникой дисассемблирования, то можно сравнительно просто найти в `ROM BIOS` таблицы соответствия кодов режимов VESA и OEM. Эти таблицы используются функциями `VBE 4F02h` и `4F03h`.

2.2. Обработка информации о режиме

При программировании графики надо знать количество точек на экране по горизонтали и вертикали, способ кодирования цвета, расположение базовых цветов, способ переключения окон видеопамяти, значение сегмента для доступа к видеопамяти и некоторые другие данные. Одни из этих величин зависят от видеорежима, другие — от особенностей видеокарты, третьи — от установленного на компьютере оборудования и программного обеспечения. Поэтому есть только один способ сделать задачу переносимой и заключается он в максимальном использовании данных, приведенных в табл. 1.2. После выполнения примеров 2.1—2.3 эти данные расположены в массиве `Info`. Значение сегмента массива находится в регистре `fs`, а адрес начала массива (смещение от начала сегмента) равен нулю и расположен в регистре `di`.

Для сохранения и последующего использования служебной информации в разделе данных задачи надо зарезервировать переменные требуемого размера и присвоить им соответствующие имена. Например, размер рабочего поля экрана по вертикали мы будем хранить в переменной `Versize`, а по горизонтали — в переменной `Horsize`. Подробнее о резервировании и назначении переменных сказано в разделе 2.6 данной главы. Описанные там и в примерах всей главы имена переменных будут неоднократно использоваться в тексте книги.

Значения Vbuff, Horsize и Versize. В примере 2.5 показано, как можно сохранять данные из массива `Info`. Предполагается, что он выполняется сразу после примера 2.4, о чем свидетельствует метка `succ`.

Пример 2.5. Определение значений трех переменных

```
succ: mov  ax, fs:[di+08] ; читаем значение видеосегмента
      mov  Vbuff, ax      ; и сохраняем его в Vbuff
      mov  ax, fs:[di+12h]; читаем количество точек в строке
      mov  horsize, ax    ; и сохраняем его в horsize
      mov  ax, fs:[di+14h]; читаем количество строк на экране
      mov  versize, ax    ; и сохраняем его в versize
```

Команда пересылки не может перемещать данные из одного места памяти в другое, в подобных случаях нужны две команды и регистр, используемый в качестве посредника. В примере 2.5 посредником является регистр `ax`, но можно применять любой из регистров общего назначения. При чтении в регистр `ax` использован индексный способ адресации второго операнда с указанием смещения в виде числа. Макроассемблер включает смещение в код формируемой команды. Конкретное значение адреса операнда вычисляет микропроцессор при выполнении команды. Для этого он суммирует текущее содержимое регистра `di` и указанное в коде команды смещение.

Напомним, что в слове массива `Info` со смещением `10h` находится размер отображаемой на экране строки (`Scan Line`) в байтах. Если эта величина будет использоваться в задаче, то ее также надо сохранить. В примере 2.11 для этого выделена переменная `Bperline`.

Переменные для работы с окнами. Массив `Info` содержит три величины, значения которых используются при работе с окнами видеопамяти. В примере 2.6, который является продолжением примера 2.5, показан способ сохранения этих величин и вычисления единицы приращения значения окна.

Пример 2.6. Сохранение величин, используемых при работе с окнами

```
mov  al, fs:[di+3]          ; читаем состояние окна B
mov  winB, al               ; и сохраняем его в байте winB
mov  cl, es:[di+4]          ; читаем в cl window granularity
mov  ax, 64                 ; помещаем в ax число 64
div  cl                     ; деление ax = ax/cl
mov  GrUnit, ax             ; сохраняем результат деления
mov  eax, es:[di+0Ch]       ; читаем в eax адрес подпрограммы
mov  VMC, eax               ; и сохраняем его в VMC
```

Две первые команды примера 2.6 сохраняют в памяти содержимое байта состояния окна `B`. Эта величина нужна, если видеокарта поддерживает работу с двумя окнами. Она может принимать одно из четырех значений:

- 0 — второе окно не существует;
- 3 — окно используется только при чтении из видеопамяти;

- 5 — окно используется только при записи в видеопамять;
- 7 — окно используется при записи и при чтении.

Далее в примере 2.6 команды 3—6 вычисляют единицу для приращения (увеличения или уменьшения) значений окон видеопамяти делением числа 64 на содержимое байта массива `Info` со смещением 4. Такой способ вычисления `GrUnit` рекомендован стандартом VESA.

Две последние команды примера 2.6 сохраняют в области данных задачи адрес подпрограммы `BIOS`, выполняющей работу с видеоокнами. Адрес состоит из сегмента и смещения, поэтому при его пересылке в качестве посредника используется 32-разрядный регистр `eax`.

Имена `GrUnit` и `VMC` заимствованы из описания стандарта VESA, их расшифровка приводится в следующем разделе данной главы.

В байте массива `Info`, имеющем смещение 6, хранится размер окна, выраженный в килобайтах. Его стандартным значением является код `40h`, соответствующий 64 Кбайт. Для семейства IBM PC стандартным значением сегмента оперативной памяти является именно 64 Кбайт. Возможно, по этой причине автор не встречал видеокарт, имеющих окна аномального размера.

В данном разделе мы показали способ получения и сохранения только той информации о видеокарте, которая будет многократно использоваться в примерах, приведенных в книге. Об использовании других величин, содержащихся в массиве `Info`, речь пойдет по мере изложения материала.

Советуем вам составить программу, которая позволяет ввести код видеорежима, прочитать в память массив `Info` и вывести на экран или печать содержимое его первых 20 слов в шестнадцатеричном коде. Затраченные на ее составление и отладку усилия окупятся при дальнейшей работе.

2.3. Процедуры для работы с одним окном видеопамяти

На видеокарте обязательно расположена оперативная память, которую принято называть видеопамятью (`video memory`). Видеоконтроллер непрерывно выводит содержимое части видеопамяти на экран монитора, причем размер этой части зависит от установленного видеорежима. На современных видеокартах базовый объем памяти составляет не менее 1 Мбайт (1 Мбайт равен 1 048 576 байтам) и может быть расширен, по крайней мере, до 4 Мбайт при установке на видеокарту дополнительных микросхем. Напомним, что для работы в режиме VESA `11Bh` четырех мегабайтов недостаточно. У акселераторов объем видеопамяти существенно больше (до 64 Мбайт).

Доступ к видеопамяти. Специфической особенностью семейства IBM PC является ограничение пространства доступных адресов размером 1 Мбайт.

Оно делится на сегменты, предельный размер которых составляет 64 Кбайт (1 Кбайт равен 1024 байтам). Всего в пространстве адресов помещается 16 сегментов предельного размера, 10 из них занимает оперативная память ПК.

Для доступа к видеопамяти выделяется один сегмент размером в 64 Кбайт, который чаще всего имеет адрес `A000h` для графических и `B800h` для текстовых режимов. Только при указании этих сегментов графическая или текстовая информация будет направлена в видеопамять или считана из нее, поэтому их значения ни при каких условиях не должны изменяться задачей. Размер видеосегмента значительно меньше реального объема видеопамяти. Для того чтобы задачи могли работать со всей памятью, в современных видеоконтроллерах реализован следующий механизм.

Все пространство видеопамяти делится на сегменты размером по 64 Кбайт, которые пронумерованы начиная с нуля. По принятой терминологии такие сегменты называют окнами или видеоокнами. В специальном регистре видеоконтроллера хранится номер текущего окна. При обращениях к видеопамяти контроллер добавляет его к 16-разрядному адресу, указанному задачей, и получает абсолютный адрес. Количество разрядов в абсолютном адресе зависит от предельного объема памяти, которая может быть установлена на видеокарте. Если на видеокарте может быть установлено 2 Мбайт памяти, то адрес занимает 21 разряд, если установлено 4 Мбайт — то 22 разряда и т. д.

Таким образом, полный адрес видеопамяти складывается из двух частей. Младшая часть является относительным адресом (смещением в сегменте), а старшая часть — номером текущего окна, хранящимся в одном из регистров видеоконтроллера. Задача может изменять текущее окно с помощью функции `4F05h`.

Прямое обращение к BIOS. Видеокарты различаются не только адресами регистров, в которых хранится номер текущего окна, но и тем, в каких разрядах этих регистров он располагается. Поэтому при описании функции `4F05h` в документации специально оговорено, что номер окна выражается в единицах приращения его значений, дословно `in granularity unit`. В примере 2.6 показан способ вычисления единицы приращения, она хранится в переменной `GrUnit`. Обычно она равна 1, только для одной видеокарты получилось другое значение, но если есть исключения, то надо следовать рекомендациям VESA и использовать переменную `GrUnit`.

Для установки окна с помощью функции `4F05h` его номер помещается в регистр `dx`, а регистр `bx` очищается. После этого в регистр `ax` записывается код функции `4F05h` и выполняется команда `int 10h`. В примере 2.7 показан способ прямого обращения к `BIOS` для установки окна. В нем и во всех последующих примерах текущий номер окна, выраженный в единицах `GrUnit`, выбирается из переменной `Cur_win`. Она имеет размер слова и располагается в разделе данных задачи (см. пример 2.11).

Пример 2.7. Установка окна с использованием функции BIOS

```
mov dx, Cur_win        ; запись в dx значения окна
xor bx, bx             ; признак установки окна
mov ax, 4F05h          ; код функции BIOS
int 10h                ; обращение к BIOS
```

Ускорение работы с окнами. В описании стандарта VESA не рекомендуется использовать прямое обращение к `BIOS` для установки окна. Причина простая — при обращениях к `BIOS` с использованием прерываний, например `int 10h`, выполняется много вспомогательных действий, связанных с сохранением регистров и расшифровкой кода запроса. Специфической особенностью прерывания `int 10h` является то, что на обращения к нему реагируют компоненты DOS и некоторые резидентные (постоянно находящиеся в оперативной памяти) задачи, например русификатор KEYRUS. В результате количество дополнительных и ненужных в данном конкретном случае действий оказывается значительно больше количества полезных действий, выполняющих запись или чтение окна. По этой причине разработчики стандарта VESA рекомендуют использовать только процедуру `BIOS`, которая выполняет чтение или запись окна и состоит всего из 10—15 команд. В описании стандарта процедура называется `Video Memory Control` (`VMC`). Способ сохранения ее адреса показан в примере 2.6.

При вызове процедуры `VMC` для чтения или установки окна регистры `bx` и `dx` заполняются так же, как и при обращении к функции `4F05h`. Прежде чем рассматривать конкретные примеры, мы обсудим несколько вопросов, имеющих отношения к работе с окнами.

Описание гнезда подпрограмм. Установка или изменение текущего окна производится в задачах при построении, перемещении или копировании рисунков, управлении курсором, выводе и редактировании текста, т. е. при любых манипуляциях с графическими объектами. Поэтому вызов процедуры `VMC` целесообразно оформить в виде подпрограмм, расположенных в теле основной задачи.

Опыт показывает, что при работе с графикой номер нового окна чаще всего отличается от текущего значения на единицу. Поэтому целесообразно составить гнездо подпрограмм, которые производят установку не только указанного, но и следующего или предыдущего окна.

Номер текущего окна бывает нужен сравнительно часто, поэтому его надо хранить в рабочей области памяти в переменной `Cur_win`, имеющей размер слова. Конкретное значение этой переменной зависит от выполняемых задачей действий. Обычно в текущем окне располагаются графический курсор, обрабатываемая часть рисунка или редактируемый текст. В некоторых случаях текущее окно может соответствовать области видеопамяти, не отображаемой в данный момент на экране.

В примере 2.8 описаны три подпрограммы, предназначенные для установки значения окна путем вызова процедуры `VMC`. Подпрограммы оформлены как внутренние, поэтому они должны быть расположены в том же сегменте памяти, в котором находится основная программа. Обращения к ним производятся с помощью команды `call`, в которой указано имя (метка) подпрограммы. Входным параметром для всех трех подпрограмм является переменная `Cur_win`, исходное значение которой изменяется при установке следующего или предшествующего окна. Так и должно быть, поскольку новое окно становится текущим. Подпрограммы не изменяют содержимое регистров общего назначения, с которыми они работают.

Подпрограммы SetWin, NxtWin и PrevWin. Ядром примера 2.8 является подпрограмма `SetWin` (вызов `call SetWin`). Она помещает текущее окно в регистр `dx`, очищает регистр `bx` и вызывает процедуру `VMC` для установки указанного окна. Прерывание `int 10h` не используется, что соответствует рекомендациям стандарта VESA.

Подпрограмма `NxtWin` (обращение `call NxtWin`) устанавливает следующее окно. При ее выполнении текущее значение переменной `Cur_win` увеличивается на единицу приращения (`GrUnit`), а затем выполняется `SetWin`.

Подпрограмма `PrevWin` (обращение `call PrevWin`) устанавливает предыдущее окно. При ее выполнении текущее значение переменной `Cur_win` уменьшается на единицу приращения (`GrUnit`), а затем вызывается `SetWin`.

Пример 2.8. Три подпрограммы для работы с видеоокнами

```
        ; Установка следующего окна
NxtWin: push  ax            ; сохраняем содержимое ax
        mov   ax, GrUnit    ; читаем единицу приращения окна
        add   Cur_win, ax   ; увеличиваем номер окна
        pop   ax            ; восстанавливаем содержимое ax
        ; Установка окна, указанного в Cur_win
SetWin: PushReg <ax,bx,dx>  ; сохранение содержимого регистров
        xor   bx, bx        ; признак установки окна
        mov   dx, Cur_win   ; номер устанавливаемого окна
        call  [VMC]         ; обращение к подпрограмме BIOS
        PopReg <dx,bx,ax>   ; восстановление содержимого регистров
        ret                 ; возврат из подпрограммы
        ; Установка предыдущего окна
PrevWin: push  ax           ; сохранение содержимого ax
        mov   ax, GrUnit    ; читаем единицу приращения окна
        sub   Cur_win, ax   ; уменьшаем номер окна
        pop   ax            ; восстанавливаем содержимое ax
        jmp   SHORT SetWin  ; переходим на установку окна
```

Процедура VMC расположена в "удаленном" сегменте, т. е. в пространстве адресов, не принадлежащих задаче. В таких случаях для обращения к подпрограммам используется команда call, у которой операнд является двойным словом, содержащим полный адрес (сегмент и смещение). При компиляции Макроассемблер формирует специальный код, указывающий процессору, что переход производится на удаленный адрес. В примере 2.8 операндом команды call является переменная VMC, описанная как двойное слово. Заключение имени переменной в квадратные скобки указывает на то, что адресом процедуры является не VMC, а хранящееся в ней значение, которое было установлено при выполнении команд примера 2.6.

Указание типа Short в команде jmp заставит Макроассемблер сформировать короткую команду (для перехода не более чем на 128 байтов), код которой занимает два байта. PushReg и PopReg — это макросы (макровызовы). Первый эквивалентен трем командам push ax, push bx и push dx, а второй — трем командам pop dx, pop bx, pop ax. В реальной программе вы должны либо заменить макросы указанными командами, либо поместить в начале текста программы соответствующие им макроопределения, текст которых приведен в примере 2.12. Сохранение исходного содержимого регистров в стеке и восстановление при выходе делается для того, чтобы находящиеся там данные не изменялись в результате выполнения подпрограммы.

Чтение текущего окна. Еще раз вернемся к текущему окну. При корректной работе задачи с окнами значение переменной Cur_win всегда соответствует номеру, находящемуся в регистре видеоконтроллера. Если по каким-то причинам такое соответствие нарушено, то текущее значение окна можно восстановить, прочитав его из видеоконтроллера. Оформлять чтение текущего окна в виде самостоятельной процедуры не целесообразно, поскольку регулярных обращений к ней не должно быть.

Для чтения окна с помощью функции 4F05h надо выполнить следующие три команды:

```
mov  bx, 100h          ; признак чтения окна
mov  ax, 4F05h         ; код запрашиваемой функции
int  10h               ; обращение к BIOS
```

Другой способ чтения текущего значения окна заключается в прямом обращении к процедуре BIOS, а именно:

```
mov  bx, 100h          ; признак чтения окна
call [VMC]             ; обращение к процедуре BIOS
```

В обоих случаях номер текущего окна возвращается в регистре dx. Каким из двух способов лучше воспользоваться, решает программист. Мы привели эти примеры для того, чтобы еще раз проиллюстрировать различие между прерыванием и прямым обращением к процедуре.

Контроль ошибок. В данном разделе ничего не говорилось о контроле ошибок при установке или изменении окон видеопамяти. Это объясняется несколькими причинами. Прежде всего, признак ошибки функция `4F05h` возвращает только в том случае, если содержимое регистра `bh` больше единицы, т. е. если неправильно указано запрашиваемое действие. Контроль правильности указанного значения окна не производится, поскольку его не так просто осуществить.

Если задача проверила поддержку установленного видеорежима, т. к. это рекомендовалось в предыдущем разделе, то при корректно организованной работе с видеопамятью значение устанавливаемого окна не может выходить за допустимые пределы. При организации любого контроля над ходом вычислительного процесса надо, прежде всего, руководствоваться соображениями здравого смысла, иначе можно дойти до проверки результатов каждого выполняемого действия.

В рассматриваемом случае возможен единственный способ контроля — проверка соответствия номера окна реально существующему объему видеопамяти. Но выполнять такую проверку при каждом изменении номера окна едва ли целесообразно. В крайнем случае, ее можно временно на период отладки задачи включить в подпрограмму `SetWin` и убрать после отладки. Но лучше тщательно продумать алгоритм работы задачи и проверить правильность его воплощения.

Если вы умеете работать с отладчиками и восстанавливать по кодам команд исходный текст на языке ассемблера, то имеет смысл разобраться в том, как воплощена процедура `VMC` в `BIOS`. Это позволит вам узнать, как выполняется чтение или установка номера окна на уровне работы с портами ввода-вывода.

2.4. Работа с двумя окнами видеопамяти

Большинство исследованных автором видеокарт поддерживало работу только с одним окном `A`. Исключением явился акселератор `mach64` фирмы ATI Technologies Inc., у которого для доступа к видеопамяти используется два окна. Запись данных в видеопамять осуществляется через окно `A`, а чтение — через окно `B`. Оба окна отображены на один видеосегмент `A000h`. При обращениях к видеопамяти видеоконтроллер самостоятельно выбирает нужное окно, в зависимости от того записываются данные или считываются. Программисту остается "только" следить за тем, какое из двух окон надо переключать при достижении границы сегмента. Но это "только" может стать серьезным камнем преткновения для неискушенного программиста.

Стандартом VESA предусмотрена возможность работы с двумя окнами `A` и `B`, о чем говорилось при описании функции `BIOS 4F05h`. Для работы с окном `A` (его установки или чтения) регистр `bl` (младший байт регистра `bx`) должен

быть очищен, а для работы с окном B в него записывается единица. Если существует только одно окно, то оно имеет имя A и доступно для записи и чтения. Для проверки количества окон достаточно проанализировать состояние окна B, т. к. A заведомо существует. Если окно B доступно только для чтения, то A будет доступно лишь для записи или наоборот. Оба окна не могут быть доступны для выполнения одной и той же операции с видеопамятью, поскольку в таком случае придется вводить специальный механизм выбора нужного окна, а он стандартом VESA не предусмотрен.

При поддержке видеокартой двух окон возможны два способа работы с ними. Первый способ (в описании стандарта VESA он называется overlapping windows) основан на одновременном (синхронном) изменении номеров обоих окон. Второй (в описании стандарта VESA он называется distinguished windows) — на независимом изменении номеров окон (различение окон).

Перекрытие окон. При программировании большинства графических алгоритмов удобнее иметь дело с одним окном видеопамяти, независимо от того, сколько их есть на самом деле. Для этого надо изменить описанное в примере 2.8 гнездо подпрограмм так, чтобы при их вызове всегда изменялось столько окон видеопамяти, сколько их поддерживает видеокарта.

Изменения, которые надо внести только в подпрограмму SetWin, поскольку только она обращается к процедуре VMC, заключаются в следующем. Окно A существует в любом случае, поэтому подпрограмма, в первую очередь, должна установить его номер. После этого надо проверить существование второго окна. Если его нет, то установка завершена. Если окно B существует, то ему надо присвоить тот же номер, который был присвоен окну A.

Напомним, что в примере 2.6 значение байта, отражающего состояние второго окна, копируется в переменную WinB. При описании примера 2.6 было сказано, что содержимое WinB может иметь только 4 значения: 0, 3, 5 или 7. Если оно равно нулю, то второго окна не существует. Три остальных значения указывают, для каких действий доступно второе окно. В данном случае для нас важен сам факт существования окна, а не то, для чего оно предназначено. Поэтому надо просто проверить состояние нулевого разряда байта WinB, если он очищен, то окно не существует, а если установлен, то существует.

Подпрограмма SetWin для двух окон. Текст измененной подпрограммы SetWin приведен в примере 2.9. Его надо включить в пример 2.8 вместо описанного там варианта SetWin.

Пример 2.9. Процедура установки одного или двух окон

```
SetWin:  PushReg <ax,bx,dx>  ; сохранение используемых регистров
         xor   bx, bx        ; признак установки окна A
         mov   dx, Cur_win   ; номер устанавливаемого окна
```

```
        call  [VMC]          ; установка окна A
        test  winB, 01       ; окно B существует ?
        je    stw            ; -> нет, переход на метку stw
        mov   bx, 01         ; признак установки окна B
        call  [VMC]          ; установка окна B
stw:    PopReg <dx,bx,ax>    ; восстановление регистров
        ret                  ; возврат из подпрограммы
```

В дальнейшем, при описании примеров работы с графикой мы будем считать, что выполнение подпрограмм `SetWin`, `NxtWin` и `PrevWin` не зависит от количества окон видеопамяти, поддерживаемых видеокартой, а при наличии двух окон их номера изменяются одновременно.

Раздельная работа с окнами. Раздельная (независимая) работа с двумя окнами применяется в редких случаях. Если в вашей практике возникнет такой случай, то следует поступить так:

- выяснить, какое окно доступно для записи, а какое для чтения;
- ввести две разные переменные для хранения текущих значений окон;
- использовать два разных указателя адреса в текущем окне;
- составить независимые подпрограммы для работы с двумя окнами;

Обсудим эти вопросы более подробно.

При составлении конкретных программ или подпрограмм надо знать, какое окно доступно для записи, а какое для чтения. Например, у акселератора `mach64` фирмы ATI Technologies Inc. для записи доступно окно `A`, а для чтения — `B`. Но это не общее правило. Стандарт VESA не распространяется на назначение окон, он только требует, чтобы оно было отражено в байте состояния окна. Поэтому наиболее простой способ выяснения назначения окон заключается в визуальной проверке кодов байтов состояния обоих окон, делать это программно намного сложнее.

В процессе выполнения задачи текущие значения окон изменяются независимо, поэтому их надо хранить в разных переменных. Учитывая, что для изменения номеров окон будут использоваться разные подпрограммы, можно выделить в разделе данных задачи две переменные `Cur_winA` и `Cur_winB`. В таком случае при вызове подпрограмм не надо будет затрачивать лишние действия на передачу параметров.

Раздельная работа с окнами подразумевает и раздельную работу с адресами расположенных в них точек. При программировании циклов построения или преобразования графических объектов адреса точек хранятся в указателях адресов, обычно это индексные регистры. Количество используемых указателей адресов зависит от реализуемого алгоритма. Если текущие значения адресов в обоих окнах совпадают, то достаточно одного указателя адреса, в противном случае их должно быть два, что неизбежно усложнит структуру подпрограммы, выполняющей нужные действия.

Два комплекта подпрограмм. Мы не будем приводить подпрограммы, а просто опишем, что необходимо сделать.

Прежде всего, надо решить вопрос об именах подпрограмм `SetWin`, `NxtWin` и `PrevWin` и имени переменной `Cur_win`. Проще всего добавить в конце каждого из этих имен буквы `A` и `B`, например, `Cur_winA` и `Cur_winB`.

Для получения двух комплектов подпрограмм сделайте копию примера 2.8 с тем вариантом `SetWin`, который в нем описан. В тексте оригинала добавьте в конце имен точек входа и в конце имени `Cur_win` букву `A`.

В тексте копии добавьте в конце имен точек входа и в конце имени `Cur_win` букву `B`. Кроме того, в тексте копии надо заменить в подпрограмме `SetWinB` команду `xor bx, bx` командой `mov bx, 01`. Напомним, что наличие в регистре `bx` кода 1 является признаком работы с окном `B`. Это все, что надо сделать для получения двух комплектов подпрограмм.

Таковы основные особенности работы с двумя независимыми окнами. Еще раз напомним, что в практике автора только акселератор `mach64` имел два разных окна для записи и чтения. Учитывая, что карта выпущена в 1997 году, можно допустить, что появятся и другие видеокарты с раздельными окнами для записи и чтения в видеопамять.

2.5. Страничная организация видеопамяти

Одним из традиционных приемов при работе с видеопамятью является ее деление на страницы. Из нескольких страниц только одна отображается на экране монитора, а остальные не видны. Видимую (отображаемую на экране) страницу называют активной, а невидимые — пассивными. Изменение содержимого невидимых страниц никак не отражается на экране монитора. Поэтому можно заранее подготовить и расположить на пассивной странице нужное изображение, а затем "мгновенно" изменить картинку на экране, сделав эту страницу активной.

Возможность деления видеопамяти на страницы основана на том, что при взаимодействии с монитором видеоконтроллер отображает только ту ее часть, которая нужна для заполнения рабочей области экрана. Остальное пространство видеопамяти просто не используется. Размер рабочего пространства видеопамяти зависит от установленного видеорежима и изменяется в достаточно больших пределах (см. табл. 1.1). Соответственно изменяется и размер свободного пространства. В одних случаях оно может быть намного больше рабочей части видеопамяти, а в других его может просто не быть. Поэтому возможность и целесообразность деления видеопамяти на страницы решается с учетом ее реального объема и используемого в задаче видеорежима.

Стандарт VESA не распространяется на страничную организацию видеопамяти и, вообще, его авторы предпочитают говорить не о страницах, а о раз-

ных изображениях, расположенных в видеопамяти. Поэтому все практические вопросы решаются по усмотрению программиста. Именно он выбирает способ переключения и расположение страниц в видеопамяти.

Смена активной страницы. Как уже говорилось, активной является та страница, содержимое которой в данный момент отображается на экране монитора. Специальный механизм переключения страниц отсутствует, но существует функция VBE с кодом 4F07h (см. раздел 1.2.2), которая позволяет переместить начало рабочей области (Display Start) в любую точку видеопамяти. Есть только одно ограничение на ее применение — от выбранной точки до конца видеопамяти должно оставаться пространство, достаточное для размещения рабочей области. Координаты выбранной точки указываются в виде строки и столбца.

Перед вызовом этой функции в регистр dx помещается порядковый номер первой отображаемой строки, а в регистр cx — номер ее первой точки. Напомним, что все номера начинаются с нуля. Кроме этого регистр bx очищается, что является признаком изменения начала отображаемой области, а в регистр ax помещается код функции 4F07h. После чего производится обращение к BIOS. Описанные действия выполняет группа команд примера 2.10.

Пример 2.10. Установка нового начала отображаемой области памяти

```
xor     bx, bx          ; признак смены страницы
mov     cx, BaseCol     ; номер точки в исходной строке
mov     dx, BaseRow     ; номер исходной строки
mov     ax, 4F07h       ; код запрашиваемой функции
int     10h             ; обращение к BIOS
```

В примере 2.10 значения координат начала рабочей области выбираются из переменных BaseCol и BaseRow, которые должны быть описаны в сегменте данных программы. Для превращения этого примера в подпрограмму переключения активных страниц к нему надо добавить вычисление значений указанных переменных по номеру страницы. Способ вычисления выбирается по усмотрению программиста, а зависит он от выбранного расположения страниц в видеопамяти.

При работе в режимах packed pixel graphics, использующих регистры палитры, перед переключением страниц может понадобиться сохранение в оперативной памяти текущей палитры и установка новой. Способы работы с палитрой цветов описаны в главе 4. При работе в режимах Hi-Color и TrueColor палитра не используется, т. к. цвет каждой точки указан в ее коде.

Изменения в вычислениях адресов. Страничная организация памяти влияет только на способы определения адресов точек графических объектов. Поэтому желательно выбрать такое расположение страниц, при котором изме-

нение работы с адресами точек будет минимально возможным. Иначе говоря, нас интересует такое расположение страниц в видеопамяти, при котором основные процедуры рисования, построения или преобразования графических объектов почти не изменяются.

Перед построением или изменением графического объекта обычно выбирается некая опорная точка, относительно которой вычисляются адреса всех остальных точек объекта. Чаще всего это левый верхний угол прямоугольной области экрана, в которой располагается или будет расположен объект. Координаты опорной точки обычно задают в виде номера строки и столбца, на пересечении которых она находится. Но для дальнейшей работы их надо преобразовать в адрес видеопамяти.

Если видеопамять не разделена на страницы, то начало ее рабочего пространства находится на пересечении нулевой строки и нулевого столбца. В таком случае существует простая связь между координатами произвольной точки и ее адресом в видеопамяти. Адрес вычисляется как сумма двух произведений: номера строки на размер строки в байтах и номера столбца на размер кода точки в байтах. Здесь имеется в виду размер строки, отображаемой на экране (`Scan Line`). Способы вычисления адресов описаны в разделах 3.1.3 (режимы `packed pixel graphics`) и 7.2 (режимы `direct color`), а соответствующие подпрограммы приведены в примерах 3.4 и 7.3.

При делении видеопамяти на страницы вычисленные адреса точек становятся относительными. Для получения абсолютных значений адресов в этом случае при вычислениях надо учитывать адрес начала страницы или ее координаты (номера нулевой строки и нулевого столбца). Следовательно, при введении страничной организации памяти придется изменить подпрограммы, выполняющие вычисление адресов точек по их координатам. Но это не все.

В процессе работы с графическим объектом приходится вычислять адреса начала его строк. В примерах, приведенных в данной книге, для этой цели использовалась "константа переадресации", значением которой является разность между размером отображаемой на экране строки (`Scan Line`) и шириной графического объекта, выраженной в байтах. Обе величины зависят от установленного задачей видеорежима, способ учета этой зависимости описан в разделе 7.3.3, а варианты соответствующих подпрограмм приведены в примере 7.3. При их составлении предполагалось, что отображаемые на экране строки начинаются с нулевого столбца.

Следовательно, если страницы расположить так, чтобы каждая из них начиналась с начала одной из строк видеопамяти, то значение константы переадресации строк не будет зависеть от номера страницы. При таком расположении страниц изменяются только подпрограммы, вычисляющие адреса точек по их координатам.

Расположение и размеры страниц. Если размер страниц равен размеру отображаемой на экране (рабочей) части видеопамяти, то каждая из них будет

начинаться с новой строки. При таких размерах страницы располагаются в видеопамяти подряд друг за другом.

Если предполагается использовать `N` страниц, то в оперативной памяти надо выделить массив размером `2N` слов. При выполнении подготовительных действий в эти слова задача должна поместить номера окон, в которых начинаются страницы и адреса нулевых строк страниц в этих окнах. Эти величины нужны для подпрограммы, выполняющей преобразование значений координат в адреса точек. Учитывая, что реальное количество страниц невелико, такой массив можно хранить в разделе данных задачи.

Оценим возможное количество страниц. Предположим, что установленный на видеокарте объем памяти составляет 4 Мбайт и используется видеорежим с разрешением 640×480 точек. Если это режим `packed pixel graphics`, то в видеопамяти помещается 13 страниц, в режиме `Hi-Color` их количество сократится в 2 раза, а в режиме `True Color` — в 4 раза. При более высоком геометрическом разрешении количество страниц сокращается. Поэтому можно считать, что оно не больше десяти.

Для определения допустимого количества страниц при выполнении задачи надо прочитать байт массива `Info` со смещением `1Dh`. В нем хранится номер последней страницы, которую можно установить в конкретном видеорежиме при имеющемся объеме видеопамяти. Страницы нумеруют начиная с нуля, поэтому их количество на 1 больше числа, хранящегося в байте `1Dh`.

Кроме указанного массива, в разделе данных задачи надо выделить две переменные, например, `Base_win` и `Base_addr` для хранения исходного окна и адреса текущей страницы, с которой работает задача. В исходном состоянии эти переменные очищены, поскольку обычно работа начинается с нулевой страницы. В дальнейшем задача изменяет их значения в зависимости от номера используемой страницы.

Страница в начале окна. Особый интерес представляет случай, когда каждая страница начинается с нового окна видеопамяти. При этом базовые адреса равны нулю, и в оперативной памяти надо хранить только номера окон, с которых начинаются страницы. Кроме того, в этом случае значения абсолютных и относительных адресов отличаются только на номер окна. Поэтому после вычисления относительного адреса его надо просто увеличить на номер окна. Как это делается, показано в примерах 3.4 и 7.3.

Рекомендованные стандартом VESA видеорежимы перечислены в табл. 1.1. Если исключить из рассмотрения текстовые и 16-цветные, то остается 20 режимов, которые по геометрической разрешающей способности делятся на 5 групп. В каждую группу входит 4 режима, различающихся по количеству цветов.

При работе с тремя группами видеорежимов начала страниц можно совместить с началом окон видеопамяти. В табл. 2.1 приведены размеры страниц для этих групп режимов.

Таблица 2.1. *Размеры страниц для трех групп видеорежимов*

Разрешающая способность в точках	Количество окон (k = 1, 2, 4)	Количество строк
640×480	5·k	512
1024×768	12·k	768
1280×1024	20·k	1024

Во втором столбце табл. 2.1 буква `k` указывает количество байтов, которое занимает код точки. Это косвенная характеристика цветовой палитры. 1 байт — 256 цветов (`packed pixel graphics`), 2 байта — 32K или 64K цветов (`Hi-Color`), 4 байта — 16M цветов (`True Color`).

При разрешении 1024×768 и 1280×1024 точки размер страницы совпадает с размером рабочей области видеопамяти. При разрешении 640×480 точек размер страницы на 32 строки больше размера рабочей области, т. е. по сравнению с последовательным расположением страниц теряется некоторое пространство видеопамяти, но это не имеет принципиального значения.

Из двух групп, не попавших в табл. 2.1, практический интерес представляют видеорежимы с разрешением 800×600 точек. Целое количество строк, содержащих 800 точек, укладывается в 25 окнах, что почти в три раза превышает размер рабочей области видеопамяти. В этом случае при совмещении начала страниц с началом окон будет потеряно большое пространство видеопамяти. Поэтому при разрешении 800×600 точек страницы лучше располагать в видеопамяти последовательно друг за другом.

Заключение. Использование страниц видеопамяти расширяет возможности работы с графикой только при решении определенного класса задач. Например, страницы видеопамяти применяются всеми текстовыми и графическими редакторами. В некоторых источниках встречаются указания об использовании переключения страниц для получения спецэффектов, основанных на быстром изменении картинки на экране. Однако не надо забывать, что требуется определенное время на создание нужного изображения на пассивных страницах. Поэтому вопрос о целесообразности введения страниц видеопамяти и способах работы с ними надо решать, учитывая особенности конкретной задачи.

2.6. Часто используемые в примерах имена

В примерах, приводимых в данной и последующих главах книги, многократно повторяются имена переменных, подпрограмм для работы с окнами видеопамяти и макроопределений, предназначенных для записи в стек или

выталкивания из него содержимого регистров. Для того чтобы каждый раз не объяснять назначение и способы описания и определения этих имен, мы сделаем это в данном разделе.

Подпрограммы для работы с окнами. Текст подпрограмм, выполняющих установку текущего (`Setwin`), следующего (`Nxtwin`) и предыдущего (`Prevwin`) окна, приведен в примере 2.8. Здесь мы обратим ваше внимание на следующие особенности, связанные с их использованием в примерах.

Прежде всего, обычно Макроассемблер не различает заглавные и строчные буквы в именах, независимо от их назначения. Поэтому, например, имена `SetWin`, `Setwin` и `setwin` для него тождественны. Буквы разного размера используются для того, чтобы человек мог визуально различить отдельные части составных имен. Если же вы принудительно заставите Макроассемблер различать строчные и заглавные буквы, что вполне возможно, то придется использовать только одну форму записи имен.

Подпрограммы должны быть включены в текст основной программы, для того чтобы Макроассемблер мог опознать их имена и сформировать команды вызова. Возможны разные способы оформления текстов подпрограмм и их размещения в теле задачи, подробно эти вопросы рассмотрены в приложении В. В примерах основной части книги описаны подпрограммы, расположенные в кодовом сегменте, т. е. там, где находятся команды, образующие тело задачи. Обычно в исходном тексте он имеет имя `Code`. Место расположения подпрограмм в пределах кодового сегмента выбирается по усмотрению разработчика.

Основные переменные. Переменная — это последовательность байтов оперативной памяти, которым присвоено уникальное имя и в которых хранятся величины, применяемые при вычислениях. В командах обычно используются переменные, содержащие 1, 2 или 4 байта. Переменные, содержащие большее количество байтов, принято называть строками, массивами или таблицами.

Переменные обычно хранятся в отдельном сегменте оперативной памяти, который принято называть сегментом данных. В исходных текстах программ его наиболее распространенным именем является `Data`. Для большей наглядности рекомендуется располагать сегмент данных перед кодовым сегментом. В примере 2.11 показан вариант оформления сегмента данных и описания в нем основных переменных, используемых в последующих примерах. Способ определения (формирования значений) большинства из них был описан в данной главе (см. примеры 2.1—2.6).

Пример 2.11. Описания основных переменных в сегменте данных

```
Data    SEGMENT         ; директива указывает начало сегмента
OldMode db 0            ; исходный видеорежим
```

```
NevMode  dw 101h        ; видеорежим VESA 101h
Vbuff    dw 0A000h      ; адрес видеобуфера
Horsize  dw 640         ; количество точек в строке
Bperline dw 640         ; количество байтов в строке
Versize  dw 480         ; количество строк на экране
Cur_win  dw 0           ; номер текущего окна
Cur_pos  dw 0           ; адрес (смещение) в текущем окне
GrUnit   dw 0           ; единица приращения номера окна
VMC      dd 0           ; адрес процедуры BIOS
winB     db 0           ; параметры окна B
         ; Далее до конца сегмента располагаются другие описания
Data     ENDS           ; директива указывает конец сегмента
```

Описание сегмента открывает директива `SEGMENT`, а закрывает директива `ENDS`. Перед обеими директивами указывается одно и то же имя, в данном случае `Data`. Назначение и способы оформления сегментов описаны в приложении Б данной книги.

В примере 2.11 описаны 11 основных переменных. Каждая из них имеет уникальное имя. После имени расположены директивы `db`, `dw` или `dd`, указывающие тип переменной, т. е. ее размер в байтах: `db` — байт (8 разрядов), `dw` — слово (16 разрядов), `dd` — двойное слово (32 разряда).

После директивы указывается значение, которое Макроассемблер присваивает переменной. В примере 2.11 переменным сразу присвоены конкретные значения, но на практике они формируются в процессе выполнения задачи. Например, код сегмента видеобуфера (значение переменной `Vbuff`) обычно `A000h`, но возможны исключения, поэтому его значение надо выбирать из массива `Info`.

В конце каждой строки примера 2.11 расположен комментарий, поясняющий назначение переменных и директив. Признаком начала комментария является символ "точка с запятой". Обнаружив его, Макроассемблер просто пропускает весь текст до конца строки.

Обратите внимание, в тексте примера 2.11 после всех имен отсутствует символ "двоеточие". Существует простое правило: двоеточие должно указываться только после имен меток, расположенных перед командами.

Следует отметить, что после директив описания типа может указываться не одно, а несколько значений, при этом Макроассемблер размещает эти значения в последовательно расположенных байтах, словах или двойных словах. Имя переменной в таком случае относится только к первому байту, слову или двойному слову. В отдельных случаях оно может вообще отсутствовать (см. пример 6.3). Существует специальный оператор повторения, который позволяет связать с именем переменной требуемое количество байтов, например:

```
Buffpal  dd 256 DUP (0)      ; резервирование и очистка памяти
```

В этом примере имя `buffpal` соответствует буферу размером в 256 двойных слов, содержимое которых принудительно очищается.

И последнее замечание, директива `db` может использоваться для описания текстовых строк, предназначенных для вывода на экран подсказок, предупреждений, аварийных и других сообщений. В этом случае расположенный после директивы текст заключается в одиночные или двойные кавычки. Вот пример оформления спецификации файла:

```
filspc db 'c:\tmp\current.pal', 00; описание спецификации файла
```

В этом примере описана спецификация файла `current.pal`, который находится на диске `C` в каталоге `TMP`. Спецификация предназначена для процедуры `BIOS`, выполняющей открытие файла для чтения или записи, поэтому ее текст заканчивается пустым байтом.

Макросы PushReg и PopReg. Подпрограммы во время выполнения не должны изменять чужие или исходные данные. Для этого перед началом основных действий в стеке сохраняется содержимое используемых регистров или переменных, а перед выходом из подпрограммы восстанавливаются исходные значения сохраненных величин. Сохранение в стеке одной величины выполняет одна команда, это же относится и к восстановлению. Существует специальное средство, позволяющее сократить запись повторяющихся однотипных действий и сделать ее более наглядной. Таким средством являются *макросы*. Подчеркнем, что сокращается только исходный текст, а не количество команд в теле задачи.

Понятие макрос (`macro`) распространяется на макроопределение (`macro definition`) и макроподстановку или макровызов (`macro substitution`). Макроопределение описывает некую заготовку текста программы, а макровызов — способ ее использования. Макроопределение существует только в исходном тексте, оно модифицируется в зависимости от указанных при вызове параметров и в измененном виде включается в тело задачи на месте каждого макровызова.

Ниже приводится пример двух простых макроопределений (пример 2.12). Их вызовы уже использовались в примерах 2.8 и 2.9, и будут неоднократно встречаться во многих примерах. Первое из них `PushReg` предназначено для сохранения в стеке содержимого регистров, а `PopReg` — для восстановления из стека ранее сохраненных регистров.

Пример 2.12. Описание макроопределений PushReg и PopReg

```
;     Сохранение в стеке регистров, перечисленных в списке reg.
PushReg   macro reg       ; заголовок макроопределения
          irp   r,<reg>   ; начало оператора повторения
```

```
          push  r          ; заготовка повторяемой команды
          endm             ; конец оператора повторения
endm                       ; конец макроопределения
;     Восстановление из стека регистров, перечисленных в списке reg.
PopReg    macro reg        ; заголовок макроопределения
          irp   r,<reg>    ; оператор повторения
          pop   r          ; заготовка повторяемой команды
          endm             ; конец оператора повторения
endm                       ; конец макроопределения
```

Макросы примера 2.12 различаются только заготовкой повторяемой команды. В одном случае это запись в стек, а в другом — выталкивание из него.

При оформлении макросов используются специальные директивы. Текст любого макроопределения начинается директивой `macro`, перед ней указывается имя макроса, а после нее, в той же строке, список аргументов, если таковые имеются, в данном примере это `reg`.

Другая, часто используемая директива — `endm`. В зависимости от контекста она указывает конец макроопределения или оператора, что и показано в примере 2.12.

Тела макроопределений примера 2.12 состоят из директивы повторения `irp`. После нее, в той же строке, указываются параметр `r` и имя списка аргументов, заключенное в угловые скобки. Оно должно совпадать с именем, указанным в директиве `macro`. В следующей строке записывается повторяемая команда, один из операндов которой `r` соответствует параметру директивы `irp`.

В общем случае тело директивы `irp` может состоять из нескольких команд или содержать другие директивы, поэтому нужен признак конца директивы `endm`.

Обнаружив в тексте программы макроопределение, Макроассемблер проверяет его синтаксис и запоминает имя и текст, не включая его в тело задачи. Исполнение макроопределения (вставка команд в тело задачи) будет производиться при каждом макровызове.

Параметры макровызова. Макровызовы или макрокоманды — это вставка текста макроопределения в нужных местах программы с подстановкой конкретных параметров, если они имеются. Для макроопределений примера 2.12 макровызов состоит из имени `PushReg` или `PopReg` и списка регистров, заключенного в угловые скобки. Угловые скобки позволяют использовать запятые между именами регистров, входящих в список.

Обнаружив макровызов, Макроассемблер находит и обрабатывает соответствующее определение, в результате чего формируются обычные команды, которые сразу компилируются и полученный код вставляется в тело задачи.

В частности, при вызове макроопределений примера 2.12 директива `irp` повторяется столько раз, сколько имен содержит список `reg`. Название каж-

дого имени выбирается из списка и подставляется в команду `push` или `pop` вместо параметра `r`.

В макроопределениях примера 2.12 разнообразие имен не ограничено явно, допустимо указание любых величин, которые могут быть операндами команд `push` и `pop`. Ими могут быть не только имена 16- и 32-разрядных регистров, но и имена переменных, кроме того, команда `push` может сохранять в стеке значения констант (непосредственное указание сохраняемой величины). Все эти величины можно использовать в макровызове. Например, во многих подпрограммах будет использоваться такой вариант макровызовов:

`PushReg <fs, gs, Cur_win>` и `PopReg <Cur_win, gs, fs>`

Обратите внимание, имена в списках `PushReg` и `PopReg` расположены в обратном порядке, поскольку работа со стеком организована по принципу "последнее записанное — первое считанное".

Альтернативным способом сохранения группы регистров в стеке и восстановления их оттуда одной командой являются операции `pusha` и `popa`, не имеющие параметров. Их исполняют все модели микропроцессоров, кроме Intel 8086. Эти операции сохраняют в стеке или восстанавливают из него регистры в таком порядке:

```
PUSHA  -> ax, cx, dx, bx, sp, bp, si, di.
POPA   ->  di, si, bp, sp, bx, dx, cx, ax.
```

Начиная с модели Intel 80386, микропроцессоры поддерживают операции `pushad` и `popad`, также не имеющие параметров. Они сохраняют в стеке или восстанавливают из него 32-разрядные регистры в таком порядке:

```
PUSHAD-> eax, ecx, edx, ebx, esp, ebp, esi, edi.
POPAD -> edi, esi, ebp, esp, ebx, edx, ecx, eax.
```

В отличие от макрокоманд, четыре описанные операции сокращают не только исходный текст программы, но и размер задачи и время выполнения операции, поскольку вместо восьми команд используется одна. Их недостаток в том, что обязательно сохраняются все восемь указанных регистров, но не сохраняются сегментные регистры. Поэтому в каждом конкретном случае надо решать, что лучше использовать — указанные операции или макрокоманды.

2.7. Раздел для начинающих

В данной главе приведены первые примеры программ, поэтому имеет смысл поговорить о принятом в книге оформлении текстов примеров, об описании используемых в них переменных и о некоторых общих вопросах, связанных

с оформлением программ на языке ассемблера. Если вы владеете ассемблером и имеете опыт программирования на этом языке, то данный раздел можно пропустить без ущерба для понимания излагаемого в последующих главах материала. Начинающим рекомендуем обратиться к специальной литературе, например [6, 7], а при наличии доступа к Internet посетите сайт `www.assembler.ru`. Данная книга не является руководством по программированию на языке ассемблера, она содержит лишь минимум сведений, необходимый для понимания действий, выполняемых в примерах.

Ассемблер — это язык команд семейства компьютеров, в котором коды инструкций и операндов заменены мнемоническими обозначениями, т. е. именами. Используемые в ассемблерных программах имена делятся на две категории — зарезервированные и выбираемые по усмотрению разработчика.

К зарезервированным относятся имена регистров, операций, директив, операторов и некоторые другие. Их нельзя изменять или использовать не по назначению, в противном случае при компиляции программы будет выдано сообщение об ошибке.

По усмотрению программиста выбираются имена меток, констант, переменных, структур данных, макросов, сегментов программ и некоторые другие. Желательно, чтобы имена несли смысловую нагрузку, т. е. указывали назначение величин, к которым они относятся, и легко читались.

В именах можно использовать заглавные и строчные буквы латинского алфавита, цифры и те символы, которые не имеют специального назначения. К последним относятся: пробел, запятая, двоеточие, точка с запятой, все виды скобок и знаки арифметических операций. По умолчанию, т. е. если явно не указано обратное, ассемблер не различает в именах заглавные и строчные буквы. Поэтому не имеет значения заглавными или строчными буквами набрано имя, или в нем чередуются те и другие.

Для большей наглядности текста отдельные части составных имен могут выделяться заглавными буквами, например `OldMode`, или отделяться друг от друга нижней чертой, например `Cur_win`. Не следует увлекаться слишком длинными именами, поскольку теряется наглядность и повышается вероятность ошибки при их наборе.

Команда — это элементарная единица любой ассемблерной программы, исполняемая процессором ПК. Нас интересует язык ассемблера для микропроцессоров Intel, поскольку на их основе собираются компьютеры семейства IBM PC. В записи на этом языке команда состоит из условного обозначения операции и операндов, количество которых изменяется от 0 до 3 (чаще всего 1 или 2). Операнды отделяются от операции пробелами, а друг от друга запятой и пробелами. Наличие запятой обязательно, а количество пробелов не ограничивается, поэтому вы можете оформлять текст своей программы так, как сочтете нужным.

Перед командой может находиться метка, имя которой заканчивается символом "двоеточие". Метки являются операндами команд передачи управления — условных и безусловных переходов и обращений к подпрограммам.

После команды может записываться комментарий, который отделяется от нее символом "точка с запятой". Обнаружив этот символ в текущей строке, Макроассемблер просто пропускает весь текст до конца строки. Поэтому его можно использовать только как признак последующего комментария, а в комментарии допустимы любые символы.

Вернемся к командам. Операции имеют зарезервированные имена и их изменения недопустимы. Ассемблер формирует код машинной инструкции исходя из имени операции и результатов анализа операндов. Поэтому одной операции может соответствовать несколько разных кодов машинных инструкций. Например, имя `mov` обозначает операцию пересылки, которую выполняют 8 разных инструкций микропроцессора Intel 80386.

В качестве операндов могут использоваться имена регистров, констант, переменных, меток и выражения, составленные из перечисленных величин.

Регистры являются внутренними устройствами микропроцессора, т. е. они входят в его состав, поэтому обращение к ним происходит быстрее, чем к оперативной памяти. Начиная с Intel 8086, все модели микропроцессоров содержат 16-разрядные регистры, имеющие следующие имена:

/1/ `AX, BX, CX, DX, DI, SI, BP, SP, CS, DS, ES, SS.`

Первые четыре имени списка /1/ относятся к регистрам общего назначения. При выполнении задач в них обычно находятся сами операнды, а не их адреса. Исключением является регистр `BX`, в котором может храниться адрес операнда. Каждый из регистров общего назначения делится на два независимых байта. Старшим байтам соответствуют имена `AH`, `BH`, `CH`, `DH`, а младшим байтам — `AL`, `BL`, `CL`, `DL`.

Имена `DI`, `SI`, `BP`, `SP` относятся к регистрам-указателям. Обычно они содержат адреса оперативной памяти, в которых хранятся операнды. В таких случаях при записи команды имя регистра заключается в квадратные скобки. Регистры-указатели не делятся на байты, поскольку адрес не может быть 8-разрядным. `DI` и `SI` обычно используются при работе с данными, поэтому по умолчанию в качестве сегментного регистра процессор выбирает `DS`. Специально для работы со стеком предназначены `BP` и `SP`, поэтому по умолчанию в качестве сегментного регистра используется `SS`.

Существует специальный регистр, содержащий адрес очередной выполняемой команды. В русскоязычной литературе его называют счетчиком команд, а в англоязычной — указателем инструкций (`IP`). В явном виде он не указывается ни в одной команде, поэтому его имя отсутствует в списке /1/. Тем не менее, все команды передачи управления изменяют содержимое `IP`.

Последняя четверка имен списка /1/ соответствует сегментным регистрам. Они предназначены для хранения старшей части адресов операндов или команд. Содержимое CS (сегмент кодов) процессор использует при выборке очередной команды. Содержимое DS (сегмент данных) — при чтении и записи операндов. В SS хранится сегмент оперативной памяти, отведенный для стека. Регистр ES используется строковыми командами при записи результата, в остальных случаях программист может распоряжаться им по своему усмотрению. Во всех примерах, приводимых в данной книге, регистр ES используется при обращениях к видеопамяти, поэтому в нем должно находиться значение сегмента видеопамяти (видеосегмента).

В записях операндов имя сегментного регистра предшествует адресу и обязательно заканчивается символом "двоеточие". Адресами, чаще всего, являются имена меток или переменных и индексные выражения, но допустимо и явное указание адреса в виде конкретного числа.

Начиная с модели Intel 386, регистры общего назначения и указатели расширены до 32-х разрядов. Размеры сегментных регистров не изменились, но добавились два новых. Новые имена перечислены в списке /2/.

/2/ EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, FS, GS.

Расширение регистров и введение новых имен никак не отразилось на назначении и возможности использования старых. Просто 16-разрядные регистры стали младшими словами 32-разрядных регистров. Однако старшие слова 32-разрядных регистров самостоятельно не существуют и поэтому не имеют собственных имен.

Одновременно с введением новых регистров был расширен набор способов адресации операндов. Адрес может находиться в любом из 32-разрядных регистров, а EAX, EBX, ECX и EDX могут использоваться в индексных выражениях.

Поскольку полный адрес помещается в 32-разрядных регистрах, то сегментные регистры просто не нужны, именно поэтому их разрядность не была увеличена. Таким образом, сегментные регистры являются атрибутом вычислений с использованием 16-разрядных адресов операндов.

Кроме регистров, перечисленных в списках /1/ и /2/, существуют еще специальные регистры, используемые в системных задачах, программирование которых не рассматривается в данной книге.

Константы — это постоянные величины, которым присвоены определенные имена. Они описываются (или определяются) с помощью операторов присваивания, имеющих следующую структуру:

```
<имя константы> = <арифметическое или логическое выражение>
```

Простейший пример такого оператора CR = 0Dh. Если его включить в текст программы, то при компиляции все имена CR будут заменены на код 0Dh

(аббревиатура `CR` расшифровывается как "возврат каретки" и ей соответствует код `0Dh`). В более сложных случаях в выражениях кроме чисел могут использоваться имена констант и переменных, символы, обозначающие арифметические или логические операции, и круглые скобки для указания порядка выполнения вычислений.

При программировании на ассемблере константы могут быть только целыми числами, поэтому операторы типа `PI = 3.14` вызовут сообщение об ошибке. Не все операции допускают использование констант, например, константу нельзя записать в сегментный регистр с помощью операции пересылки (`mov`).

Более общей формой описания констант является использование директивы `EQU` вместо знака равенства. Если справа (после директивы) указано арифметическое или логическое выражение, то обе директивы (`EQU` и `=`) равноценны. Однако после `EQU` можно записать символьное выражение, которое Макроассемблер будет подставлять вместо имени константы при каждом ее использовании в программе. Кроме того, после `EQU` можно указать произвольный текст, заключенный в угловые скобки. В таком случае имя константы будет соответствовать указанному тексту (за исключением угловых скобок).

Ассемблер не выделяет специального места в теле задачи для хранения констант. При каждом обнаружении имени константы он просто подставляет в формируемую команду соответствующее значение, символьное выражение или строку текста. Этим константы отличаются от переменных.

Сегментирование. При компиляции программы (при ее преобразовании в объектный код) Макроассемблер преобразует имена переменных в относительные адреса. Следовательно, должна существовать некая точка отсчета, адрес которой можно принять за нуль. Такой точкой является начало сегмента. В памяти ПК сегмент — это произвольно выбранный участок адресов, размер которого не меньше чем 16 и не больше чем 65 536 байтов. Наименьшее значение объясняется принятым на IBM PC способом вычисления полного адреса, а наибольшее соответствует предельному значению числа, которое может быть записано в 16-разрядный регистр.

Формально в тексте программы должен быть описан хотя бы один сегмент, в противном случае вычисление адресов будет невозможно и Макроассемблер выдаст сообщение об ошибке. Существует специальный класс задач, при программировании которых можно ограничиться одним сегментом. Это так называемые резидентные задачи, которые постоянно находятся в памяти ПК. Типичным примером являются загружаемые драйверы. Обычно при работе в ПК загружено несколько резидентных задач. Поэтому, чем меньше места в памяти занимает такая задача, тем лучше.

В задачах среднего размера, обычно используется три сегмента для размещения данных, команд и стека. Они ассоциируются с тремя сегментными

регистрами DS, CS и SS, используемыми процессором при выполнении команд. Выделение нескольких сегментов в структуре программы преследует две основные цели. Во-первых, ее текст становится более удобочитаемым. Во-вторых — это один из способов увеличения пространства памяти, отведенного для задачи. Вопросы сегментирования программ и распределения памяти обсуждаются в приложении Б данной книги. Здесь мы попробуем разобраться с тем, как устанавливается связь между сегментами программы и соответствующими регистрами процессора.

Любая задача должна иметь явно описанную точку входа, в противном случае DOS не сможет начать ее выполнение. Точкой входа является метка первой выполняемой команды задачи, кроме того, ее имя указывается после директивы END, которой заканчивается текст программы. По этому имени Макроассемблер находит сегмент, в котором описана метка, и значение этого сегмента будет записано в регистр CS перед пуском задачи. Отметим, что определить точные значения сегментов может только DOS при загрузке задачи для выполнения, поскольку именно в этот момент известно распределение памяти ПК и ее доступное пространство. Таким образом, сегмент, содержащий команды, опознается Макроассемблером независимо от присвоенного ему в программе имени.

Для ассоциирования других сегментов с соответствующими регистрами процессора надо либо использовать специальные директивы при их описании, либо загружать значения сегментов в регистры в процессе выполнения задачи. Покажем, как записать в регистр данных значение сегмента, описанного в примере 2.11.

```
;   точка входа в задачу, имя start надо указать после последнего end.
start: mov   ax, data     ; запись значения сегмента в ax
       mov   ds, ax       ; копирование ax в ds
;   продолжение программы
```

Первая команда этого фрагмента записывает значение указанного сегмента в регистр ax, а вторая переписывает содержимое ax в ds. Промежуточный регистр ax нужен потому, что имя сегмента (data) является константой и его нельзя записать в сегментный регистр с помощью операции пересылки. После выполнения этих двух команд процессор "знает", где находятся операнды команд. Таким способом можно записать в любой сегментный регистр значение нужного сегмента.

Вне сегментов, т. е. в начале текста программы располагаются директивы, которые используются только при компиляции. К ним относятся описание констант, о котором говорилось выше, и макроопределения.

Макроопределения и макровызовы объединяет одно общее понятие "Макросы". Это средство для сокращения исходных текстов программ (но не задач), придания ему большей наглядности и упрощения процесса программирования. В комплект поставки Макроассемблера входит несколько

специальных файлов, содержащих различные полезные макроопределения. Эти файлы обычно имеют тип (расширение) `inc`. Если при установке компилятора создается каталог `Include`, то они располагаются в нем.

В данном разделе автор попытался ответить на часть вопросов, которые могут возникнуть у читателя при изучении приведенных в книге примеров. По мере изложения материала будут описаны особенности выполнения некоторых команд, использованных в примерах. Многие из приведенных в книге примеров оформлены в виде подпрограмм. Поэтому приложение В специально посвящено вопросам, связанным с разработкой и использованием подпрограмм.

ГЛАВА 3

Видеорежимы packed pixel graphics

Видеорежимы стандарта VESA различаются по разрешающей способности и размерам палитры цветов, которые можно одновременно изобразить на экране. В данной, а также в трех последующих главах изложен материал, относящийся, в первую очередь, к режимам `packed pixel graphics` (упакованная точечная графика), которые в дальнейшем будут сокращенно обозначаться как `PPG`. При работе в этих режимах код точки занимает один байт и является номером строки палитры, содержащей описание цвета. В палитре может быть описано только 256 цветов. Работа с цветом во многом отличается от построения графических объектов, поэтому ее описание вынесено в отдельную (следующую) главу.

В этой главе рассмотрены способы построения простейших графических объектов. В ней описано, как выводить на экран точки, рисовать линии, прямоугольники, рамки и заранее заготовленные рисунки. В большинстве графических приложений эти действия являются основными, и автор счел целесообразным описать логику их выполнения независимо от манипуляций с цветом точек создаваемого изображения.

При построении графических объектов надо учитывать разрешающую способность режима и размер кода точки изображения. Все приведенные примеры будут выполняться независимо от разрешающей способности режима при условии, что код точки занимает один байт. При работе в полноцветных режимах `direct color` размер кода точки изменяется, он занимает 2 или 4 байта. Для того чтобы примеры могли выполняться в этих режимах, в них придется внести незначительные изменения.

Стандарт VESA допускает использование палитры, содержащей 16 цветов (`EGA graphics`), но мы не будем рассматривать такие режимы. Они описаны в многочисленных руководствах, например в книге [5], где рассмотрены особенности всех режимов, соответствующих стандарту IBM, и приведены примеры программ.

3.1. Работа с отдельными точками

Компьютерная графика, независимо от ее сложности, в конечном итоге сводится к работе с отдельными точками изображения. В режимах PPG каждой точке экрана соответствует байт видеобуфера. При неизменной палитре цвет точки зависит от содержимого этого байта.

3.1.1. Команды для манипуляции с точками

В графических режимах VESA, за исключением EGA graphics, доступ к видеобуферу ничем не отличается от доступа к оперативной памяти (ОЗУ, RAM). Поэтому для чтения или изменения содержимого байтов видеопамяти используются команды, выполняющие пересылку и сдвиг операндов, логические, арифметические и прочие операции. Начиная с микропроцессора Intel 386, они могут манипулировать байтами, словами и двойными словами. Соответственно, при работе в режимах PPG одной командой можно обработать одну, две или четыре подряд расположенные на экране точки. Пересылка операндов является одним из наиболее частых действий при работе с видеопамятью. Ее выполняют команда mov и строковые команды movs, stos и lods. Напомним их основные свойства и различия.

Команда mov двухадресная, она копирует содержимое источника (source) в приемник (destination). Приемник всегда является первым операндом, а источник вторым. Команда допускает разные способы адресации операндов, от этого зависит конкретный код машинной инструкции, которую Макроассемблер формирует при компиляции. Однако оба операнда не могут одновременно находиться в памяти. Для копирования содержимого одного байта, слова или двойного слова в другой байт, слово или двойное слово нужны две команды пересылки, использующие в качестве посредника один из регистров общего назначения (табл. 3.1).

Таблица 3.1. *Пересылка из памяти в память*

Пересылка байта	Пересылка слова	Пересылка двух слов
mov al, fs:[di] mov gs:[si], al	mov ax, fs:[di] mov gs:[si], ax	mov eax, fs:[di] mov gs:[si], eax

При записи операндов команды mov можно использовать имена всех сегментных регистров: cs, ds, es, fs, gs и ss. Если сегментный регистр не указан явно, то подразумевается, что это ds. В сегментном регистре должно находиться конкретное значение сегмента обычной, расширенной или видеопамяти. Смещение (относительный адрес) байта, слова или двойного слова в этом сегменте, чаще всего, указывается в индексном регистре, имя которого

заключается в квадратные скобки (это признак адреса). В табл. 3.1 при чтении из памяти полный адрес операнда задается в регистрах `fs:di`, а при записи в память — в регистрах `gs:si`.

Строковые инструкции lods, movs и stos отличаются от команды пересылки ***(mov)*** следующими особенностями:

- имя инструкции может содержать дополнительную пятую букву — `b`, `w` или `d`, указывающую количество пересылаемых байтов (1, 2 или 4);
- если имя инструкции содержит пять букв, то операнды не указываются, их местонахождение определено по умолчанию и зависит от инструкции;
- после выполнения соответствующей пересылки содержимое индексных регистров, содержащих адреса операндов, увеличивается или уменьшается;
- увеличение или уменьшение адресов операндов зависит от состояния специального признака направления пересылки (`direction flag`);
- только перед строковой инструкцией может быть указана специальная команда `rep`, вызывающая ее многократное повторение.

Назначение инструкций пересылки, имена которых состоят из пяти букв, показано в табл. 3.2. Местонахождение операндов у них фиксировано, и изменить его нельзя. Один из операндов может находиться в регистре-аккумуляторе, а другой или оба — в оперативной памяти. Последняя буква имени инструкции (`b`, `w` или `d`) указывает, какой из этих регистров является аккумулятором — `al`, `ax` или `eax`.

Адрес операнда, находящегося в оперативной памяти, задается в одной из двух пар регистров — `ds:si` или `es:di`. Содержимое этой пары должно быть определено в задаче до выполнения инструкций пересылки.

Таблица 3.2. *Назначение строковых операций*

Размер операнда в байтах	Чтение памяти в аккумулятор accum = ds:[si]	Запись аккумулятора в память es:[di] = accum	Пересылка из памяти в память es:[di] = ds:[si]
1	lodsb	Stosb	movsb
2	lodsw	Stosw	movsw
4	lodsd	Stosd	movsd

Смена сегмента источника. Очевидным недостатком строковых операций являются фиксированные сегменты операндов источника и приемника. Сегмент приемника менять нельзя, а вот сегмент источника можно изменить. Для этого у имени операции отбрасывается последняя буква (`b`, `w` или `d`) и явно описывается сегмент операнда источника. Имя регистра, содержащего адрес (смещение) операнда, в этом сегменте изменить нельзя, у

источника это регистры si или esi. Вот пример корректной записи строковых операций:

```
lods    byte ptr gs:[si]           ; загрузка байта al = gs:[si]
movs    dword ptr es:[di], fs:[si] ; копирование двойного слова
stos    word ptr es:[di]           ; эквивалентна stosw  es:[di] = ax
```

Из этого примера видно, что в записи операнда источника можно указать любой сегментный регистр (в данном случае gs или fs), но в записи операнда приемника может быть указан только регистр es. Если вы укажете у приемника другое имя сегментного регистра, то Макроассемблер просто использует имя es, не выдавая сообщение об ошибке.

При отсутствии пятой буквы в имени инструкции Макроассемблер не может определить размер (тип) операнда исходя из текста программы. Поэтому обязательно используется оператор ptr, перед которым указывается размер операнда — byte, word или dword. При отсутствии явного описания типа операнда Макроассемблер выдаст сообщение об ошибке.

Следует отметить, что смену сегмента операнда источника допускают все строковые операции, а не только перечисленные в табл. 3.2.

Направление пересылки. После выполнения строковой операции адрес, находящийся в индексном регистре (или в двух регистрах) увеличивается или уменьшается на размер операнда (на 1, 2 или 4). В первом случае принято говорить о пересылке в прямом направлении, а во втором — в обратном.

Перед коррекцией адреса микропроцессор проверяет состояние флага направления (direction flag), который хранится в седьмом разряде регистра флагов. Если этот разряд очищен, то содержимое индексных регистров увеличивается на размер операнда, а если он установлен, то уменьшается.

Состояние седьмого разряда регистра флагов изменяют две специальные команды cld и std. Первая (cld) очищает разряд, разрешая тем самым пересылку в прямом направлении. Вторая (std), наоборот, устанавливает разряд, разрешая пересылку в обратном направлении. Обе команды не имеют операндов. Обычное состояние флага направления — очищенное.

Таким образом, выражение "пересылка в прямом направлении" означает, что операнды записываются в память или считываются из нее в порядке увеличения их адресов. Соответственно, выражение "пересылка в обратном направлении" означает обработку операндов в порядке уменьшения их адресов.

Программные циклы. Циклом принято называть многократное возвращение на начало группы команд до тех пор, пока не будет выполнено заданное условие. Способ управления повторами цикла зависит от того, что именно является условием их прекращения. Здесь нас интересуют только циклы, управляемые по счетчику, при входе в них указывается требуемое число повторов.

Для управления циклом по счетчику предназначена специальная команда `loop`. Она имеет два параметра, но явно в команде указывается только один из них — метка, на которую передается управление (обычно это метка начала цикла). Неявно `loop` использует счетчик повторов, которым является содержимое регистра `cx`. При каждом выполнении команды содержимое `cx` уменьшается на 1, и если результат отличен от нуля, то управление передается на указанную в команде метку. В противном случае будет выполняться команда, расположенная после `loop`.

В примере 3.1 показан простой цикл очистки полного сегмента памяти, т. е. 65 536 байтов. Код очищаемого сегмента должен находиться в регистре `es`. Для ускорения при каждом повторе очищаются 4 байта.

Пример 3.1. Очистка сегмента с использованием команды mov

```
      xor   eax, eax      ; очистка регистра eax
      xor   di, di        ; di = 0, адрес начала сегмента
      mov   cx, 16384     ; cx = 16384, счетчик повторов
lps:  mov   es:[di], eax  ; запись 4-х байтов в сегмент
      add   di, 04        ; di = di + 4, коррекция адреса
      loop  lps           ; управление повторами цикла
```

В примере 3.1 цикл очистки имеет имя `lps` и состоит из трех команд. Первая из них очищает 4 байта памяти, вторая увеличивает хранящийся в регистре `di` адрес на 4, а третья повторяет цикл 16 384 раза.

Важно

Команда `loop` может передать управление на метку, которая отстоит от нее не далее чем на 128 байтов. Если это условие нарушено, то при компиляции Макроассемблер выдаст сообщение об ошибке.

Для многократного повторения одной строковой операции предназначена команда `rep`, которую называют префиксом повторения. Так же, как команда `loop`, она использует счетчик повторов, хранящийся в регистре `cx`. В примере 3.2 показано применение строковой операции для очистки сегмента.

Пример 3.2. Микропрограммный цикл очистки сегмента памяти

```
xor       eax, eax     ; очистка регистра eax
xor       di, di       ; di = 0, адрес начала сегмента
mov       cx, 16384    ; cx = 16384, счетчик повторов
rep       stosd        ; очистка сегмента, указанного в ES
```

В примере 3.2 цикл пересылки сократился до одной команды, выполнение которой микропроцессор повторяет до тех пор, пока содержимое `cx` не ока-

жется равным нулю. Безусловно, микропрограммный цикл выполняется значительно быстрее, чем программный.

Инструкции `rep` и `loop` различаются способом работы со счетчиком повторов. `Rep` *сначала проверяет* содержимое регистра `cx` и, если оно отлично от нуля, выполняет строковую операцию и потом уменьшает содержимое `cx` на 1. `Loop` *сначала уменьшает* содержимое `cx` на 1 и в зависимости от результата повторяет или прекращает выполнение цикла. Это различие проявляется, если при входе в цикл регистр `cx` очищен. В таком случае указанная после `rep` операция не выполнится ни разу, в то время как команда `loop` будет повторять выполнение цикла 65 536 раз!

Таким образом, при обмене данными с видеопамятью можно использовать как обычные, так и строковые операции пересылки. В тех случаях, когда возможен выбор, предпочтение следует отдавать строковым операциям.

3.1.2. Окна видеопамяти

При работе в видеорежимах `SVGA` изображение, находящееся на экране монитора содержит большое количество точек. Оно зависит от разрешающей способности установленного режима и равно произведению количества точек в строке на количество строк. Например, при разрешении 640×480 на экране находится 307 200 точек, а при разрешении 1600×1200 — 1 920 000 точек. Объем видеопамяти, необходимый для хранения содержимого экрана, также зависит от размера кода точки. В режимах `PPG` код точки занимает 1 байт, поэтому объем видеопамяти совпадает с количеством точек, находящихся на экране, в режимах `direct color` он в 2 или 4 раза больше. Следовательно, для работы с графическими объектами в режимах `SVGA` нужен доступ к большому пространству видеопамяти.

Сегментирование памяти. Независимо от конкретного назначения команды IBM PC всегда имеют доступ к ограниченному пространству адресов, предельный размер которого зависит от нескольких факторов, в том числе и от режима работы микропроцессора. Начиная с модели Intel 80386, микропроцессоры могут работать в реальном, виртуальном и защищенном режимах. В реальном и виртуальном режимах все пространство памяти делится на сегменты, предельный размер которых составляет 65 536 байтов. Указанные в командах адреса операндов всегда относятся к конкретному сегменту, и ни при каких условиях не могут выходить за его пределы, — это вызывает аварийную ситуацию. Значение сегмента хранится в одном из сегментных регистров, который либо явно указывается в имени операнда, либо используется по умолчанию.

Доступ к сегментам. При работе на IBM PC в реальном режиме для доступа к пространству адресов, расположенному за пределами сегмента, используются два разных способа, выбор которых зависит от типа памяти.

Первый способ заключается в том, что в сегментный регистр записывается абсолютный адрес начала нужного сегмента. Специфической особенностью семейства IBM PC является то, что при работе с сегментами общий объем адресуемого пространства не может превышать один мегабайт. Следовательно, возможно использование только 16 сегментов предельного размера (16×65536 = 1048576 = 1 Мбайт). Поэтому прямое указание адреса начала сегмента применяется только при работе с младшей частью оперативной памяти (первые 640 Кбайт). Как это делается, описано в приложении Б данной книги.

Второй способ заключается в том, что сегмент выполняет роль окна, через которое "видна" (доступна) та или иная часть реального пространства адресов. Содержимое сегментного регистра при этом неизменно, а доступная или "отображаемая" часть адресов изменяется по запросам задачи. Такой способ подразумевает существование специального устройства и программного обеспечения, поддерживающих работу с нужным пространством адресов. На IBM PC он применяется для доступа к видеопамяти и к расширенному пространству оперативной памяти ПК (см. приложение Б).

Доступ к видеопамяти. Рассмотрим, как организуется работа со всем пространством видеопамяти. Для обращения к нему используется специальный сегмент, который принято называть видеосегментом. При работе в графических режимах он обычно имеет код `A000h`, но лучше взять его точное значение из массива `Info` (см. главу 2). Код видеосегмента является просто признаком обращения к видеокарте, а не к какому-либо другому устройству, и *не является* частью адреса видеопамяти.

Опознав обращение к себе, видеоконтроллер получает от процессора 16-разрядный адрес и прибавляет его к старшей части, хранящейся в одном из его внутренних регистров. В результате получается полный (абсолютный) адрес ячейки видеопамяти, к которой обращается команда. Ячейкой, как обычно, может быть байт, слово или двойное слово.

Во внутреннем регистре видеоконтроллера хранится число, которое принято называть номером окна (или банка) видеопамяти. У современных видеокарт размер окна фиксирован и составляет 65 536 байтов, поэтому, зная объем видеопамяти, можно вычислить количество существующих окон. Одному мегабайту видеопамяти соответствует 16 окон, двум — 32 и т. д.

При каждой смене видеорежима регистр, содержащий номер окна, очищается, т. е. устанавливается нулевое окно видеопамяти. В дальнейшем текущий номер окна зависит только от действий, выполняемых в задаче, которая может устанавливать его любое допустимое значение. При переключении окон надо изменять содержимое внутреннего регистра видеоконтроллера, поэтому для выполнения таких действий предусмотрена специальная процедура `BIOS`. Как уже говорилось в главе 2, ее вызов через прерывание `int 10h` не существенно замедляет переключение окон, и стандарт `VBE` рекомендует прямое обращение, минуя прерывание `int 10h`.

В примере 2.8 приведены тексты трех подпрограмм для установки заданного (SetWin), следующего (NxtWin) и предыдущего (PrevWin) окна. Они работают с переменной Cur_win, имеющей размер слова и содержащей номер текущего окна. В примере 2.11 показано, как зарезервировать эту переменную в разделе данных программы. Если задача составлена корректно, то при ее выполнении значение Cur_win должно совпадать с номером окна, хранящимся в видеоконтроллере.

Закрашивание рабочей области экрана. Для иллюстрации работы с окнами рассмотрим подпрограмму, которая последовательно заполняет заданным кодом отображаемую на экране часть видеопамяти, в результате чего весь экран окрашивается в заданный цвет. Ее текст приведен в примере 3.3. Перед обращением к подпрограмме в байтах регистра eax надо указать код выбранного вами цвета. Предположим, что синему цвету соответствует код 01 (см. табл. 4.2). В таком случае для вызова подпрограммы используются следующие две команды:

```
mov       eax, 01010101h     ; запись кода 01 в байты регистра eax
call      fillscr            ; выполнение подпрограммы fillscr
; продолжение текста основной программы
```

В тексте основной (вызывающей) программы должны быть описаны макроопределения PushReg и PopReg (пример 2.12) и подпрограммы для работы с окнами (пример 2.8). Кроме того, до обращения к fillscr надо установить один из видеорежимов PPG и определить значения переменных Horsize, Versize и Vbuff. Способы описания и определения этих и других, часто используемых переменных, обсуждались в главе 2.

Пример 3.3. Закрашивание всего экрана заданным цветом

```
fillscr: PushReg <es,di,dx,cx,Cur_win,eax>; сохранение в стеке
         mov    Cur_win, 0        ; очистка переменной Cur_win
         call   SetWin            ; установка нулевого окна
         mov    es, Vbuff         ; es <= значение видеосегмента
         xor    di, di            ; 0 — исходный адрес видеопамяти
         mov    ax, Versize       ; ax <= количество строк (Versize)
         mul    Horsize           ; dx:ax = Versize * Horsize
         mov    cx, dx            ; cx = количество полных окон
         mov    dx, ax            ; dx = размер последнего окна
         pop    eax               ; восстановление содержимого eax
         ; Цикл записи в полностью заполняемые окна
fl_lp:   push   cx                ; сохранение счетчика сегментов
         mov    cx, 16384         ; количество повторов для rep
         rep    stosd             ; запись в полное окно
         call   NxtWin            ; установка следующего окна
         pop    cx                ; восстановление счетчика сегментов
         loop   fl_lp             ; управление повторами цикла
```

```
        ; Запись в частично заполненное окно, если оно существует
        mov     cx, dx              ; cx <= оставшееся число байтов
        rep     stosb               ; запись в неполное окно
        ; Действия, связанные с завершением работы подпрограммы
fl_out: pop     Cur_win             ; восстановление Cur_win
        call    SetWin              ; восстановление исходного окна
        PopReg <cx,dx,di,es>        ; восстановление регистров
        ret                         ; возврат из подпрограммы
```

Первые пять команд примера 3.3 выполняют вспомогательные действия, а именно, сохранение в стеке тех величин, которые будут испорчены при выполнении подпрограммы, установку нулевого окна видеопамяти, запись в регистр `es` кода видеосегмента и очистку регистра `di`. В основной части подпрограммы полностью и частично заполненные окна обрабатываются по-разному. Поэтому надо предварительно вычислить количество полных окон, содержащихся в рабочей области памяти, и количество байтов в частично заполненном окне, если оно есть. В примере 3.3 эти величины вычисляются путем умножения значений переменных `Versize` и `Horsize`.

Один из сомножителей команды умножения должен находиться в регистре `ax`, а второй указывается в самой команде. Старшая часть результата умножения находится в регистре `dx`, а младшая — в `ax`. Таким образом, произведение равно 65 536 · [dx] + [ax], квадратные скобки обозначают содержимое указанных в них регистров. Другими словами, после умножения в регистре `dx` находится количество полных окон, а в регистре `ax` — количество байтов в частично заполненном окне.

После умножения содержимое регистра `dx` копируется в `cx`, содержимое регистра `ax` — в `dx` и восстанавливается из стека испорченное при умножении содержимое регистра `eax` (код цвета точек). Далее действия выполняются в следующем порядке: сначала окрашиваются точки полностью заполненных окон, а затем точки частично заполненного окна, если такое есть.

Окрашивание точек полностью заполненных окон происходит в цикле, имеющем метку `fl_lp`, он повторяется столько раз, сколько полных окон содержит рабочая область экрана. Заполнение окна выполняет микропрограммный цикл `rep stosd`, для его ускорения строковая операция (`stosd`) записывает в видеопамять сразу четыре байта. Поэтому предварительно в регистре `cx` задается 16 384 повтора этой операции. После закрашивания всего сегмента происходит обращение к подпрограмме `NxtWin` для установки следующего окна. Затем из стека восстанавливается содержимое регистра `cx` и команда `loop` управляет повторами цикла.

После выхода из цикла `fl_lp` в регистр `cx` копируется из `dx` оставшееся количество байтов и выполняется микропрограммный цикл `rep stosb`.

Замечание

Если в регистре `cx` находится 0, то строковая операция не будет выполняться ни разу (см. раздел 3.1.1).

Заключительные действия выполняет фрагмент, имеющий метку `fl_out`. В нем восстанавливаются сохраненные в стеке величины, исходное окно видеопамяти и происходит возврат из подпрограммы.

Приведенный пример иллюстрирует простейший случай работы с окнами видеопамяти, когда не требуется специальная проверка достижения границы сегмента и необходимости переключения окна. В большинстве случаев такая проверка нужна и на ее выполнения затрачиваются дополнительные действия, замедляющие процесс построения изображения на экране. Исключить такие действия невозможно, но можно попытаться свести их к необходимому минимуму, в зависимости от конкретного алгоритма построения изображения. Мы будем неоднократно возвращаться к этому вопросу при описании приводимых в книге примеров.

3.1.3. Точки и их адреса

Для того чтобы записанная в видеопамять точка оказалась в нужном месте экрана, надо связать ее расположение на экране с адресом видеопамяти, по которому производится запись. Расположение точки на экране принято указывать с помощью координат, задаваемых в виде номеров строки и столбца. В данном разделе описан общий способ вычисления адреса видеопамяти по значению координат и возможные варианты его упрощения при работе со смежными точками.

Процесс отображения видеопамяти. При работе в графических режимах видеоконтроллер непрерывно выводит на экран монитора содержимое отображаемой области видеопамяти. Изображение на экране строится в направлении слева направо и сверху вниз, а коды точек выбираются из видеопамяти в порядке увеличения их адресов. Адреса памяти традиционно начинаются с нуля, поэтому если точки на экране тоже пронумеровать начиная с нуля, то порядковый номер точки будет совпадать с порядковым номером ячейки видеопамяти.

Замечание

Размер ячейки видеопамяти (кода точки) зависит от установленного видеорежима и может изменяться от одного до четырех байтов.

Мы сознательно описали упрощенную схему отображения содержимого видеопамяти на экран монитора. Фактически она может быть более сложной, поскольку стандарт VESA позволяет перемещать начало отображаемой области (`display start`) и изменять логический размер строки. Об этом гово-

рилось в главе 1 при описании функций VBE. На практике упрощенная схема применяется наиболее часто, поэтому примем ее за основу.

При описанном способе отображения адрес точки равен ее порядковому номеру, умноженному на размер кода в байтах (на 1, 2, 3 или 4). В режимах PPG код точки занимает один байт, поэтому ее адрес в видеопамяти совпадает с порядковым номером.

Связь адресов с координатами. Порядковый номер точки (N) вычисляется по значениям координат, задаваемых в виде номеров строки (row) и столбца (column), на пересечении которых она расположена. Номера строк и столбцов начинаются с нуля. Поэтому для вычисления порядкового номера точки надо номер строки умножить на количество точек в строке (Horsize) и к произведению прибавить номер столбца:

```
N = row · Horsize + column
```

При работе в видеорежимах PPG полученный результат является абсолютным адресом байта памяти, содержащего код точки. С учетом сегментирования видеопамяти из него надо выделить номер окна и адрес (смещение) в этом окне.

При умножении с помощью команды mul результат автоматически делится на две нужные нам части (вспомните пояснения к примеру 3.3). Номер окна находится в регистре dx, а адрес — в регистре ax. Адрес не требует никакой коррекции, а номер окна надо умножить на единицу приращения значения окна (GrUnit), которая зависит от особенностей видеоконтроллера, способ ее определения описан в главе 2. Если задача работает со страницами видеопамяти, то к результату умножения номера окна на GrUnit прибавляется значение базового окна (значение переменной Base_win).

При работе в графических режимах VGA такие вычисления выполняют две специальные функции прерывания int 10h. Функция 0Ch записывает, а 0Dh считывает код точки по заданным номерам страницы, строки и столбца. Однако для режимов SVGA эти функции непригодны, поскольку реализованный в них алгоритм рассчитан на расположение точек в пределах одного сегмента видеопамяти. Поэтому вам придется составить и включить в текст задачи собственную подпрограмму для вычисления адресов точек видеопамяти.

Подпрограмма CallWin. В примере 3.4 приведен текст подпрограммы, вычисляющей адрес точки описанным способом. Вычисленный адрес окна присваивается переменной Cur_win, окно устанавливается и оказывается текущим. Смещение (адрес) в этом окне помещается в регистр di. Перед обращением к подпрограмме в регистрах cx и dx указываются, соответственно, номера столбца и строки.

Пример 3.4. Вычисление и установка окна и адреса точки

```
CallWin: PushReg <dx, ax>    ; сохранение регистров dx и ax
         mov   ax, horsize   ; помещаем в ax размер строки
         mul   dx            ; умножаем на номер строки
         add   ax, cx        ; прибавляем номер столбца
         adc   dx, 00        ; учитываем возможность переполнения
         mov   di, ax        ; копируем адрес в регистр di
         mov   ax, GrUnit    ; единица приращения окна
         mul   dl            ; умножаем на номер окна
         add   ax, Base_win  ; !! только при работе со страницами !!
         mov   Cur_win, ax   ; копируем окно в Cur_win
         PopReg <ax, dx>     ; восстанавливаем регистры
         jmp   SetWin        ; установка окна и выход
```

Прежде чем рассматривать примеры использования этой подпрограммы, несколько слов об особенностях команды умножения (`mul`). При ее записи явно указывается только один операнд, второй выбирается из аккумулятора (`al`, `ax` или `eax`), куда его надо предварительно поместить. Команда может умножать байты, простые или двойные слова, размер сомножителей определяется по размеру (типу) указанного в команде операнда. В зависимости от размера сомножителей произведение может содержать 16, 32 или 64 разряда и соответственно находиться в регистре `ax`, в регистрах `dx` и `ax`, или в регистрах `edx` и `eax`. В примере 3.4 первая команда `mul dx` умножает слова, поэтому произведение расположено в регистрах `dx` и `ax`, а вторая (`mul dl`) умножает байты, поэтому результат занимает только регистр `ax`.

Для того чтобы с адресом точки можно было работать, надо установить вычисленное окно видеопамяти. Поэтому в примере 3.4 номер окна записывается в `Cur_win` и происходит переход на процедуру `SetWin`, которая устанавливает окно. Смещение в окне помещается в регистр `di` для того, чтобы его можно было использовать для записи кодов точек с помощью строковой операции `stos`.

В примере 3.4 нет команды возврата из подпрограммы (`ret`). Она не нужна потому, что процедура установки окна не вызывается командой `call SetWin`, а происходит безусловный переход на ее начало (`jmp SetWin`). Возврат на вызывающий модуль выполнит процедура `SetWin`.

В данном случае нет острой необходимости в указанной замене. Мы привели ее в качестве примера того, как можно исключать ненужные действия. При каждой такой замене исключаются одна команда и несколько тактов при обращении к процедуре.

Важно

При замене команды `call` командой `jmp` надо следить за тем, чтобы в верхушке стека находился адрес возврата на вызывающий модуль.

Указание номеров столбца и строки в регистрах `cx` и `dx` выбрано для совместимости с драйвером манипулятора "мышь", который описан в главе 6. При каждом перемещении мыши приходится вычислять новый адрес начала рисунка курсора, это и объясняет выбор указанных регистров.

Использование CallWin. В примере 3.5. показано, как можно поместить точку белого цвета в центр экрана. Будем считать, что переменные `Horsize` и `Versize` содержат количество точек на экране по горизонтали и вертикали, белый цвет имеет код `0Fh`, а регистр `es` содержит код сегмента видеобуфера.

Пример 3.5. Вывод белой точки в центр экрана

```
mov       dx, versize     ; количество точек по вертикали
shr       dx, 01          ; уменьшаем в 2 раза
mov       cx, horsize     ; количество точек по горизонтали
shr       cx, 01          ; уменьшаем в 2 раза
call      CallWin         ; устанавливаем окно и адрес
mov       al, 0Fh         ; помещаем в al код белого цвета
mov       es:[di], al     ; рисуем точку
; продолжение программы
```

Адреса каждой точки вычисляются тем или иным способом при любой работе с графическими объектами — от вывода на экран заранее подготовленного рисунка до построения сложных геометрических фигур. Поэтому эффективность любого алгоритма, предназначенного для работы с графикой, во многом зависит от того, как организована работа с адресами точек.

Наибольшее время занимает вычисление адреса каждой точки по значениям ее координат. Поэтому процедуры типа `CallWin` используются только для нахождения адресов опорных точек, начиная с которых производится построение изображения. Например, такой точкой может быть левый верхний угол прямоугольной области, в которой должен располагаться рисунок.

Адреса остальных точек изображения вычисляются упрощенными способами, в основе которых лежат рекуррентные соотношения, связывающие значения координат или адресов текущей и следующей точек.

Смежные точки и их адреса. Смежные точки расположены на экране монитора рядом друг с другом. Если опорная точка не лежит на границе экрана, то ее окружает 8 смежных точек. Их расположение показано в левой части табл. 3.3, где опорной является точка с номером 0. В правой части таблицы приведены приращения адресов видеопамяти смежных точек относительно адреса опорной точки. Для вычисления адреса смежной точки к опорному адресу прибавляется смещение, указанное в соответствующей ячейке таблицы. Буква `h` обозначает переменную `Horsize`.

Таблица 3.3. *Расположение и адреса смежных точек*

Расположение точек			Приращения адресов		
8	6	7	-1-h	-h	1-h
5	**0**	1	-1	**0**	1
4	2	3	h-1	h	h+1

Замечание

Команды, выполняющие строковые операции, обязательно изменяют содержимое индексного регистра, поэтому после их выполнения он содержит адрес не текущей, а следующей точки. Аналогичная ситуация возникает и после выполнения цикла построения строки с использованием обычных команд. Поэтому при составлении программы разберитесь, какой именно адрес вы будете корректировать, — возможно, он уже увеличен на 1.

При применении рекуррентных формул вычисленный адрес может выходить за пределы текущего окна. Например, горизонтальная линия может быть нарисована в таком месте экрана, что коды ее точек расположатся в двух соседних окнах, случай редкий, но вполне реальный. В зависимости от длины вертикальной линии коды ее точек могут располагаться сразу в нескольких окнах. Это обстоятельство надо учитывать при работе с адресами и своевременно изменять номера текущих окон.

3.2. Построение геометрических фигур

Геометрические фигуры являются наиболее подходящими объектами для первого знакомства с компьютерной графикой. Способы их построения зависят только от природы самой фигуры (линия, прямоугольник, эллипс и т. д.) и от используемого видеорежима. Если угодно, это программирование компьютерной графики в чистом виде, без многочисленных вспомогательных действий, которые неизбежны при выводе заранее заготовленных рисунков.

3.2.1. Прямые линии

Прямые линии бывают горизонтальные, вертикальные и наклонные, от этого зависят способы (алгоритмы) их рисования. Линии на экране далеко не всегда являются гладкими, в большинстве случаев они ступенчатые. Гладкими могут быть только линии, угол наклона которых равен нулю или кратен 45 градусам. При других углах наклона линия становится ступенчатой.

В данном разделе описаны подпрограммы для рисования гладких линий, иллюстрирующие различные способы работы с адресами точек. Основную часть раздела занимают алгоритмы рисования горизонтальных прямых.

Рисование линии слева направо. В примере 3.6 приведены два варианта подпрограммы, для рисования горизонтальной линии в направлении слева направо. Перед их вызовом должно быть установлено окно видеопамяти, содержащее первую точку прямой, а ее адрес в этом окне указан в регистре `di`. В регистрах `cx` и `al` помещаются, соответственно, количество точек в линии (длина прямой) и их код (цвет).

В этом и ***всех последующих*** примерах предполагается, что регистр `es` содержит адрес видеосегмента (значение переменной `Vbuff`).

Пример 3.6. Подпрограммы для рисования горизонтальной линии

```
        ; Вариант 1, используется команда пересылки
horline:mov   es:[di], al ; запись кода точки в видеобуфер
        inc   di          ; увеличение адреса на 1
        jne   @F          ; переход, если не нуль
        call  NxtWin      ; установка следующего окна
@@:     loop  horline     ; управление повторами цикла
        ret               ; возврат из подпрограммы
        ; Вариант 2, используется строковая операция
horline:stosb             ; запись кода точки в видеобуфер
        or    di, di      ; начало нового сегмента ?
        jne   @F          ; -> нет
        call  NxtWin      ; установка следующего окна
@@:     loop  horline     ; управление повторами цикла
        ret               ; возврат из подпрограммы
```

Различие между подпрограммами примера 3.6 состоит в том, что в одном случае для записи кода точки в видеопамять использована обычная команда пересылки, а в другом — строковая, которая сама увеличивает содержимое регистра `di` на 1. Поэтому в первом варианте адрес надо увеличивать, что и делает команда `inc di`, а во втором варианте просто проверяется его новое значение, это делает команда `or di, di`.

Как уже говорилось, при коррекции значение адреса может выйти за границу сегмента. В таком случае надо установить следующее окно. По мере записи кодов точек в видеопамять содержимое регистра `di` возрастает вплоть до значения 65 535 (код `0FFFFh`). Если к этой величине прибавить 1, то регистр `di` окажется очищенным, это и использовано в примере 3.6 в качестве признака необходимости смены окна. Если содержимое регистра `di` отлично от нуля, то команда `jne @F` обходит вызов процедуры `NxtWin`, а если равно нулю, то она выполняется и происходит установка следующего окна.

В примере 3.6 впервые использованы *локальные метки*, поэтому опишем правила работы с ними. Все локальные метки имеют имя `@@`, после которого, как обычно, ставится двоеточие. В командах переходов или ветвлений вместо имени локальной метки применяются операторы `@F` или `@B`. Оператор `@F` (переход вперед) указывается, если локальная метка расположена ниже по тексту. Оператор `@B` (переход назад) применяется, если локальная метка расположена выше по тексту. Обнаружив один из этих операторов, Макроассемблер ищет в нужном направлении *ближайшую локальную метку*. Количество локальных меток в программе не ограничено, но их применение не должно затруднять визуальный анализ текста.

В обоих вариантах примера 3.6 линия рисуется слева направо, при этом номера точек и адреса байтов от шага к шагу увеличиваются. Это естественный способ построения изображения, при котором адреса точек корректируются наиболее просто, но он не применим, если при рисовании прямой значения одной или обеих координат уменьшаются. Простейшим примером является прямая линия, соединяющая правый верхний и левый нижний углы любой прямоугольной области. Ее можно провести снизу вверх или сверху вниз, но в любом случае значение одной координаты будет увеличиваться, а другой уменьшаться.

Рисование линии справа налево. Рассмотрим, как можно нарисовать на экране горизонтальную прямую линию в направлении справа налево. Два варианта подпрограмм приведены в примере 3.7, их вызов отличается от вызова подпрограмм примера 3.6 только тем, что исходное окно видеопамяти и адрес в регистре `di` соответствуют крайней правой точке прямой.

Пример 3.7. Рисование горизонтальной линии справа налево

```
        ; Вариант 1, используется команда пересылки.
invline:mov   es:[di], al ; запись кода точки в видеобуфер
        sub   di, 01      ; уменьшение адреса на 1
        jnc   @F          ; переход, если нет переноса
        call  PrevWin     ; установка предыдущего окна
@@:     loop  invline     ; управление повторами цикла
        ret               ; возврат из подпрограммы
        ; Вариант 2, используется строковая операция.
invline:std               ; установка флага направления
invlp:  stosb             ; запись кода точки в видеобуфер
        cmp   di, -1      ; начало нового сегмента ?
        jne   @F          ; -> нет
        call  PrevWin     ; установка предыдущего окна
@@:     loop  invlp       ; управление повторами цикла
        cld               ; очистка флага направления
        ret               ; возврат из подпрограммы
```

В примере 3.7 после записи в видеопамять содержимое регистра `di` уменьшается на 1, поэтому каждая следующая точка располагается на экране слева от предыдущей. Если при очередном уменьшении адреса будет пройдена нижняя граница сегмента, то надо установить *предыдущее окно* видеопамяти. Нижней границей текущего сегмента является нулевой адрес. При его уменьшении на 1 получается отрицательный результат, имеющий код `0FFFFh`, который является старшим адресом предыдущего сегмента.

Контроль текущего адреса выполняется по-разному. В первом варианте для этого проверяется состояние C-разряда регистра флагов после операции вычитания. При вычитании единицы из нуля он будет установлен, что приведет к вызову подпрограммы `PrevWin`. Во втором варианте вычитание выполняет строковая операция, не вырабатывающая признаки, поэтому проверяется код результата и если он равен "−1", то вызывается подпрограмма `PrevWin`.

Важно

В первом варианте примера 3.7 вместо команды `sub di, 01` нельзя использовать `dec di`, поскольку последняя не вырабатывает признак переноса.

Подпрограммы примера 3.6 достаточно просты, но это не самый быстрый способ рисования горизонтальной линии. Если не происходит смена окна, то при записи кода каждой точки выполняются четыре команды. В первом варианте одна из них (`jne @F`), а во втором две (`or di, di` и `jne @F`) производят проверку текущих значений адреса.

Вероятность того, что при рисовании горизонтальной линии значение адреса выйдет за границу сегмента не превышает 1%. Например, при работе в режиме `101h` точки только 4 из 480 строк расположены в двух окнах. Следовательно, примерно в 99% случаев проверка текущего адреса в процессе рисования не нужна, и выполняющие ее команды можно исключить из тела цикла записи точек. Сказанное не означает, что проверка не нужна вообще, просто она должна выполняться перед циклом рисования, а не в самом цикле.

Ускорение цикла рисования. Если из подпрограмм примера 3.6 исключить проверку адресов и установку следующего окна, то цикл записи в первом варианте подпрограммы будет состоять из трех команд, а во втором — из двух (`stosb` и `loop`). Пару команд `stosb` и `loop` можно заменить одной командой `rep stosb`, т. е. использовать микропрограммный цикл, выполняющийся быстрее программного.

В примере 3.8 приведена подпрограмма для быстрого рисования горизонтальных линий в направлении слева направо с использованием одного или двух микропрограммных циклов. При обращении к ней входные параметры задаются так же, как для подпрограмм примера 3.6.

Пример 3.8. Подпрограмма быстрого рисования горизонтальной линии

```
horline: push  dx          ; сохранение содержимого регистра dx
         mov   dx, di      ; копирование адреса в регистр dx
         add   dx, cx      ; сумма текущего адреса и количества точек
         jc    @F          ; -> прямая расположена в двух окнах
         xor   dx, dx      ; очистка регистра dx
@@:      sub   cx, dx      ; количество точек в текущем окне
         rep   stosb       ; рисуем всю прямую или ее начало
         or    di, di      ; адрес в пределах текущего окна ?
         jne   @F          ; -> да, линия нарисована полностью
         call  NxtWin      ; установка следующего окна
         mov   cx, dx      ; количество не нарисованных точек
         rep   stosb       ; рисуем остаток линии
@@:      pop   dx          ; восстановление содержимого dx
         ret               ; возврат из подпрограммы
```

Линия может размещаться в текущем окне полностью или частично. Для проверки этого в примере 3.8 текущий адрес копируется в регистр `dx` и к нему прибавляется размер линии. Если при этом не произошло переполнение, то линия полностью помещается в текущем окне и регистр `dx` надо очистить. Если при сложении произошло переполнение, то команда `jc @F` исключает очистку регистра `dx`, поскольку в нем находится количество точек остатка, который будет нарисован после смены окна. Команда `sub cx, dx` вычитает остаток (или 0) из общего числа точек и таким способом определяет количество повторов первого микропрограммного цикла. Следующая команда `rep stosb` рисует часть линии, расположенную в исходном окне видеопамяти.

Затем проверяется текущий адрес видеопамяти и если он отличен от нуля, то линия нарисована полностью и происходит выход из подпрограммы. Если же текущий адрес равен нулю, то устанавливается следующее окно видеопамяти, в регистр `cx` копируется содержимое регистра `dx` и выполняется команда `rep stosb`, рисующая остаток прямой. После этого происходит выход из подпрограммы. Поскольку подпрограмма работает с регистром `dx`, то его исходное содержимое сохраняется в стеке и восстанавливается при выходе.

Уважаемые читатели, попробуйте ответить на вопрос — почему в примере 3.8 перед установкой окна проверяется текущий адрес видеопамяти (`or di, di`), а не размер остатка строки (содержимое регистра `dx`)?

Давайте посчитаем, чего мы добились. Если прямая содержит `N` точек и полностью помещается в текущем окне, то при ее построении по подпрограммам примера 3.6 будет выполнено `4N` команд, а при построении по подпрограмме 3.8 всего 10 команд (не считая `ret`). Одной из них является

команда `rep stosb`, которая записывает `N` байтов в видеопамять. От нее зависит время, затрачиваемое на рисование линии. Можно считать, что мы сократили это время, по крайней мере, в 4 раза по сравнению с примером 3.6 и это вполне оправдывает увеличение размера подпрограммы примера 3.8.

Дополнительные возможности ускорения. Для дальнейшего ускорения процесса рисования в микропрограммном цикле нужно записывать коды сразу двух или четырех точек.

Для записи точек парами вместо команды `rep stosb` надо использовать команду `rep stosw`, предварительно уменьшив содержимое регистра `cx` в два раза путем сдвига на 1 разряд вправо. Если в регистре `cx` находится нечетное число, то при таком сдвиге младшая единица кода попадет в С-разряд регистра флагов (признак переполнения). Следовательно, после сдвига надо проверить состояние С-разряда и записать дополнительную точку, если он установлен. Таким образом, для сокращения цикла записи в два раза в примере 3.8 *каждую команду* `rep stosb` надо заменить следующей группой команд (см. пример 3.9).

Пример 3.9. Замена команды rep stosb на rep stosw

```
      shr     cx, 01   ; уменьшаем количество точек в два раза
      jnc     @F       ; -> обход следующей команды (stosb)
      stosb            ; запись дополнительной точки
@@:   rep     stosw    ; основной цикл записи по два байта
```

Для записи кодов четырех точек при каждом обращении к видеопамяти нужно использовать команду `rep stosd`, предварительно уменьшив содержимое регистра `cx` в четыре раза. В тех случаях, когда содержимое `cx` не кратно четырем, надо дополнительно рисовать 1, 2 или 3 точки. Для упрощения выполняемых действий содержимое `cx` изменяется в два приема. Сначала оно уменьшается в два раза, и если получен признак переноса, то рисуется одна дополнительная точка. Затем оно повторно уменьшается в два раза и если опять получен признак переноса, то рисуются две дополнительные точки. После этого можно использовать команду `rep stosd`. Способ выполнения этих действий показан в примере 3.10.

Пример 3.10. Замена команды rep stosb на rep stosd

```
      shr   cx, 01     ; уменьшаем количество точек в два раза
      jnc   @F         ; -> обход следующей команды (stosb)
      stosb            ; запись дополнительной точки
@@:   shr   cx, 01     ; уменьшаем количество точек в два раза
      jnc   @F         ; -> обход следующей команды (stosw)
      stosw            ; запись двух дополнительных точек
@@:   rep   stosd      ; основной цикл записи по четыре байта
```

Важно

При использовании ускоренных вариантов рисования код цвета точки надо записать в оба байта регистра `ax` или в четыре байта регистра `eax`.

Таким образом, процесс рисования горизонтальной прямой можно дополнительно ускорить примерно в два или четыре раза, но при этом текст примера 3.8 увеличится на 6 или 12 команд. Поэтому в каждом конкретном случае вам придется решать, что важнее, размер подпрограммы или время рисования.

Советуем вам составить вариант примера 3.8 для ускоренного рисования горизонтальной прямой в обратном направлении. Все, что надо при этом учесть, уже описано в данной главе.

Замечание

Описанные подпрограммы рисуют одноцветные линии. Разноцветная линия — это уже рисунок. Для того чтобы ее точки (или группы точек) имели разные цвета, недостаточно простого изменения их кодов в процессе рисования. Должна быть подготовлена и установлена палитра цветов, соответствующих кодам точек. Обо всем этом речь пойдет в третьем разделе данной главы и в главе 4.

Рисование гладких линий. Гладкие линии не содержат ступенек, они могут быть горизонтальными, вертикальными или наклонными под углом, кратным 45 градусам. При их построении адреса смежных точек отличаются на некоторую постоянную величину. Например, у вертикальных линий ее модуль равен значению переменной `Horsize`.

В примере 3.11 приведен текст подпрограммы, которая рисует гладкие линии, при условии, что адреса их точек монотонно возрастают. Перед ее вызовом должно быть установлено окно видеопамяти, в котором расположена опорная точка, а ее адрес указан в регистре `di`. Количество выводимых точек (длина линии) и их коды (цвета) помещаются, соответственно, в регистры `cx` и `al`. Кроме того, в регистр `bx` записывается приращение адреса каждой точки.

Пример 3.11. Подпрограмма для рисования гладких линий

```
anyline: mov   es:[di], al ; запись кода точки в видеобуфер
         add   di, bx      ; коррекция текущего адреса
         jnc   @F          ; -> адрес в пределах окна
         call  NxtWin      ; установка следующего окна
@@:      loop  anyline     ; управление повторами цикла
         ret               ; возврат из подпрограммы
```

При рисовании гладких линий использовать операцию `stosb` для записи кодов точек в видеопамять нецелесообразно.

Давайте разберемся, что именно можно нарисовать с помощью подпрограммы 3.11. Обозначим константу переадресации, значение которой записывается в регистр `bx`, буквой `k` и условимся, что она может быть только положительным числом. Вспомните табл. 3.3 — положительные приращения адресов смежных точек могут иметь четыре значения: `1`, `Horsize`, `Horsize+1` и `Horsize-1`. Если в качестве константы `k` использовать эти значения, то соответственно будут нарисованы горизонтальная, вертикальная и две наклонные прямые. Последние являются диагоналями квадрата, сторона которого содержит количество точек, указанное в регистре `cx`.

Подпрограмма 3.11 позволяет рисовать пунктирные линии. Например, если задавать `k=2`, `k=2*Horsize`, `k=2*(Horsize+1)` и `k=2*(Horsize-1)`, то будут нарисованы пунктирные линии, у которых расстояние между соседними точками равно 2. Однако такая возможность является побочным эффектом и не представляет особого интереса.

Если вам часто приходится рисовать вертикальные линии, то сделайте копию примера 3.11, замените в ней команду `add di, bx` на `add di, Horsize` и подберите подходящее имя подпрограммы (не забудьте указать его в команде `loop`).

Произвольные линии. При рисовании произвольных линий, в отличие от гладких, вычисляется приращение адреса в каждой точке. В этом случае при перемещении от точки к точке значение одной координаты (`X` или `Y`) увеличивается на 1, а значение другой либо увеличивается на 1, либо остается неизменным. В результате на экране возникают ступеньки, размер и количество которых, при прочих равных условиях, зависят от угла наклона.

Рассмотрим, какие действия выполняют подпрограммы, предназначенные для рисования линии, проходящей через две заданные точки. При ее вызове задаются координаты двух крайних точек линии `X1,Y1` и `X2,Y2`. Выполнение подпрограмм начинается с анализа значений координат. Если окажется, что `X1 > X2`, то производится перестановка значений координат в памяти так, чтобы `X2 > X1`. Затем начало координат переносится в точку `X1,Y1`, что позволит работать с приращениями адресов относительно этой точки. При выполнении дальнейших действий учитываются следующие величины:

```
Dx = X2 — X1; Dy = Y2 — Y1
```

Если `Dx` = 0 или `Dy` = 0, то задана вертикальная или горизонтальная линия, которая строится обычным способом, описанным в данной главе. В противном случае выбирается способ построения линии.

Если `Dx > Dy`, то линия строится относительно оси `X`. Это значит, что при переходе от точки к точке приращение значения координаты `X` равно 1, а приращение координаты `Y` равно `Dy/Dx`, причем `0 < Dy/Dx < 1`.

Если `Dx < Dy`, то линия строится относительно оси `Y`. Это значит, что при переходе от точки к точке приращение значения координаты `Y` равно 1, а приращение координаты `X` равно `Dx/Dy`, причем `0 < Dx/Dy < 1`.

При Dx = Dy линия гладкая, например, с углом наклона 45 градусов, способ ее построения выбирается разработчиком.

Линия рисуется на экране последовательно точка за точкой, поэтому значение координат в любой точке равно *сумме приращений* их значений во всех предыдущих и в данной точке. В одних случаях эти суммы вычисляются явно, в других неявно, но без них линию нарисовать невозможно.

Значения одной координаты (X или Y) являются целыми числами и не требуют дополнительных преобразований. Значения другой координаты (Y или X) могут иметь дробную часть, поэтому их надо преобразовать в целые числа. Описанные в литературе подпрограммы различаются, главным образом, способом вычисления ступенчатой функции, аппроксимирующей прямую, проходящую через две заданные точки. В простейшем случае при вычислении приращения адреса учитывается только целая часть числа, а при суммировании приращений — все число.

Приращения значений координат преобразуются в приращение адреса, и очередная точка выводится на экран. Знак приращения зависит от взаимного расположения крайних точек линии, поэтому в подпрограммах учитывается возможность как положительных, так и отрицательных приращений адресов.

Описание алгоритмов рисования линий и примеры подпрограмм, правда, предназначенных для работы в видеорежимах VGA, вы найдете в книге [8]. Ее оригинал (на английском языке) распространялся на компакт-дисках вместе с текстами подпрограмм.

3.2.2. Прямоугольники

При рисовании прямоугольников можно преследовать две разные цели — закрашивание (заливка) прямоугольной области экрана выбранным цветом или рисование сторон (контура) прямоугольника. Первая задача является более общей, поэтому сначала мы рассмотрим способы ее решения.

Полоса заданного цвета. Предположим, что ширина прямоугольника равна ширине рабочей области экрана (Horsize), а ее высота (толщина) составляет N точек. В примере 3.12 приведен текст подпрограммы, которая последовательно рисует заданное количество горизонтальных линий длиной Horsize, в результате чего получается полоса нужной высоты.

Перед вызовом подпрограммы надо установить окно видеопамяти, в котором находится левый верхний угол полосы, а его адрес в этом окне записать в регистр di. Количество строк в полосе указывается в регистре cx. Для рисования строк подходит любой вариант подпрограммы horline, описанный в предыдущем разделе. В зависимости от того, какой из них вы выберете, код цвета указывается только в регистре al, в обоих байтах регистра ax или в четырех байтах регистра eax.

Пример 3.12. Закрашивание прямоугольной полосы

```
stripe:  PushReg <di,cx,Cur_win> ; сохранение di, cx и Cur_win
fillbar: push  cx                ; сохранение текущего значения cx
         mov   cx, horsize       ; задание размера строки
         call  horline           ; вывод очередной строки
         pop   cx                ; восстановление счетчика строк
         loop  fillbar           ; управление выводом строк
         PopReg <Cur_win,cx,di>  ; восстановление Cur_win, cx и di
         call  SetWin            ; восстановление исходного окна
         ret                     ; возврат из подпрограммы
```

Выполнение подпрограммы примера 3.12 начинается с сохранения в стеке содержимого регистров `di`, `cx` и переменной `Cur_win`. Закрашивание производится в цикле, имеющем метку `fillbar`. Регистр `cx` используется в этом цикле в качестве счетчика. Кроме того, в нем передается размер строки для подпрограммы `horline`. Поэтому в начале цикла содержимое `cx` сохраняется в стеке и восстанавливается после возвращения из `horline`. Благодаря этому команда `loop fillbar` работает с той величиной, которая была указана в регистре `cx` при обращении к подпрограмме `stripe`.

После завершения цикла `fillbar` восстанавливаются сохраненные в стеке величины и исходное окно видеопамяти, для чего вызывается подпрограмма `SetWin`. Последняя команда выполняет возврат из подпрограммы.

В рассмотренном примере ширина полосы равнялась ширине рабочей области экрана. В таком случае после записи последней точки текущей строки автоматически происходит переход к началу следующей строки, для этого не нужны никакие дополнительные действия подпрограммы. Если же ширина прямоугольной области меньше `Horsize`, то адрес начала следующей строки должна устанавливать подпрограмма.

Вычисление адресов строк. Первые точки строк прямоугольной области находятся в одном столбце, поэтому в режимах `PPG` адреса начала смежных строк различаются на величину `Horsize` (см. табл. 3.3). Следовательно, если адрес начала текущей строки увеличить на `Horsize`, то получится адрес начала следующей строки. Обсудим, как это проще всего сделать.

Наиболее очевидный способ — сохранять адрес начала текущей строки и в нужный момент увеличивать его на `Horsize`. Однако это не самый простой способ, и вот почему. Если при вычислении адреса начала следующей строки происходит переполнение, то надо установить новое окно видеопамяти. Но оно уже могло быть изменено при рисовании текущей строки и повторное изменение недопустимо. Поэтому, прежде чем устанавливать новое окно, надо проверить, не изменилось ли оно при рисовании строки. Таким образом, простой на первый взгляд способ коррекции адреса плох тем, что

неизвестно, какому окну видеопамяти соответствует сохраненный адрес и нужны дополнительные проверки для выяснения этого обстоятельства.

Лишние проверки можно исключить, если изменить способ коррекции так, чтобы использовалось текущее значение адреса, которое заведомо соответствует установленному окну видеопамяти. После вывода последней точки строки текущий адрес больше адреса ее первой точки на ширину прямоугольной области. Следовательно, прибавив к нему значение переменной `Horsize`, уменьшенное на ширину прямоугольной области, мы получим адрес начала следующей строки. Если при сложении происходит переполнение, то устанавливается следующее окно видеопамяти без проверки каких-либо дополнительных условий. Мы будем использовать такой способ вычислений при построении прямоугольных рисунков различного назначения.

Закрашивание прямоугольной области. В примере 3.13 приведена подпрограмма, закрашивающая заданным цветом прямоугольную область произвольного размера. Перед ее вызовом адрес левого верхнего угла прямоугольника помещается в регистр `di` и устанавливается окно видеопамяти, которому принадлежит этот адрес. Ширина прямоугольника (количество точек в строке) помещается в регистр `dx`, а высота (количество строк или точек по вертикали) — в регистр `cx`. Задание кода цвета точек зависит от того, какой вариант подпрограммы `horline` вы будете использовать. `Horline` может записывать в видеопамять байты, слова или двойные слова (см. раздел 3.2.1). Соответственно один и тот же код цвета указывается в регистре `al`, в обоих байтах регистра `ax` или в четырех байтах регистра `eax`.

Пример 3.13. Подпрограмма закрашивания прямоугольной области

```
rctngl:  PushReg <bx,cx,di,Cur_win> ; сохранение в стеке
         mov   bx, horsize          ; копируем horsize в регистр bx
         sub   bx, dx               ; и вычитаем ширину прямоугольника
fillar:  push  cx                   ; сохранение счетчика повторов
         mov   cx, dx               ; задание размера строки
         call  horline              ; рисуем линию
         pop   cx                   ; восстановление счетчика повторов
         add   di, bx               ; адрес начала следующей строки
         jnc   @F                   ; -> адрес в пределах окна
         call  NxtWin               ; переход к следующему окну
@@:      loop  fillar               ; управление повторами цикла
         PopReg <Cur_win,di, cx,bx> ; восстановление из стека
         call  setwin               ; восстановление исходного окна
         ret                        ; возврат из подпрограммы
```

Выполнение примера 3.13 начинается с сохранения в стеке содержимого используемых регистров и переменной `Cur_win`. После этого с помощью двух команд в регистре `bx` формируется константа для коррекции адресов

строк. Как говорилось выше, она равна разности между значением переменной `Horsize` и шириной прямоугольника, указанной в `dx` перед вызовом подпрограммы.

Закрашивание прямоугольной области производится в цикле, имеющем метку `fillar`. Он отличается от аналогичного цикла `fillbar` примера 3.12 тем, что после рисования каждой строки производится коррекция текущего адреса (команда `add di, bx`). Если при сложении не происходит переполнение результата, то новое значение адреса находится в пределах сегмента и команда `jnc @F` исключает вызов процедуры `NxtWin`. В случае переполнения условный переход не выполняется и происходит установка следующего окна.

После выхода из цикла `fillar`, перед возвратом в вызывающий модуль, восстанавливаются сохраненные в стеке величины и исходное окно видеопамяти.

Рисование контура прямоугольника. Контур прямоугольника состоит из двух горизонтальных и двух вертикальных линий, поэтому для его рисования понадобятся подпрограммы `horline` и `anyline`, описанные в предыдущем разделе. Прежде чем рассматривать подпрограмму, обсудим, как можно нарисовать контур прямоугольника с минимальными затратами на вычисление адресов начала его граней (сторон).

Будем считать, что опорной точкой является верхний левый угол контура, адрес которого известен. Если грани прямоугольника рисовать, например, в таком порядке — верхняя, правая, нижняя, левая, то вычислять адреса начала граней вообще не потребуется. Однако в таком случае понадобятся не две, а четыре подпрограммы. При рисовании верхней и правой граней адреса видеопамяти будут изменяться в естественном порядке в сторону их увеличения и можно использовать подпрограммы `horline` и `anyline`. При рисовании же нижней и левой граней адреса видеопамяти будут изменяться в сторону их уменьшения и понадобятся еще две подпрограммы, рисующие линии в обратном направлении. Чтобы ограничиться двумя подпрограммами, изменим последовательность рисования граней.

Сначала рисуем верхнюю и правую грани, затем возвращаемся в левый верхний угол контура прямоугольника и рисуем левую и нижнюю грани. Можно сначала нарисовать левую и нижнюю грани, а затем верхнюю и правую. В обоих случаях при рисовании граней адреса видеопамяти изменяются в сторону их увеличения.

В примере 3.14 приведена подпрограмма, рисующая сначала верхнюю и правую, а затем левую и нижнюю грани. Входные параметры для нее совпадают с параметрами подпрограммы примера 3.13.

Пример 3.14. Подпрограмма рисования контура прямоугольника

```
round: PushReg <bx,Cur_win,cx,di>  ; сохранение исходного состояния
       mov    cx, dx               ; cx = ширина прямоугольника
```

```
dec     cx                      ; уменьшаем  ширину на 1
call    horline                 ; рисуем верхнюю грань
mov     bx, horsize             ; bx = horsize
pop     cx                      ; восстанавливаем содержимое cx
push    cx                      ; и сохраняем его в стеке
call    anyline                 ; рисуем правую грань
PopReg <di,cx,Cur_win>          ; восстанавливаем исходное состояние
PushReg <Cur_win,cx,di >        ; и вновь запоминаем его
call    SetWin                  ; устанавливаем исходное окно
dec     cx                      ; уменьшаем высоту на 1
call    anyline                 ; рисуем левую грань
mov     cx, dx                  ; cx = ширина прямоугольника
call    horline                 ; рисуем нижнюю грань
PopReg <di,cx,Cur_win,bx >      ; восстановление исходного состояния
call    SetWin                  ; восстановление исходного окна
ret                             ; возврат из подпрограммы
```

Выполнение примера 3.14 начинается с сохранения в стеке переменной `Cur_win` и регистров `bx`, `cx` и `di`. При вызове подпрограммы переменная `Cur_win` и регистр `di` задают адрес левого верхнего угла контура прямоугольника, а в регистре `cx` указывается высота прямоугольника (количество точек по вертикали).

При рисовании верхней грани ее размер сокращается на 1 точку, для того чтобы при возврате из подпрограммы `horline` в регистре `di` находился адрес первой точки правой грани. Правую грань рисует подпрограмма `anyline`, поэтому в регистр `bx` надо записать значение `Horsize`, а из стека восстановить и тут же снова сохранить в нем содержимое регистра `cx`. После возврата из подпрограммы `anyline` будут нарисованы верхняя и правая грани.

Теперь надо вернуться в левый верхний угол контура прямоугольника, восстановив исходное состояние, сохраненное в стеке, и заново сохранить его для использования при выходе их подпрограммы. Кроме того, восстанавливается исходное окно (команда `call SetWin`), поскольку оно могло измениться при рисовании.

При рисовании левой грани ее размер сокращается на 1, благодаря этому при возврате из подпрограммы `anyline` регистр `di` содержит адрес первой точки нижней грани. В данном случае записывать в регистр `bx` значение `Horsize` не требуется, поскольку оно было записано туда раньше. После возврата из подпрограммы `anyline` в регистре `cx` указывается ширина прямоугольника, и подпрограмма `horline` рисует нижнюю замыкающую грань. Остается восстановить сохраненные величины, исходное окно видеопамяти и выполнить возврат из подпрограммы.

Описанная подпрограмма рисует прямоугольный контур, ширина граней которого равна одной точке. Если грани должны быть более широкими, то можно нарисовать несколько вложенных прямоугольных контуров так, чтобы получить грани нужной ширины. Можно поступить иначе — нарисовать

две вложенные прямоугольные области. Сначала рисуется большая область, цвет которой совпадает с цветом граней, а затем в ней меньшая область, имеющая цвет внутренней части прямоугольника.

Мы закончили рассмотрение способов рисования простых геометрических фигур. Если читателя интересуют способы рисования более сложных фигур, то рекомендуем обратиться к книге [8], в которой приведены некоторые алгоритмы и их теоретическое обоснование.

3.3. Построение рисунков

В отличие от геометрических фигур рисунки не создаются в процессе выполнения задачи, а готовятся заранее и хранятся в файлах, на внешних носителях. К сожалению (или к счастью), не существует единого стандарта структуры таких файлов, но существуют специальные программы для их преобразования из одного стандарта (формата) в другой. Кроме того, такое преобразование выполняют все распространенные графические редакторы. Поэтому вы можете выбрать один из стандартов и использовать только его. В приложении А данной книги подробно описан один из основных стандартов — BMP. По мере изложения основного материала будут рассмотрены некоторые характеристики и других наиболее распространенных стандартов.

В структуре файла, содержащего точечный рисунок, можно выделить три основные части: заголовок, палитру и образ рисунка.

Заголовок располагается в начале файла и содержит исчерпывающую информацию, необходимую для вывода рисунка на экран или на печать.

Палитра находится после заголовка или после образа рисунка. Она содержит коды использованных в рисунке цветов. Описанию назначения палитры и способов работы с ней посвящена следующая глава книги.

Образ рисунка содержит коды точек, образующих рисунок. Адрес его начала (смещение) в файле обычно указывается в заголовке. В некоторых случаях перед построением или в процессе построения рисунка может потребоваться преобразование его образа.

Прежде чем выводить рисунок на экран, весь файл или его часть надо прочитать в оперативную память, выделить из заголовка все необходимые величины и установить палитру используемых цветов. Мы будем считать, что все подготовительные действия выполнены, и образ рисунка находится в оперативной памяти. Такое допущение позволит рассматривать графические аспекты построения рисунков в чистом виде.

3.3.1. Варианты построения строк

Точечные рисунки, независимо от их конкретного содержания, всегда занимают на экране прямоугольную область, а их образы хранятся в файлах в виде последовательности строк одинакового размера. Поэтому построение

рисунка сводится к последовательному построению его строк на экране монитора. Изложение материала мы начнем с нескольких примеров подпрограмм для работы со строками рисунков.

Строки рисунков отличаются от линий геометрических фигур тем, что их образы существуют в оперативной или в видеопамяти. Поэтому в простых случаях надо просто переместить коды точек из одного места памяти в другое. В данной главе нас будут интересовать строки, у которых код точки занимает 1 байт.

Построение строки слева направо. В примере 3.15 приведен текст подпрограммы, выполняющей копирование образа строки из оперативной памяти в видеопамять. В результате на экране появится изображение строки. Копирование производится в прямом направлении, т. е. в сторону увеличения адресов.

Перед обращением к подпрограмме устанавливается окно видеопамяти, в котором должны располагаться точки строки, а в регистре `di` указывается адрес первой (левой) точки. Кроме того, пара регистров `fs:si` должна содержать адрес начала строки в оперативной памяти, `fs` — сегмент, а `si` — смещение (относительный адрес) в этом сегменте. Размер строки (количество точек) помещается в регистр `cx`. Напомним, что `es` должен содержать код видеосегмента (значение переменной `Vbuff`).

Пример 3.15. Построение строки 256-цветного рисунка

```
drawline: movs byte ptr [di], fs:[si] ; запись кода точки в видеобуфер
          or   di, di          ; начало нового сегмента ?
          jne  @F              ; -> нет, обход команды call NxtWin
          call NxtWin          ; установка следующего окна
@@:       loop drawline        ; управление повторами цикла
          ret                  ; возврат из подпрограммы
```

Сравните второй вариант примера 3.6 и пример 3.15. Приведенные в них тексты различаются только первой командой. Вместо `stosb`, записывающей в видеопамять содержимое регистра `al`, в данном случае используется строковая операция `movs`, копирующая в видеопамять байты оперативной памяти. Обратите внимание на то, что в записи строковой операции сегментный регистр приемника (`es`) указывать не обязательно, а сегментный регистр источника (`fs`) вы можете изменить по своему усмотрению.

Подпрограмма примера 3.15 выводит точки в порядке увеличения их адресов. В некоторых случаях может возникнуть необходимость выводить точки в обратном порядке (справа налево) в сторону уменьшения адресов. Как можно нарисовать линию в направлении справа налево, было показано во втором варианте примера 3.7. Если в этом примере команду `lodsb` заменить командой `movs byte ptr [di], fs:[si]`, то подпрограмма будет строить

строку рисунка в обратном направлении. Когда и зачем может понадобиться такая замена, мы обсудим при описании построения рисунков.

Ускорение цикла построения. Для ускорения построения строки используется микропрограммный цикл пересылки, аналогичный описанному в примере 3.8. Мы просто перепишем этот пример, изменив в нем две строковые операции (пример 3.16). Входные параметры указываются так же, как для подпрограммы примера 3.15.

Пример 3.16. Ускоренное построение строки рисунка

```
drawline: push  dx          ; сохранение содержимого регистра dx
          mov   dx, di      ; копирование адреса в регистр dx
          add   dx, cx      ; сумма текущего адреса и количества точек
          jc    @F          ; -> прямая расположена в двух окнах
          xor   dx, dx      ; очистка регистра dx
@@:       sub   cx, dx      ; вычисляем количество точек в текущем окне
          rep   movs byte ptr [di], fs:[si]; строим строку или ее часть
          or    di, di      ; адрес в пределах текущего окна ?
          jne   dhl_out     ; -> да, строка построена полностью
          call  NxtWin      ; установка следующего окна
          mov   cx, dx      ; количество не построенных точек
          rep   movs byte ptr [di], fs:[si]; строим остаток строки
dhl_out:  pop   dx          ; восстановление содержимого dx
          ret               ; возврат из подпрограммы
```

Для дальнейшего ускорения выполнения построения строки нужно использовать пересылку одновременно двух или четырех байтов. В примерах 3.9 и 3.10 показаны изменения, которые позволяли это сделать при рисовании линии. Аналогичные изменения надо внести и в пример 3.16, только не забудьте заменить строковые операции следующими:

`stosb` на `movs byte ptr [di], fs:[si]`
`stosw` на `movs word ptr [di], fs:[si]`
`stosd` на `movs dword ptr [di], fs:[si]`

Замечание

При обработке строковых операций пересылки слов или двойных слов с измененным сегментом операнда источника MASM 5.1 формирует правильный код, но выдает предупреждающее сообщение. В последующих версиях MASM эта ошибка устранена.

Следует заметить, что если образ рисунка находится в файле, то время, затрачиваемое на чтение и предварительные действия, значительно больше чистого времени, построения всех его строк. В таком случае целесообразность ускорения построения строк весьма проблематична.

Кроме 256-цветных рисунков довольно широко распространены 16- и 2-цветные. Они используются, например, Windows и ее приложениями для оформления рабочей области экрана и в других целях. Такие рисунки хранятся в упакованном виде и перед записью в видеопамять должны быть распакованы. Способ упаковки является общепринятым и не зависит от конкретного стандарта, в котором подготовлен файл. При распаковке надо учесть, что в зависимости от количества точек в исходной строке последний байт упакованной строки может быть заполнен частично.

Распаковка 16-цветных строк. Код 16-цветного рисунка занимает 4 разряда, и для сокращения размера файла в одном байте располагаются коды двух точек. Перед записью в видеопамять находившиеся в одном байте коды точек надо расположить в младших разрядах двух разных байтов. Эта операция достаточно проста, и ее целесообразно выполнять в процессе построения строки. Тем более что при предварительной распаковке рисунка его размер увеличится в два раза.

Подпрограмма для распаковки строки в процессе построения 16-цветного рисунка приведена в примере 3.17. Перед обращением к ней надо установить окно видеопамяти, в котором должны располагаться точки строки, а адрес первой точки указать в регистре `di`. Пара регистров `fs:si` должна содержать адрес оперативной памяти, начиная с которого хранится упакованная строка. В регистре `cx` указывается количество точек в строке.

Пример 3.17. Подпрограмма построения строки 16-цветного рисунка

```
drwlin4:  mov   al, fs:[si] ; читаем в al код пары точек
          shr   al, 04      ; выделяем код старшей точки
          stosb             ; записываем его в видеопамять
          or    di, di      ; начало нового сегмента ?
          jne   @F          ; -> нет, обход команды call NxtWin
          call  NxtWin      ; установка следующего окна
@@:       lods  byte ptr fs:[si]    ; повторное чтение кода пары точек
          dec   cx          ; cx = cx - 1
          je    dr4ret      ; если cx = 0, то конец строки
          and   al, 0Fh     ; выделяем код младшей точки
          stosb             ; записываем его в видеопамять
          or    di, di      ; начало нового сегмента ?
          jne   @F          ; -> нет, обход команды call NxtWin
          call  NxtWin      ; установка следующего окна
@@:       loop  drwlin4     ; управление повторами цикла
dr4ret:   ret               ; выход из подпрограммы
```

В примере 3.17 сначала выделяется и записывается в видеопамять код старшей тетрады очередного байта упакованной строки, а затем код его младшей

тетрады. После записи кода старшей тетрады содержимое `cx` уменьшается на 1 и проверяется значение результата. Если оно равно нулю, то построение строки завершено. Указанные действия нужны потому, что в последнем байте упакованной строки при нечетном количестве точек в строке будет заполнена только одна (старшая) тетрада.

После каждой записи в видеопамять проверяется значение адреса, хранящегося в регистре `di`. Если он равен нулю, то произошел выход за пределы сегмента и надо изменить текущее окно видеопамяти. Если адрес не равен нулю, то выполнение команды `call NxtWin` исключается.

Последняя команда `loop` управляет повторами цикла, если строка содержит четное количество точек, то именно она завершит выполнение подпрограммы. При нечетном количестве точек в строке выполнение подпрограммы завершит команда `je dr4ret`.

Распаковка 2-цветных строк. Если при построении рисунка использовано только два цвета, например черный и белый, то код точки помещается в одном разряде и может принимать только два значения 0 и 1. В таких случаях для сокращения размеров файла в одном байте располагаются коды восьми точек. В старшем разряде байта находится код первой точки, а в младшем — последней, поэтому выделять коды точек надо начиная со старшего разряда. В зависимости от количества точек в строке последний байт может быть заполнен частично. Не следует считать, что двухцветные рисунки обязательно черно-белые — их цвета зависят от кодов, находящихся в прилагаемой к файлу палитре.

Подпрограмма для распаковки строки в процессе построения 2-цветного рисунка приведена в примере 3.18. Перед ее вызовом устанавливается окно видеопамяти, в котором должна располагаться строящаяся строка, а адрес первой точки помещается в регистр `di`. Пара регистров `fs:si` должна содержать адрес оперативной памяти, начиная с которого хранится упакованная строка. В регистре `cx` указывается количество точек в строке.

Пример 3.18. Подпрограмма построения строки 2-цветного рисунка

```
drwlin1:  push  bx                ; сохраняем содержимое bp
          mov   bx, cx            ; bp = cx (количество точек в строке)
lpdrwl1:  lods  byte ptr fs:[si]  ; читаем в al код очередного байта
          mov   ah, al            ; копируем коды из al в ah
          mov   cx, 08            ; количество повторов цикла распаковки
out8pnt:  xor   al, al            ; очищаем регистр al
          shl   ah, 01            ; сдвигаем ah на разряд влево
          adc   al, 00            ; прибавляем переполнение к al
          stosb                   ; записываем код очередной точки
          or    di, di            ; начало нового сегмента ?
```

```
          jne   @F                ; -> нет, обход команды call NxtWin
          call  NxtWin            ; установка следующего окна
@@:       dec   bx                ; bx = bx - 1
          je    dr1ret            ; если bx = 0, то строка построена
          loop  out8pnt           ; управление внутренним циклом
          jmp   short lpdrwl1     ; -> на обработку следующего байта
dr1ret:   pop   bx                ; восстановление содержимого bx
          ret                     ; выход из подпрограммы
```

Подпрограмма примера 3.18 представляет собой два вложенных цикла. Имя внешнего цикла `lpdrwl1`, а внутреннего — `out8pnt`.

Внешний цикл считывает очередной байт образа строки, копирует его в регистр `ah` и задает количество повторов внутреннего цикла.

Во внутреннем цикле производится распаковка очередной группы точек и запись их кодов в видеопамять. Распаковку выполняют три первые команды внутреннего цикла. Первая из них очищает регистр `al`, вторая сдвигает содержимое регистра `ah` на разряд влево. При сдвиге старший разряд регистра `ah` переносится в С-разряд регистра флагов, поэтому если он содержал единицу, то вырабатывается признак переполнения. Третья команда (`adc al, 00`) прибавляет содержимое С-разряда к регистру `al`. В результате, в зависимости от кода очередной точки, в регистре `al` окажется 0 или 1. Полученный код точки команда `stosb` записывает в видеопамять. Затем проверяется текущий адрес видеопамяти и при необходимости устанавливается следующее окно видеопамяти.

После записи каждой точки содержимое регистра `bx` уменьшается на 1 и если оно окажется равным нулю, то происходит переход на метку `dr1ret` для завершения подпрограммы. В противном случае команда `loop` управляет выводом восьми точек. После этого происходит короткий безусловный переход на начало внешнего цикла для обработки следующего байта.

Чтение строки из видеопамяти. Во всех описанных выше подпрограммах производилось копирование содержимого оперативной памяти в видеопамять. На практике сравнительно часто приходится решать обратную задачу, т. е. копировать содержимое видеопамяти в оперативную память. Например, это может понадобиться для сохранения исходного фона перед построением рисунка. При работе с Windows и ее приложениями вы наверняка видели различные варианты меню, информационные строки, диалоговые окна и другие картинки, которые временно появляются на экране, а после своего исчезновения не оставляют никаких следов. Это достигается за счет сохранения и последующего восстановления исходного фона участка, временно используемого в других целях.

В примере 3.19 приведена подпрограмма, выполняющая копирование строки из видеопамяти в оперативную память. При ее вызове адреса задаются

так же, как во всех предыдущих примерах, а именно, пара регистров `es:di` содержит адрес видеопамяти, а пара `fs:si` — адрес оперативной памяти. Предварительно устанавливается окно, в котором расположено начало копируемой строки, а ее размер указывается в регистре `cx`.

Пример 3.19. Копирование строки из видеопамяти в ОЗУ

```
readlin:  mov   al, es:[di]  ; чтение очередного байта видеопамяти
          mov   fs:[si], al  ; запись кода точки в ОЗУ
          inc   si           ; увеличение адреса ОЗУ
          inc   di           ; увеличение адреса видеопамяти
          jne   @F           ; -> адрес в пределах окна
          call  Nxtwin       ; переход к следующему окну
@@:       loop  readlin      ; управление внутренним циклом
          ret                ; выход из подпрограммы
```

В примере 3.19 использованы обычные команды пересылки, поэтому очередной байт сначала считывается из видеопамяти в регистр `al`, а затем содержимое `al` копируется в оперативную память. После этого содержимое регистров `si` и `di` увеличивается на 1 и проверяется значение нового адреса видеопамяти. Если он окажется равным нулю, то выполняется команда `call NxtWin`, в результате чего устанавливается следующее окно видеопамяти. Команда `loop readlin` повторяет выполнение цикла до тех пор, пока не будут скопированы все байты строки.

В рассмотренном варианте подпрограммы, если не происходит смена окна, то при пересылке одного байта выполняется 6 команд. Такое количество вспомогательных действий существенно замедляет пересылку, что будет особенно ощутимо при сохранении больших объемов видеопамяти. Для сокращения вспомогательных действий пересылку нужно выполнять с помощью строковой операции `movs`.

Улучшение цикла копирования. У операции `movs` фиксировано назначение индексных регистров `di` и `si` и сегментного регистра `es`. Поэтому для применения строковой операции надо изменить расположение адресов источника и приемника. Пара регистров `fs:si` должна содержать адрес видеопамяти, а пара `es:di` — адрес оперативной памяти, но для удобства лучше сохранить единообразный способ расположения адресов и временно изменять его в самой подпрограмме пересылки.

В примере 3.20 показано, как можно переставлять адреса источника и приемника в теле подпрограммы на время выполнения цикла пересылки. При обращении к подпрограмме этого примера регистры `es:di`, как обычно, должны содержать адрес видеопамяти, а регистры `fs:si` — адрес оперативной памяти.

Пример 3.20. Копирование строки из видеопамяти в оперативную память

```
;         Перестановка входных параметров
readlin:  push  fs          ; сохраняем содержимое fs
          pop   es          ; выталкиваем его в es
          mov   fs, Vbuff   ; fs = Vbuff (код видеосегмента)
          xchg  di, si      ; перестановка содержимого di и si
;         Цикл копирования строки из видеопамяти в оперативную память
readlp:   movs  byte ptr [di], fs:[si]; копирование очередного байта
          or    si, si      ; адрес в пределах сегмента ?
          jne   @F          ; -> да, обход команды call NxtWin
          call  NxtWin      ; установка следующего окна
@@:       loop  readlp      ; управление повторами цикла
;         Восстановление исходного расположения входных параметров
          push  es          ; сохраняем содержимое es
          pop   fs          ; сохраняем содержимое fs
          mov   es, Vbuff   ; выталкиваем содержимое fs в es
          xchg  di, si      ; перестановка содержимого di и si
          ret               ; возврат из подпрограммы
```

В примере 3.20 перед выполнением цикла пересылки содержимое регистров `fs` копируется в регистры `es` через стек, в `fs` помещается код видеосегмента (содержимое переменной `Vbuff`) и переставляется содержимое индексных регистров `di` и `si`. Так получаются нужные адреса источника и приемника.

Цикл пересылки имеет метку `readlp`, он отличается от аналогичного цикла примера 3.15 только тем, что вместо команды `or di, di` используется `or si, si`, поэтому мы не будем повторять его описание. После пересылки восстанавливается исходное расположение входных параметров в сегментных и индексных регистрах и происходит выход из подпрограммы.

Что дает улучшение цикла. В примере 3.20 цикл `readlp` содержит на две команды меньше, чем цикл подпрограммы примера 3.19, т. е. количество вспомогательных команд сократилось на третью часть. На первый взгляд, это немного, но появилась возможность дальнейшего ускорения процесса копирования за счет использования микропрограммного цикла пересылки. Для этого применяется способ, показанный в примере 3.16, и варианты его дополнительного ускорения, описанные в пояснениях к этому примеру.

Выше подчеркивалось, что если рисунок воспроизводится из файла, то проблема ускорения записи в видеопамять не столь актуальна. Однако если рисунок сохраняется в оперативной памяти или восстанавливается из образа, сохраненного в памяти, то от времени, затрачиваемого на копирование из одного вида памяти в другой, зависит быстродействие вашей задачи. В таком случае имеет смысл увеличивать размер подпрограммы для ускорения ее работы с видеопамятью.

Теперь мы располагаем, хотя и не полным, но вполне достаточным набором подпрограмм и это позволяет перейти к рассмотрению способов построения завершенных рисунков.

3.3.2. Воспроизведение не сжатых рисунков

Строки образа рисунка могут храниться в файле в прямом или обратном порядке. В первом случае они расположены по возрастанию номеров, т. е. сначала в файле записаны точки первой строки, затем второй и так вплоть до последней. Во втором случае они расположены по убыванию номеров, т. е. сначала в файле записаны точки последней строки, затем предпоследней и так до первой строки. Первый способ хранения образа рисунка применяется, например, в файлах, соответствующих стандарту `PCX`, а второй — в файлах, соответствующих стандарту `BMP`. Распознать принадлежность файла к этим стандартам можно по их типу (расширению), который совпадает с названием стандарта. Например, файл `leaves.bmp` подготовлен в стандарте `BMP`.

Последовательность расположения строк в файле определяет логику работы с их адресами при воспроизведении образа рисунка на экране. В данном разделе мы рассмотрим построение рисунков, у которых строки расположены в файле, или в оперативной памяти, в естественном порядке. При этом работа с адресами наиболее проста. Такой порядок расположения строк используется не только в большинстве стандартов, но и при сохранении в оперативной памяти или восстановлении из нее содержимого видеопамяти.

Построение рисунка небольшого размера. Важной характеристикой, влияющей на выбор варианта построения рисунка, является его размер. В первую очередь нас будут интересовать такие рисунки, образ которых помещается в одном сегменте оперативной памяти, т. е. их размер не превышает 65 536 байт. Этому требованию удовлетворяет большинство рисунков, предназначенных для оформления рабочей области экрана. В частности, стандартные пиктограммы занимают на экране квадрат размером 32×32 точки.

Текст подпрограммы, выполняющей построение рисунка, образ которого целиком помещается в одном сегменте оперативной памяти, а строки расположены в естественном порядке, приведен в примере 3.21. Перед обращением к подпрограмме должно быть установлено окно видеопамяти, содержащее левый верхний угол рисунка, а адрес этого угла указан в регистрах `es:di`. Адрес начала образа рисунка в оперативной памяти задает пара `fs:si`. В регистрах `dx` и `cx` указываются ширина и высота рисунка.

Пример 3.21. Работа с прямоугольной областью небольшого размера

```
draw:   PushReg <di,si,cx,bx,Cur_win>; сохранение исходных величин
        mov     bx, horsize  ; копируем в bx значение horsize
        sub     bx, dx       ; и вычитаем из него ширину рисунка
```

```
drwout: push    cx          ; сохраняем счетчик повторов
        mov     cx, dx      ; задаем размер строки рисунка
        call    drawline    ; !! или call bp, пояснения в тексте
        pop     cx          ; восстанавливаем счетчик повторов
        add     di, bx      ; адрес начала следующей строки
        jnc     @F          ; -> адрес в пределах сегмента
        call    NxtWin      ; установка следующего окна
@@:     loop    drwout      ; управление повторами цикла
        PopReg <Cur_win,bx,cx,si,di>; восстановление исходных величин
        call    setwin      ; восстановление исходного окна
        ret                 ; возврат из подпрограммы
```

Построение рисунка отличается от закрашивания прямоугольной области тем, что код каждой выводимой точки выбирается из оперативной памяти, а не их регистра-аккумулятора. Поэтому тексты примеров 3.13 и 3.21 различаются только именем подпрограммы, которая вызывается в цикле построения: `horline` в примере 3.13 и `drawline` в данном случае.

Выполнение примера 3.21 начинается с сохранения в стеке тех величин, которые могут измениться в процессе построения. Затем две команды формируют константу для переадресации строк. Ее назначение обсуждалось в разделе 3.2.2 перед описанием примера 3.13.

Цикл построения рисунка имеет имя `drwout`. Он начинается с сохранения в стеке значения счетчика повторов (регистра `cx`) и записи в него размера строки. Затем происходит вызов подпрограммы `drawline` для вывода на экран очередной строки рисунка. После возврата из подпрограммы восстанавливается сохраненное в стеке значение счетчика повторов и вычисляется адрес начала следующей строки. Если при сложении будет получен признак переполнения, то произойдет обращение к подпрограмме `NxtWin` для установки следующего окна видеопамяти. Последняя команда цикла `loop` повторяет его выполнение до тех пор, пока не будут построены все строки рисунка.

После выхода из цикла восстанавливаются сохраненные в стеке величины, исходное окно видеопамяти и происходит возврат на вызывающий модуль.

Выбор вспомогательной подпрограммы. В разделе 3.3.1 было описано несколько вариантов подпрограмм построения строки, каждый из которых применим в конкретных случаях. Все они совместимы по расположению входных параметров в регистрах. Поэтому в пример 3.21 можно подставить имя любой из них, но изменять каждый раз текст или использовать несколько вариантов примера 3.21, различающихся именем вспомогательной подпрограммы, не целесообразно. Проще ввести дополнительный параметр, являющийся адресом вызываемой подпрограммы.

Лучше всего его задавать в регистре `bp`, в виде адреса нужной подпрограммы, а в тексте примера 3.21 вместо команды `call drawline` записать `call bp`, как это указано в комментарии. Если вспомогательные подпрограммы

включены в текст задачи, то для формирования адреса используется команда lea, например:

```
lea   bp, drawline  ; для рисования 256-цветных рисунков
lea   bp, drwlin4   ; для рисования 16-цветных рисунков
lea   bp, drwlin1   ; для рисования двухцветных рисунков
lea   bp, horline   ; для закрашивания прямоугольной области
```

Таким образом, мы получили универсальную процедуру, позволяющую:

- закрашивать прямоугольные области произвольного размера;
- рисовать рисунки небольшого размера;
- сохранять в оперативной памяти содержимое прямоугольной области;
- восстанавливать сохраненное ранее содержимое прямоугольной области.

При рисовании, сохранении или восстановлении содержимого видеопамяти прямоугольная область может содержать не более чем 65 536 точек. По существу это единственное ограничение описанной процедуры. Вы можете составить подпрограммы для более сложных манипуляций со строками небольших рисунков. При этом должно соблюдаться только одно требование — по расположению параметров в регистрах они должны быть совместимы с описанными выше.

Особенности работы с большими рисунками. Большие рисунки не помещаются в одном сегменте оперативной памяти, и их приходится считывать и выводить на экран по частям. В этом случае при построении основное время затрачивается не на рисование строк, а на чтение данных из файла в оперативную память. Очевидно, что чем больше размер порции данных, считываемых за одно обращение к файлу, тем меньше повторных обращений к процедуре чтения и тем быстрее будет построен рисунок. Стандартные средства DOS, например функция 3Fh прерывания int 21h, позволяют прочитать за одно обращение к диску от 1 до 65 535 байт. Однако считывать каждый раз по 65 535 байтов не рационально, и вот почему.

Размер считываемой порции. Пока образ рисунка помещается в одном сегменте, не нужен контроль значений адресов оперативной памяти. Однако при считывании части образа большого рисунка в сегменте может оказаться только часть последней строки и при ее построении без контроля значений адресов оперативной памяти произойдет сбой в работе задачи. Контролировать адреса после вывода каждой точки рисунка явно не рационально, поскольку на отслеживание достаточно редкого события будет затрачено много вспомогательных действий. Проще поступить иначе: при обращении к диску считывать максимально возможное *количество полных строк*, помещающееся в сегменте.

Чтобы узнать его, надо число 65 535 разделить на размер строки рисунка. После деления частное умножается на размер строки, в результате получает-

ся размер порции для чтения в байтах. Последняя порция данных наверняка окажется меньшего размера и это надо учесть при построении рисунка.

Новые переменные. При построении рисунка используются следующие новые переменные, которые должны быть описаны в разделе данных. Первые четыре из них являются параметрами подпрограммы.

```
SwpOffs   dw 0   ; адрес (смещение) в буфере обмена
SwpSeg    dw 0   ; значение сегмента, содержащего буфер обмена
iwidth    dw 0   ; ширина строки рисунка
iheight   dw 0   ; количество строк в рисунке
numbyte   dw 0   ; количество байтов в считываемой порции данных
part      dw 0   ; количество строк в считываемой порции данных
remline   dw 0   ; остающееся не выведенным количество строк
```

Переменные `SwpOffs` и `SwpSeg` указывают полный адрес буфера обмена, в который считываются данные из файла. Как резервируется пространство оперативной памяти, описано в приложении Б данной книги. Значения переменных `iwidth` и `iheight` получаются при обработке заголовка файла, содержащего образ строящегося рисунка. Значения переменных `numbyte`, `part` и `remline` формирует сама подпрограмма `BigDraw`.

Текст подпрограммы, выполняющей построение рисунка произвольного размера, приведен в примере 3.22. Перед ее вызовом в регистре `di` надо указать адрес левой верхней точки рисунка в видеопамяти и установить окно, которому принадлежит этот адрес. Напоминаем, что регистр `es` должен содержать код видеосегмента (содержимое переменной `Vbuff`).

Пример 3.22. Построение рисунка произвольного размера

```
BigDraw:  pusha                  ; сохраняем стандартные регистры
          PushReg <fs, Cur_win>  ; сохраняем fs и Cur_win
          mov   fs, SwpSeg       ; fs = сегмент буфера обмена
          mov   SwpOffs, 0       ; нулевой адрес в буфере обмена
          xor   dx, dx           ; старшая часть делимого dx=0
          mov   ax, -1           ; младшая часть делимого ax=65535
          div   iwidth           ; ax = 65535 / iwidth
          mov   part, ax         ; число строк в порции данных
          mul   iwidth           ; размер порции данных в байтах
          mov   numbyte, cx      ; сохраняем его в numbyte
          mov   ax, iheight      ; копируем количество строк в рисунке
          mov   remline, ax      ; в счетчик не выведенных строк
          mov   bx, horsize      ; bx = horsize
          sub   bx, iwidth       ; bx = horsize – iwidth
NewPart:  mov   cx, numbyte      ; указываем размер порции для чтения
          call  readf            ; чтение очередной порции данных
```

```
            jnc   sucread           ; -> чтение без ошибок
;           Здесь должны выполняться действия при ошибке чтения
sucread:    mov   cx, part          ; cx = стандартное количество строк
            cmp   remline, cx       ; осталось меньше строк ?
            jae   @F                ; -> нет, обходим команду пересылки
            mov   cx, remline       ; cx = оставшееся число строк
@@:         sub   remline, cx       ; уменьшаем оставшееся число строк
            xor   si, si            ; адрес начала в буфере обмена
drwout:     push  cx                ; сохраняем счетчик повторов
            mov   cx, iwidth        ; задаем размер строки рисунка
            call  drawline          ; построение очередной строки
            pop   cx                ; восстанавливаем счетчик повторов
            add   di, bx            ; адрес начала следующей строки
            jnc   @F                ; -> адрес в пределах сегмента
            call  NxtWin            ; установка следующего окна
@@:         loop  drwout            ; управление повторами цикла
            cmp   remline, 0        ; все строки выведены ?
            jne   NewPart           ; -> нет, продолжаем построение
            PopReg <Cur_win, fs>    ; восстановление Cur_win и fs
            popa                    ; восстановление всех регистров
            call  setwin            ; восстановление исходного окна
            ret                     ; возврат из подпрограммы
```

Выполнение подпрограммы примера 3.22 начинается с подготовительных действий. Две первые команды сохраняют в стеке содержимое используемых регистров и значение переменной `Cur_win`. Затем в регистр `fs` помещается сегмент буфера обмена, а переменная `SwpOffs` очищается для расположения считываемых из файла данных с начала буфера обмена. Следующие восемь команд формируют значение переменных `part`, `numbyte`, и `remline`, последняя является счетчиком еще не выведенных строк, поэтому ее исходное содержимое равно высоте рисунка. Остается сформировать в регистре `bx` константу `horsize` — `iwidth` для коррекции адресов строк в видеопамяти.

Построение рисунка выполняют два вложенных цикла. Внешний имеет метку `NewPart`, а внутренний — `drwout`. Во внешнем цикле производится чтение из файла очередной порции данных, уточнение размера прочитанной порции (содержимого регистра `cx`), оставшегося количества строк (переменная `remline`) и очистка регистра `si`. После этого выполняется внутренний цикл, выводящий на экран прочитанную порцию строк. Внутренний цикл полностью совпадает с аналогичным циклом примера 3.21, поэтому мы опустим его описание.

По окончании работы внутреннего цикла проверяется значение переменной `remline` и если оно отлично от нуля, то происходит возврат на начало внешнего цикла. Если все строки выведены на экран, то восстанавливаются сохраненные в стеке величины, исходное окно видеопамяти и происходит возврат на вызывающий модуль.

Подпрограмма чтения с диска. Во внешнем цикле происходит обращение к подпрограмме `Readf`. Она считывает в буфер обмена порцию данных, размер которой (в байтах) задается в регистре `cx`. Ниже (см. пример 3.23) приведен один из возможных примеров такой подпрограммы.

Пример 3.23. Чтение фрагмента файла в буфер обмена

```
Readf: PushReg <bx,dx,ds>   ; сохраняем в стеке bx и ds
       mov    bx, handler   ; указываем заголовок файла
       lds    dx, dword ptr SwpOffs; задаем адрес буфера для чтения
       mov    ax, 3F00h     ; код функции DOS "чтение файла"
       int    21h           ; обращение к DOS
       PopReg <ds,dx,bx>    ; восстанавливаем из стека bx и ds
       ret                  ; возврат из подпрограммы
```

В примере 3.23 адрес буфера обмена выбирается из переменных `SwpOffs` и `SwpSeg`, описанных выше. Если при чтении данных возникнет ошибка, то после возврата из `int 21h` и из подпрограммы будет установлен C-разряд регистра флагов (признак переполнения). При успешном чтении DOS возвращает в регистре `ax` размер прочитанной порции данных в байтах. Если при чтении обнаружен конец файла, то эта величина может быть меньше указанной в регистре `cx`.

В примере 3.23 используется еще одна переменная `handler`, содержащая ссылку на файл (`file handle`). Эту величину формирует DOS при открытии файла по указанной спецификации и передает задаче в регистре `ax`. Ее надо сохранить, например в переменной `handler` и использовать при дальнейшей работе с файлом.

Способ открытия файла для чтения описан в приложении А в примере А.1. Для получения более подробной информации на эту тему рекомендуем обратиться к электронной справочной системе `Tech Help`, или к любому руководству по программированию MS DOS.

Рисунок не помещается на экране. Существуют рисунки, размеры которых превышают размер рабочей области экрана. В таких случаях способ построения выбирается в зависимости от назначения рисунка. Например, если вы собираетесь использовать его в качестве заставки, то целесообразно сократить размеры до допустимых значений с помощью графического редактора и использовать в задаче обрезанный рисунок.

Прежде чем обрезать рисунок, имеет смысл определить его размеры. Возможно, что вам попалась одна из заставок, предназначенная для вывода при высоком разрешении, например 800×600 или 1280×1024 точки. Если вас устраивает работа в таких видеорежимах, то проблемы не существует, достаточно просто устанавливать в задаче нужный видеорежим.

Обрезать рисунок можно и в процессе его построения. Если ширина рисунка превосходит `Horsize`, то при построении строк в видеопамять записывается ровно `Horsize` точек, а остатки отбрасываются. Если высота рисунка превосходит `Versize`, то выводится ровно `Versize` строк, а остальные отбрасываются. Изменить подпрограмму построения рисунка не сложно, вопрос в том, целесообразно ли это делать?

Более гибкий способ заключается в принудительном изменении размера сканируемой строки (`Scanline`). Эта величина задает количество точек, после вывода которых видеоконтроллер переходит на новую строку. После установки видеорежима `Scanline` и `Horsize` равны. В разделе 1.2.2 описана функция `BIOS 4F06h`, которая позволяет установить нужное значение `Scanline`, но не меньшее чем `Horsize`. Если `Scanline` > `Horsize`, то в процессе отображения каждой строки видеоконтроллер выводит на экран `Horsize` точек, а остальные (`Scanline` — `Horsize`) пропускает.

Поэтому можно просто копировать рисунок в видеопамять, предварительно установив значение `Scanline` равным его ширине. После построения на экране будет видна левая верхняя часть рисунка, размер которой равен `Horsize * Versize`. Только не забывайте, что изменение значения `Scanline` сказывается на работе с другими рисунками, в том числе с курсором и текстом. Для учета нового значения надо либо ввести специальную переменную, либо изменить значение переменной `Horsize`.

В видеопамять можно записать большой рисунок и просматривать его по частям, перемещая начало отображаемой области. В разделе 1.2.2 описана функция `BIOS 4F07h`, позволяющая перемещать эту точку. После установки видеорежима начало отображаемой области находится на пересечении нулевой строки и нулевого столбца. В разделе 2.5 было показано, как можно использовать функцию `4F07h` для выбора страниц видеопамяти.

Программирование управления перемещением отображаемой области существенно усложнит вашу задачу, особенно если для управления использовать горизонтальный и вертикальный лифты, как это делается в Windows и ее приложениях. Поэтому описанный способ работы с большими рисунками имеет смысл применять в тех случаях, когда это действительно необходимо и у вас есть достаточный опыт программирования графики.

3.3.3. Воспроизведение сжатых рисунков

Для сокращения размера файлов образы рисунков могут храниться в сжатом виде. Частным случаем является упаковка точек 16- и 2-цветных рисунков, когда в байте располагаются коды двух или восьми подряд расположенных точек (см. раздел 3.3.1). Здесь нас будут интересовать способы упаковки и распаковки 256-цветных рисунков.

Сразу отметим, что в этой области нет никакой унификации, и разработчики стандартов для хранения и передачи файлов выбирают способ сжатия по

своему усмотрению. В данном разделе основное внимание уделено способу сжатия, получившему название RLE (Run-Length-Encoding), который предусмотрен в стандартах PCX, BMP и некоторых других. Он дает далеко не лучшие результаты, но имеет одно неоспоримое преимущество, которое заключается в простоте распаковки. Это позволяет привести исчерпывающее описание способа построения рисунка. Стандарт BMP описан в приложении А данной книги, здесь описан стандарт PCX.

Стандарт создала фирма ZSoft — разработчик графических редакторов PaintBrush, PhotoFinish и пр. Ему, как и многим другим стандартам, присущи некоторые разночтения, вызванные тем, что улучшать устаревающие версии пыталась не только ZSoft, но и некоторые другие фирмы, например Genius Microprogramming.

Заголовок PCX-файла имеет фиксированный размер 80h байтов, сразу после него (начиная с адреса 80h) располагается образ рисунка. Нас будут интересовать лишь некоторые байты и слова заголовка.

Байт 0 должен содержать код 0Ah, являющийся признаком того, что файл соответствует стандарту PCX.

Байт 1 содержит версию стандарта (от 0 до 5), в частности код 5 соответствует третьей версии стандарта, в которой впервые было введено использование 256-цветной палитры.

Байт 2 содержит 1, если образ рисунка хранится в сжатом виде, или 0 — в противном случае (распаковка не требуется).

Байт 3 содержит размер точки изображения в битах, для 256-цветных рисунков его значение равно 8.

Слова 4, 6, 8 и **0Ah** содержат минимальные и максимальные значения координат рисунка (Xmin, Ymin, Xmax, Ymax). Ширина и высота рисунка вычисляются так: iwidth = Xmax − Xmin + 1, iheight = Ymax − Ymin + 1.

Слово 42h содержит размер строки рисунка в байтах, мы обозначим его содержимое fwidth. При четном количестве точек в строке iwidth = fwidth, при нечетном количестве точек в строке fwidth = iwidth + 1. В этом случае строка содержит дополнительный байт, который учитывается при распаковке, но не выводится на экран, т. к. его содержимое не определено.

Заголовок файла содержит и другие величины, но в данном разделе они нам не понадобятся. Более подробную информацию о заголовке PCX-файлов вы найдете в книге [4], а мы вернемся к рассмотрению стандарта PCX при описании работы с палитрой и построения полноцветных рисунков.

Техника распаковки строки. Признаком упакованного рисунка является 1 во втором байте заголовка файла. Результаты упаковки по способу RLE интерпретируются так: если в текущем байте установлены два старших разряда (коды 0C0h – 0FFh), то шесть его младших разрядов указывают, сколько раз

надо повторить следующий байт. В противном случае (один или оба старших разряда очищены) текущий байт содержит код одной точки.

При упаковке строки исходного рисунка обрабатываются независимо друг от друга, т. е. цикл сжатия останавливается в конце каждой строки и начинается заново с началом следующей. Соответственно, цикл распаковки должен продолжаться до тех пор, пока количество полученных точек (байтов) не совпадет с размером строки `fwidth`.

Прежде чем рассматривать конкретный пример, обсудим некоторые общие вопросы. Нам предстоит написать программу, которая манипулирует с двумя потоками байтов. Входной поток содержит сжатый образ рисунка, а выходной — образ того же рисунка, но уже распакованный.

Будем считать, что входной поток находится в буфере обмена, адрес начала которого задают переменные `SwpOffs` и `SwpSeg`. При распаковке из этого буфера выбираются один или два очередных байта. Очевидно, что при этом надо следить за тем, чтобы нужные байты находились в буфере и при необходимости пополнять буфер новыми данными. Исходя из опыта (и не только автора) целесообразно составить специальную подпрограмму, которая при каждом обращении возвращает очередной байт данных, следит за состоянием буфера и в нужный момент пополняет его новыми данными.

Выходной поток, в конечном итоге, направляется в видеопамять, но объединять в одной подпрограмме действия, связанные с распаковкой и записью кодов точек в видеопамять, нецелесообразно — эти функции должны выполнять разные процедуры. Поэтому мы рассмотрим подпрограмму, которая помещает коды точек в специально выделенный буфер.

Подпрограмма распаковки строки. Предположим, что в оперативной памяти задача зарезервировала пространство для размещения буфера общего назначения достаточно большого размера. Сегмент, в котором расположен буфер, хранится в переменной `GenSeg`, а начало свободного пространства в этом сегменте — в переменной `GenOffs`. Эти переменные должны располагаться в разделе данных программы в следующем порядке.

```
GenOffs  dw 0     ; адрес (смещение) в буфере общего назначения
GenSeg   dw 0     ; сегмент, содержащий буфер общего назначения
```

Способы резервирования пространства в оперативной памяти описаны в приложении Б данной книги.

Текст подпрограммы распаковки строки приведен в примере 3.24.

Пример 3.24. Распаковка строки рисунка (способ RLE для PCX)

```
Unpack:  PushReg <ax,cx,dx,di,es> ; сохранение содержимого регистров
         les     di, Dword ptr GenOffs; смещение и сегмент буфера
         mov     dx, fwidth        ; логический размер строки
```

```
Unploop: call    nxt_sym            ; читаем в al следующий символ
         mov     cx, 01             ; количество повторяемых символов
         cmp     al, 0C0h           ; символ содержит счетчик повторов ?
         jbe     Un1                ; -> нет, это одиночный символ
         mov     cl, al             ; копируем содержимое al в cl
         and     cl, 3Fh            ; и выделяем количество повторов
         call    nxt_sym            ; читаем в al — повторяемый символ
Un1:     sub     dx, cx             ; уменьшаем остаток строки
         rep     stosb              ; записываем символы в буфер строки
         or      dx, dx             ; строка распакована полностью ?
         jnz     Unploop            ; -> нет, продолжение распаковки
         PopReg <es,di,dx,cx,ax>    ; восстанавливаем регистры
         ret                        ; возврат из подпрограммы
```

В примере 3.24 перед началом распаковки в стеке сохраняется содержимое используемых регистров. Затем команда `les` загружает в регистры `es:di` адрес для записи распакованной строки. Размер строки в байтах помещается в регистр `dx`, используемый в качестве счетчика распакованных символов. После этого выполняется цикл `Unploop`.

Действия при распаковке соответствуют описанному выше алгоритму. Очередной байт считывается в регистр `al`, а в счетчик повторов `cx` записывается 1. Если код символа меньше чем `C0h`, то происходит переход на метку `Un_1`. В противном случае в `cl` помещается содержимое 6-ти младших разрядов регистра `al` и читается повторяемый символ.

Команда, имеющая метку `Un_1`, вычитает из счетчика распакованных символов содержимое регистра `cx`. Следующая команда записывает в буфер строки содержимое регистра `al` столько раз, сколько указано в регистре `cx`. После этого проверяется содержимое регистра `dx` и если оно отлично от нуля, то цикл распаковки продолжается. В противном случае восстанавливается содержимое сохраненных в стеке регистров и выполняется команда возврата.

Подпрограмма Nxt_sym. Для получения кода очередного байта в примере 3.24 вызывается подпрограмма `Nxt_sym`, текст которой приведен в примере 3.25.

Пример 3.25. Чтение очередного символа из буфера обмена

```
Nxt_sym: cmp  si, incount       ; в буфере есть символы ?
         jb   @F                ; -> да, можно читать очередной символ
         push cx                ; сохраняем содержимое cx
         mov  cx, -1            ; указываем размер порции данных
         call Readf             ; читаем данные из файла
         mov  incount, ax       ; сохраняем размер порции данных
         xor  si, si            ; очищаем указатель адреса
         pop  cx                ; восстанавливаем содержимое cx
@@:      lods byte ptr fs:[si]  ; чтение очередного байта
         ret                    ; возврат из подпрограммы
```

Подпрограмма примера 3.25 сравнивает текущее значение указателя адреса буфера обмена (содержимое регистра `si`) с переменной `incount`, значение которой соответствует размеру считанной из файла порции данных, т. е. количеству байтов, находящихся в буфере обмена.

Если в буфере достаточно данных, то происходит переход на локальную метку `@@` для чтения кода символа в регистр `al`, увеличения указателя адреса на 1 и выхода из подпрограммы.

Если достигнута граница данных, хранящихся в буфере обмена, то надо прочитать новую порцию данных. Для этого сохраняется содержимое регистра `cx`, в него записывается предельное количество байтов для чтения (−1 имеет код `0FFFFh`) и происходит обращение к подпрограмме `Readf`, описанной в примере 3.23.

После чтения в переменную `incount` записывается количество прочитанных байтов, очищается регистр `si` и восстанавливается из стека содержимое регистра `cx`. Теперь можно прочитать очередной символ, увеличить значение `cx` на единицу и выйти из подпрограммы.

В примере 3.25 отсутствует проверка состояния C-разряда регистра признаков после чтения. Вы можете включить ее в текст примера 3.25, но целесообразнее контролировать правильность чтения непосредственно в подпрограмме `Readf`. Это упростит структуру всех подпрограмм, которые обращаются к `Readf`.

Замечание

В литературе встречаются сведения о существовании файлов, частично не соответствующих стандарту фирмы ZSoft — при их сжатии цикл упаковки не прекращается в конце строки. Для распаковки подобных файлов подпрограмму примера 3.24 надо изменить так, чтобы она выдавала заданное количество распакованных кодов независимо от размера строки.

Построение рисунка. В цикле построения упакованного рисунка каждая строка сначала распаковывается с помощью подпрограммы `Unpack`, а затем результат распаковки записывается в видеопамять. Текст подпрограммы построения рисунка приведен в примере 3.26.

Предварительно вы должны открыть файл и прочитать его заголовок для определения значений переменных `iheight`, `iwidth` и `fwidth`. После чтения заголовка указатель файла содержит значение `80h`, соответствующее началу образа рисунка.

Перед вызовом подпрограммы в регистре `di` указывается адрес левой верхней точки рисунка в видеопамяти и устанавливается соответствующее окно. Регистр `es` должен содержать код видеосегмента. В разделе данных задачи надо описать переменную `incount`, имеющую размер слова, в ней подпрограмма хранит количество символов, прочитанных в буфер обмена.

Пример 3.26. Построение рисунка, упакованного в стандарте PCX

```
PackDrw: PushReg <cx,si,di,Cur_win>; сохранение используемых величин
         xor   si, si         ; очистка регистра si
         mov   incount, si    ; incount = 0
         mov   cx, iheight    ; cx = количество строк в рисунке
make:    push  cx             ; сохраняем счетчик повторов
         call  Unpack         ; распаковка очередной строки
         PushReg <fs,si>      ; сохранение содержимого fs и si
         lfs   si, dword ptr GenOffs; fs:si = адрес распакованной строки
         mov   cx, iwidth     ; cx = количество точек в строке
         call  drawline       ; вывод строки рисунка на экран
         PopReg <si,fs>       ; восстановление содержимого fs и si
         mov   ax, horsize    ; копируем в ax ширину экрана и
         sub   ax, iwidth     ; вычитаем из нее ширину рисунка
         add   di, ax         ; адрес начала следующей строки
         jnc   @F             ; -> адрес в пределах видеосегмента
         call  Nxtwin         ; установка следующего окна
@@:      pop   cx             ; восстановление счетчика повторов
         loop  make           ; управление циклом рисования
         PopReg <Cur_win,di,si,cx>; восстановление из стека
         call  SetWin         ; восстановление исходного окна
         ret                  ; возврат из подпрограммы
```

Выполнение подпрограммы примера 3.26 начинается с сохранения в стеке тех величин, которые могут измениться при ее работе. Для того чтобы при первом обращении к подпрограмме `Nxt_sym` она прочитала в буфер обмена часть образа рисунка, содержимое регистра `si` должно совпадать со значением переменной `incount`. Поэтому регистр `si` и переменная `incount` очищаются. В регистре `cx` указывается количество строк рисунка `iheight`.

Цикл построения рисунка имеет метку `make`. Его выполнение начинается с сохранения в стеке содержимого регистра `cx` и вызова подпрограммы `Unpack`. После распаковки в стеке сохраняется содержимое регистров `fs`, `si` и команда `lfs` загружает в них адрес начала распакованной строки. В регистре `cx` указывается размер строки `iwidth` и вызывается подпрограмма `drawline` для записи строки в видеопамять. Какой именно вариант этой подпрограммы вы будете использовать, не имеет значения.

После возврата из подпрограммы `drawline` восстанавливается содержимое регистров `si` и `fs`, вычисляется константа для коррекции адреса строки в видеопамяти (`horsize – iwidth`), которая прибавляется к текущему адресу видеопамяти, находящемуся в регистре `di`. Если при сложении произойдет переполнение, то подпрограмма `NxtWin` установит следующее окно видеопамяти.

Из стека восстанавливается содержимое регистра `cx` и команда `loop` повторяет выполнение цикла до тех пор, пока не будет построен весь рисунок. После выхода из цикла восстанавливаются сохраненные в стеке величины, устанавливается исходное окно видеопамяти и происходит возврат на вызывающий модуль.

Мы еще дважды вернемся к теме работы с файлами стандарта `PCX` — при описании установки палитры (см. раздел 4.4) и построения полноцветных рисунков (см. раздел 7.4.2). В заключение данного раздела несколько слов о способах сжатия графических изображений.

Недостатки сжатия по способу RLE. Способ сжатия `RLE` используется не только в стандарте `PCX`, но и в стандарте `BMP`. В деталях эти варианты различаются, но в главном они совпадают. При упаковке группа одинаковых кодов (одноцветных точек) заменяется двумя байтами, в первый записывается количество повторов, а во второй — повторяемый код.

Очевидным достоинством способа `RLE` является простота его программной реализации, ради чего он и создавался, но степень сжатия рисунка не столь высока. Сжатие происходит в тех случаях, когда в рисунке подряд расположены, по крайней мере, три одноцветные точки. Если же цвет очередной точки не совпадает с цветом следующей, а ее код больше чем `0BFh`, то в выходной файл вместо одного записываются два байта. В первый будет записан код `C1h`, а во второй — код точки. Поэтому алгоритм работает эффективно, если в рисунке встречается много групп подряд расположенных одноцветных точек, и чем чаще различаются цвета смежных точек, тем меньше степень сжатия. При неблагоприятном стечении обстоятельств размер сжатого изображения может оказаться больше чем исходного.

Характеристика стандарта GIF. Проблема сжатия рисунков приобрела особую важность по мере развития сначала локальных, а затем и глобальных сетей ЭВМ. В июне 1987 года компания CompuServe Incorporated опубликовала описание стандарта `GIF` (`Graphics Interchange Format` — формат графического обмена). Для сжатия рисунков в нем применяется модифицированный алгоритм `LZW` (`Lempel-Ziv-Welch`), используемый в распространенных архиваторах текстовых данных. На сегодняшний день это наилучший способ сжатия растровых рисунков, подготовленных с использованием палитры цветов.

Алгоритмы `LZW` и `RLE` различаются принципиально. Главный недостаток `RLE` заключается в самой идее подсчитывать число подряд расположенных совпадающих кодов. Способ `LZW` избавлен от этого недостатка. При сжатии запоминаются и используются *все последовательности* встречающихся в рисунке точек, независимо от совпадения или различия их цветов. В выходной файл записываются не байты, а цепочки битов переменной длины, коды которых соответствуют кодам отдельных точек или их комбинаций.

Предположим, что размер кодов точек рисунка равен K, а их значения изменяются от 0 до N-1, где $N = 2^k$. Например, если в рисунке используются все 256 цветов, то K= 8, а N = 256. Первоначально для хранения цепочек отводится K+1 разряд. Если код цепочки меньше N, то это просто код одной точки. Коды N и N+1 имеют специальное назначение. Комбинации точек кодируются, начиная от значения N+2. Какой комбинации соответствует тот или иной код цепочки (если он больше чем N+1) можно узнать только из таблицы цепочек.

Первоначально таблица цепочек пуста, в процессе сжатия в нее записываются все новые (отсутствующие в таблице) комбинации точек. Если таблица окажется полностью заполненной, то размер цепочек увеличивается на 1 разряд и увеличивается пространство, отведенное для таблицы. Предельно допустимый размер цепочки составляет 12 разрядов. Если этого окажется недостаточно (вероятность такого события мала), то в выходной файл записывается специальный признак новой таблицы (код N), таблица цепочек очищается, выбирается размер цепочки, равный K+1, и продолжается процесс сжатия с формированием новой таблицы.

В выходной (упакованный) файл таблица цепочек не записывается. Она воспроизводится в процессе распаковки. Это значит, что при упаковке и при распаковке приходится работать с тремя структурами данных: входной и выходной массивы и таблица цепочек. Поэтому упаковка и распаковка по способу LZW занимает намного больше времени, чем те же действия по способу RLE и требуется достаточно много оперативной памяти для хранения таблиц цепочек. Однако эти издержки окупаются качеством упаковки.

Для сравнения приведем размеры файла, хранящегося в трех стандартах:

Clouds.bmp	307514	файл не упакован, 640×480 точек
Clouds.pcx	300527	файл упакован по способу RLE
Clouds.gif	159287	файл упакован по способу LZW

Файл Clouds.bmp (Облака.bmp) выбран по двум причинам. Во-первых, это одна из стандартных заставок Windows 9X и вы можете увидеть хранящийся в нем рисунок. Во-вторых, этот рисунок трудно сжимаемый, и применение способа RLE не дает ощутимых результатов. Тем не менее стандарт GIF сокращает размер файла почти в два раза, причем это не лучший результат. Сжатие файла выполнено с помощью графического редактора PhotoFinish фирмы ZSoft. Этот редактор (как и любой другой) при упаковке не изменяет количество цветов. В рисунке "Облака" используется 65 цветов. Если пожертвовать одним из них, то размер кода точки сократится с 7-ми до 6-ти разрядов, а размер файла в формате GIF сократится почти в 4 раза. Однако для анализа и сокращения количества цветов нужна специальная программа.

Стандарт GIF разрабатывался специально для передачи данных по компьютерным сетям. Поэтому упакованный файл разбит на отдельные блоки (при ошибке передачи повторно передается только один блок).

На практике совсем не обязательно составлять собственную программу для работы с рисунками, хранящимися в формате GIF. Современные графические редакторы поддерживают наиболее распространенные стандарты и позволяют преобразовывать файлы из одного формата в другой. Кроме того, существуют специализированные конвертеры для преобразования графических файлов из одного формата в другой. Поэтому вы всегда можете преобразовать нужные рисунки в тот формат, который поддерживает ваша программа.

3.3.4. Заключительные замечания

До сих пор мы рассматривали способы работы с рисунками "в чистом виде" — обсуждали варианты построения рисунков и ничего не говорили о сопутствующих действиях. Таких действий достаточно много, и при их выполнении приходится решать задачи, которые могут оказаться намного сложнее, чем простая запись кодов точек в видеопамять. В данном разделе приведена общая характеристика сопутствующих действий, а конкретные способы выполнения некоторых из них описаны в следующих главах книги.

Установка палитры. В файлах, содержащих образы точечных рисунков, обязательно хранится палитра, в которой находятся коды использованных в рисунке цветов, иногда ее называют таблицей цветов. Коды точек рисунка являются порядковыми номерами входящих в палитру цветов (или строк таблицы цветов). Поэтому перед построением рисунка палитра должна быть установлена, т. е. коды перечисленных цветов должны быть записаны в DAC-регистры видеокарты. Если при этом изменится порядок расположения или номера цветов палитры, то перед записью в видеопамять должны быть изменены коды некоторых или всех точек рисунка. Установка палитры и различные манипуляции с ней описаны в следующей главе книги.

Изменение состояния курсора. Трудно представить графическую задачу, при выполнении которой не используется курсор — специальный рисунок, указывающий на экране текущее положение манипулятора "мышь". Обычно курсор перемещается так, что находящееся на экране изображение не портится. Если же вновь добавленный рисунок закроет изображение курсора, то при перемещении последнего на месте старой позиции возникнет прямоугольник, содержащий фрагмент изображения исходного фона, а не нового рисунка. Для исключения таких случаев изображение курсора удаляется с экрана перед построением нового рисунка, а затем вновь восстанавливается на экране.

Можно организовать анализ взаимного расположения курсора и добавляемого рисунка и удалять курсор только в тех случаях, когда рисунок закрывает часть его изображения. Однако проще удалить курсор на время выполнения любых изменений находящегося на экране изображения. Работа с курсором описана в главе 6.

Сохранение исходного фона. Для того чтобы рисунок можно было удалить с экрана или переместить на экране с одного места на другое, перед его построением надо сохранить содержимое тех адресов видеопамяти, в которые записываются коды точек строящегося рисунка. Иначе говоря, надо сохранить исходную картинку (исходный фон) на той части экрана, которую займет новый рисунок. В главе 5 (см. раздел 5.2.3) описаны подпрограммы для сохранения и восстановления исходного фона на месте информационных строк, содержащих различные текстовые сообщения.

Обычно исходный фон сохраняется в оперативной памяти. Размер буфера, в котором сохраняется исходный фон, зависит от размеров рисунка и может оказаться достаточно большим. Фоновые рисунки большого размера приходится сохранять в файлах на внешних носителях. В приложении Б данной книги приведена подпрограмма, выполняющая сохранение или восстановление изображения заполняющего все пространство рабочей области экрана.

Важно

Перед сохранением исходного фона с экрана обязательно удаляется изображение курсора, в противном случае после восстановления фона на экране окажутся два изображения курсора — неподвижное и перемещаемое.

Маскировка. Существует особая категория рисунков, при построении которых используется маска. Маска может храниться в готовом виде в файле, содержащем рисунок, например она обязательно прилагается к рисункам курсоров и пиктограмм (значков). Другую категорию рисунков в англоязычной литературе принято называть "спрайтами" (`sprite`). Маска для них формируется динамически, в зависимости от значений кодов точек рисунка, но в любом случае маскировка преследует одну цель.

Какой бы формы не был сам рисунок, например треугольник, стрелка, песочные часы и т. д., его образ всегда дополняется до прямоугольника, для того чтобы все строки имели одинаковый размер. Это существенно упрощает хранение и воспроизведение рисунков и одновременно вынуждает применять маскировку, исключающую вывод на экран той части прямоугольной области, которая не относится к рисунку. За счет наложения маски вы видите на экране, например изображение стрелки курсора, а не черный прямоугольник, на фоне которого она нарисована. Более подробно мы поговорим о маскировке при рассмотрении способов построения курсора в главе 6 и продолжим эту тему при описании видеорежимов `direct color`.

Перемещение рисунков. Простейшим примером перемещаемого рисунка является изображение курсора. Работа с ним подробно описана в главе 6.

В общем случае при первом построении перемещаемого рисунка надо сохранить исходный фон на занятом им месте. Для перемещения рисунок удаляется с экрана и воспроизводится на новом месте. Перед перемещением

сохраняется исходный фон на новой позиции, а после перемещения на месте удаленного рисунка восстанавливается исходный фон. Если адреса точек старой и новой позиции рисунка относятся к одному сегменту видеопамяти (точки расположены в одном окне), то для перемещения достаточно простого копирования содержимого одних байтов видеопамяти в другие. Однако в общем случае при таком способе перемещения перед чтением и записью кода каждой точки придется проверять принадлежность адреса нужному сегменту видеопамяти и при несоответствии изменять текущее окно. В результате перемещение будет происходить крайне медленно.

Если в распоряжении задачи имеется достаточный объем видеопамяти, то имеет смысл копировать исходный рисунок в оперативную память, а оттуда в нужное место видеопамяти. При такой схеме перемещения затраты на работу с окнами видеопамяти будут минимальными. Если доступное задаче пространство оперативной памяти ограничено, то через оперативную память можно перемещать отдельные строки рисунка. Это упростит работу с окнами видеопамяти в пределах каждой строки.

Идентификация находящихся на экране объектов. Из опыта работы с компьютером вы наверняка знаете, что с помощью манипулятора "мышь" можно не только перемещать изображение курсора по экрану, но и выполнять различные преобразования объектов, на которые указывает курсор. Например, можно перемещать рисунок вместе с курсором, вызывать появление выпадающих или всплывающих меню, выполнять действия, указанные в окнах меню и т. п. Для выполнения подобных действий задача должна формировать структуру данных, содержащую исчерпывающую информацию обо всех расположенных на экране объектах.

Элемент структуры может содержать координаты конкретного объекта, его размер, указание назначения и другие сведения об особенностях работы с объектом. Например, если объект является перемещаемым рисунком, то понадобится адрес буфера, содержащего исходный фон, а если это один из элементов оформления "рабочего стола", то важно знать адрес процедуры, выполняющей связанные с ним действия.

Описание объектов обычно оформляется в виде таблицы или списка. Таблица состоит из строк одинакового размера, расположенных в оперативной памяти последовательно друг за другом. Поэтому возможен прямой доступ к любой строке. Элементы списка могут иметь переменный размер и располагаться в памяти в произвольном порядке. Прямое обращение к произвольному элементу списка невозможно, он находится по цепочке ссылок. Каждый элемент содержит указатель адреса следующего элемента (ссылку на следующий элемент), что и позволяет перемещаться по списку вперед. Если при работе возникает необходимость перемещаться в обратном направлении, то используется еще одна ссылка, указывающая адрес предыдущего элемента списка. Механизм ссылок делает список гибкой и легко из-

меняемой структурой, но одновременно замедляет процесс поиска нужных элементов. Поиск в таблице осуществляется быстрее, но изменить порядок расположения ее строк, если это понадобится, сложнее, чем изменить последовательность доступа к элементам списка.

Выбор конкретной структуры для хранения описания объектов зависит, в основном, от стиля программирования, которого вы придерживаетесь, ваших практических навыков и, в меньшей степени, от программируемой задачи.

ГЛАВА 4

Цвет на экране

Работа с цветом является неотъемлемой частью любой графической программы. В предыдущей главе мы почти не затрагивали вопросы, связанные с получением нужного цвета изображения. Это делалось не только для упрощения изложения материала. В большинстве случаев в режимах `PPG` действия, выполняемые при построении изображения, никак не связаны с цветом выводимых точек. Формирование нужных цветов обычно производится до построения изображения, при этом выполняются специфические действия, которые могут не требовать непосредственной работы с видеопамятью.

По способу указания цвета и работы с ним все видеорежимы VESA делятся на две группы. К первой относятся режимы `PPG`, в этом случае код цвета находится в специальных регистрах видеоконтроллера, а ко второй группе — режимы `direct color`, при их установке цвет зависит только от кода точки. В данной главе мы продолжим описание программирования для режимов `PPG`, режимы `direct color` будут описаны в главе 7. Независимо от того, каким видеорежимам вы отдаете предпочтение, советуем прочитать эту главу, поскольку в ней, кроме специальных приемов программирования, обсуждаются общие вопросы, связанные с цветом.

4.1. Как получается цвет точки

Цвет появляется на экране монитора в результате совмещения трех базовых цветов в одной точке. Для программирования работы с цветом не имеет принципиального значения, что является источником этих трех цветов и как они смешиваются в одной точке. Важно другое — как можно управлять интенсивностью каждого цвета и что получается в результате их смешения (наложения в одной точке).

На практике мы в большинстве случаев имеем дело с аналоговыми `RGB`-мониторами. Это значит, что на их входы, помимо прочих сигналов, посту-

пают три разных напряжения, задающие интенсивность красного (Red), зеленого (Green) и синего (Blue) цветов. В последние годы выпускаются мониторы с цифровым управлением, но оно не распространяется на сигналы цветности, которые остаются аналоговыми.

DAC-регистры видеокарты. Аналоговые входы монитора подключаются к выходам специальных регистров видеокарты. Их сокращенное название DAC (Digital-to-Analog Converter) соответствует русскому техническому термину "цифро-аналоговый преобразователь" (ЦАП). На входы DAC подается цифровая информация, а на выходе получается напряжение, плавность изменения которого зависит от количества разрядов в регистре.

В SVGA-видеокартах на каждый базовый цвет отведено 8 разрядов DAC-регистра (1 байт), поэтому возможны 256 градаций каждого цвета. При установке режимов PPG количество разрядов искусственно сокращается до 6, а количество разных уровней напряжения, соответственно, уменьшается до 64. Это сделано для соответствия требованиям стандарта VGA, разработанного, в свое время, фирмой IBM. При работе в видеорежимах direct color указанное ограничение не действует, и используются все восемь разрядов.

Для вывода точки одновременно выдаются напряжения с выходов трех регистров, иначе говоря, используется группа из трех регистров. Видеокарта содержит 256 таких групп (троек). В видеорежимах PPG код каждой точки, записываемой в видеопамять, является номером одной из троек DAC-регистров. В описаниях BIOS говорится о DAC, как об одном 18-разрядном регистре. Возможно, так оно и есть, для нас это не существенно. Важно, что при работе в видеорежимах PPG получаемый на экране цвет зависит от того, что было предварительно записано в DAC-регистр (или в тройку DAC-регистров).

В восемнадцати двоичных разрядах можно записать одну из $64 \cdot 64 \cdot 64 = 262\,144$ комбинаций нулей и единиц, следовательно, в тройке DAC-регистров можно закодировать именно такое количество цветов, но на экран одномоментно выводятся только любые 256 из них (по количеству DAC-регистров). Несоответствие между количеством оттенков, которое может иметь каждая точка, и общим количеством цветов, которое можно одновременно вывести на экран, устраняется только при работе в режимах direct color.

Естественный и искусственный цвет. Красный, зеленый и синий цвета используются в качестве базовых не только в компьютерных мониторах, но и в телевидении, видеосъемке, цветной фотографии и т. д. Возникает естественный вопрос о том, как они соотносятся с теми цветами, которые мы встречаем в окружающем нас мире.

Вся цветовая гамма, которую способен воспринять человеческий глаз, содержится в радуге. Это довольно редкое природное явление, при котором наблюдается огромная дуга, содержащая плавно переходящие друг в друга цвета от красного до фиолетового. Принято говорить о семи цветах раду-

ги — красный, оранжевый, желтый, зеленый, голубой, синий, фиолетовый. Такое деление условно и, скорее всего, является одним из отголосков веры наших предков в некие исключительные свойства числа семь. Разрешающая способность нормального человеческого глаза позволяет уверенно различать множество оттенков каждого цвета радуги.

Базовые цвета RGB делят весь видимый цветовой диапазон на три части: красный объединяет красное и оранжевое поля радуги, зеленый — желтое и зеленое, синий — голубое, синее и фиолетовое. Для получения нужного оттенка смешиваются базовые цвета разной интенсивности, но предсказать, какой получится оттенок, можно только в простых случаях. Например, если три базовых цвета имеют одинаковую интенсивность, то в зависимости от ее значения получатся разные оттенки серого цвета. При работе в режимах PPG можно получить 64 оттенка от чисто черного до чисто белого цвета.

В более сложных случаях нужный оттенок подбирается эмпирически с помощью графических редакторов. Большинство графических редакторов позволяет либо выбирать нужный оттенок из предлагаемой палитры, либо изменять конкретные значения базовых цветов, пока не будет подобрана их подходящая комбинация. Изменением или подбором цветов отдельных точек приходится заниматься, главным образом, при редактировании готовых рисунков, поэтому мы не будем углубляться в рассмотрение этого вопроса.

Цвета RGB и CMY. RGB является основным, но не единственным набором базовых цветов, используемым в вычислительной технике. В цветных струйных принтерах, производимых фирмой Hewlett Packard (и не только этой фирмой), применяется дополнительный набор цветов CMY (Cyan, Magenta, Yellow). На практике именно он удобен для печати рисунков на бумаге. Поэтому работу с палитрой CMY поддерживает большинство графических редакторов. Связь между наборами цветов RGB и CMY иллюстрирует табл. 4.1. В ней показано, что получится при наложении базовых цветов, интенсивность каждого из которых принимает одно из двух значений: 0 или 100%. В таком случае возможны восемь различных комбинаций интенсивности и столько же разных цветов. Названия двух из них требуют некоторых уточнений. Magenta обычно переводится как "фуксин" — красная анилиновая краска. Слово Cyan обычно не переводят, а используют транскрипцию (циан), этот цвет можно охарактеризовать как сине-зеленый.

Таблица 4.1. *Смешение базовых цветов 100%-ной интенсивности*

Интенсивность базовых цветов			Результат наложения цветов
Красный	Зеленый	Синий	
0	0	0	Черный (black)
0	0	100	Синий (blue)

Таблица 4.1 *(окончание)*

Интенсивность базовых цветов			Результат наложения цветов
Красный	**Зеленый**	**Синий**	
0	100	0	Зеленый (green)
0	100	100	Циан (cyan)
100	0	0	Красный (red)
100	0	100	Мажента (magenta)
100	100	0	Желтый (yellow)
100	100	100	Белый (white)

Обратите внимание на то обстоятельство, что при наложении пары цветов `red + cyan`, `green + magenta` и `blue + yellow` получается белый цвет, т. е. входящие в пару цвета дополняют друг друга. Если `red`, `green` и `blue` образуют базовую (`RGB`) палитру, то `cyan`, `magenta` и `yellow` — дополнительную (`CMY`). Между прочим, негативное изображение на цветной фотопленке получается в цветах дополнительной палитры.

Переход от базовой палитры к дополнительной выполняется так, как показано в табл. 4.1, а для обратного преобразования используются следующие соотношения: `cyan + magenta = blue`, `cyan + yellow = green`, `magenta + yellow = red`.

4.2. Исходная цветовая палитра

Исходная палитра цветов. Для того чтобы при включенном компьютере на экране монитора было видно изображение символов или рисунков в `DAC`-регистры видеокарты должны быть записаны соответствующие коды. Просто очистить все регистры нельзя, поскольку очищенное состояние байтов соответствует черному цвету. Условимся называть *палитрой цветов* ту совокупность кодов, которая может быть записана в `DAC`-регистры видеокарты. Если палитра установлена, т. е. находится в `DAC`-регистрах, то она становится *текущей палитрой* и используется видеоконтроллером при отображении на экран содержимого видеопамяти.

После включения или перезагрузки компьютера в `DAC`-регистры записывается палитра цветов, хранящаяся в `BIOS`. Ее структура не зависит от установленного видеорежима, но в зависимости от установленного режима прикладным задачам доступны только первые 16 или все 256 цветов. В последнем случае принято говорить о стандартной палитре `VGA`.

Программа для визуализации палитры. Словесное описание каждого цвета едва ли позволит наглядно представить, как выглядит эта палитра, целесообразнее составить программу, позволяющую увидеть все 256 цветов на экране монитора. Для упрощения программы надо использовать стандартный графический режим VGA IBM (его код 13h). В таком случае потребуется минимум вспомогательных действий, а сравнительно низкое разрешение позволит получить более наглядное изображение. Текст программы приведен в примере 4.1.

Для получения завершенной задачи надо выполнить следующие действия:

1. Создать исходный файл, содержащий текст примера 4.1. Имя файла вы можете выбрать по своему усмотрению, а тип должен быть asm.
2. Обработать исходный файл Макроассемблером (произвести компиляцию) для получения объектного модуля, имеющего тип obj.
3. Обработать объектный модуль с помощью компоновщика (LINK), в результате чего будет получен файл задачи, имеющий тип exe.

Конкретный способ выполнения перечисленных действий зависит от того, используете вы пакет PWB или нет. Начиная с версии 6.0, Макроассемблер MASM поставляется в комплекте с пакетом PWB (Programmer's WorkBench — инструментальные средства для программирования). В случае использования пакета все перечисленные в списке действия выполняются непосредственно в рабочей среде, которую поддерживает PWB. Если же он не используется, то сначала создается исходный файл с помощью любого текстового редактора. Затем для его обработки вызывается MASM, который создает объектный модуль. Наконец, для построения задачи из объектного модуля вызывается компоновщик. В любом случае при построении задачи примера 4.1 будет выдано сообщение об отсутствии сегмента, содержащего стек. Это простое предупреждение, а не признак ошибки.

Пример 4.1. Программа для визуализации стандартной палитры

```
Code    SEGMENT          ; начало сегмента "code"
        ASSUME  CS:code  ; связь регистра CS с сегментом "code"
start:  mov   ax, 13h    ; код установки видеорежима 13h
        int   10h        ; установка видеорежима
outscr: mov   ax, 0A000h ; A000 — сегмент видеопамяти
        mov   es, ax     ; пишем его в регистр ES
        xor   di, di     ; 0 — начальный адрес видеопамяти
        mov   cx, 200    ; количество строк на экране
lp_1:   push  cx         ; сохраняем счетчик повторов
        mov   cx, 320    ; указываем размер строки
        xor   al, al     ; код первой точки (0)
lp_2:   stosb            ; рисуем точку
        inc   al         ; увеличиваем код точки на 1
```

```
        loop  lp_2        ; управление выводом строки
        pop   cx          ; восстанавливаем счетчик строк
        loop  lp_1        ; управление выводом строк
        mov   ax, 0C01h   ; код функции ожидания ввода
        int   21h         ; DOS ждет нажатия на клавишу
        mov   ax, 03      ; код установки видеорежима 3
        int   10h         ; установка видеорежима
        mov   ax, 4C00h   ; код функции завершения задачи
        int   21h         ; DOS завершает выполнение задачи
code    ends              ; конец сегмента "code"
        end   start       ; конец текста программы
```

Суть выполняемых в программе действий заключается в следующем. На экран последовательно выводится 200 строк. При выводе каждой строки в видеопамять последовательно записывается содержимое регистра `al`, которое в исходе равно 0 и после вывода каждой точки увеличивается на 1. Может показаться, что `al` изменяется от 0 до 319, но это не так. Регистр `al` содержит восемь разрядов, поэтому его содержимое будет монотонно нарастать от 0 до 255, на 256-м шаге оно окажется равным нулю, затем будет снова нарастать и в конце строки достигнет значения 63. Все строки заполнены одинаково, поэтому при выполнении программы на экране возникнут разноцветные полосы ("занавес"), каждая из них показывает, какой цвет закодирован в конкретном `DAC`-регистре. Прежде чем обсуждать получаемую картину, завершим описание программы.

Две первые и две последние строчки программы содержат информацию, относящуюся к ее оформлению. Точка входа в программу имеет метку `start`. Выполнение программы начинается с установки видеорежима `VGA IBM`, его код `13h`, разрешение составляет 320×200 точек, размер палитры 256 цветов.

Далее в регистр `es` записывается код сегмента видеобуфера `A000h`. Прямая запись значений в сегментные регистры невозможна, поэтому используется регистр-посредник `ax`. В регистре `di` устанавливается нулевой адрес, соответствующий началу строки. Пара регистров `es:di` выбрана для того, чтобы записывать коды точек командой `stosb`.

На экране переход с одной строки на другую выполняет видеоконтроллер при достижении конца очередной строки. Программа же просто выводит в цикле `lp_2` количество точек, совпадающее с размером строки для данного режима. Измените 320 на 319 или 321 и картинка "рассыпается", поскольку начало нового цикла вывода не будет совпадать с началом строки на экране.

После заполнения экрана надо выдержать паузу, чтобы вы могли увидеть и оценить полученный результат. Для этого программа обращается к DOS с запросом на ввод символа с клавиатуры. Никаких предупреждающих сообщений на экран не выводится. Возвращение в программу произойдет после того, как вы нажмете одну из информационных клавиш клавиатуры —

букву, цифру, <пробел>, <Enter> и пр. После этого произойдет немедленное завершение задачи (возврат в DOS).

Построенная задача выведет на экран интересующие нас цвета при условии, что палитра установлена. Дело в том, что загрузку палитры при смене режимов работы видеокарты можно запретить, записав 1 в третий разряд слова с адресом `0000:0489` из области данных `BIOS`. Обычно этот разряд очищен, и палитра загружается при любых изменениях режимов (как текстовых, так и графических). Одна из функций прерывания `int 10h`, относящихся к группе `12h`, предназначена для разрешения или запрещения загрузки палитры. При ее вызове в регистре `bl` указывается код `31h`, а в регистре `al` — 0 или 1.

Устанавливаемая DOS палитра в книге [8] описана примерно так (это не цитата, а скорее вольный перевод оригинала). Первые 16 `DAC`-регистров содержат палитру для режима `CGA`, в следующих 16-ти регистрах записаны коды разных оттенков серого цвета. Затем располагаются три основные группы, занимающие по 72 регистра и содержащие коды цветов высокой, средней и низкой интенсивности. Каждая группа делится на 3 одинаковых подгруппы, содержащие коды цветов высокого, среднего и низкого насыщения. Последние восемь регистров просто очищены, им соответствует черная полоса. Тут автор книги [8] допустил неточность, — фактически при установке палитры последние 8 регистров не заполняются. После включения компьютера они очищаются, но их содержимое могут изменить программы, работающие в графических режимах. Поэтому вместо черной полосы, соответствующей последним восьми линиям, вы можете увидеть другой цвет.

Описанная программа позволяет получить качественное представление о цветах палитры, установленной по умолчанию. Если вас интересуют точные значения, т. е. коды этих цветов, то придется составить собственную программу для распечатки содержимого базовых регистров. В следующем разделе рассмотрены функции `BIOS`, позволяющие определить содержимое любого `DAC`-регистра. Здесь мы опишем младшую часть устанавливаемой `BIOS` палитры, которая является палитрой `CGA`.

Стандартная палитра CGA. Установка и поддержка `BIOS` стандартной палитры `CGA` вызвана требованием совместимости с устаревшим программным обеспечением. Программы, создававшиеся для `IBM PC/XT` и `IBM PC/AT`, должны выполняться на современных моделях ПК без каких-либо ограничений. Кроме того, палитра `CGA` нужна при работе в текстовых режимах, которые устанавливаются при первоначальной загрузке компьютера и DOS. Наконец, при описании функций `BIOS` очень часто приводятся коды цветов палитры `CGA`. В этом отношении рассматриваемые в данной книге примеры не являются исключением.

Содержимое первых 16-ти `DAC`-регистров (палитра `CGA`) показано в табл. 4.2. В ней перечислены коды, названия цветов и содержимое байтов соответст-

вующих DAC-регистров. По интенсивности цвета делятся на две группы — средняя и высокая интенсивность, соответственно таблица разделена на две половины (серый цвет является исключением). Коды — это шестнадцатеричные числа. Соответствие между ними и уровнями интенсивности в процентах такое: 3F — 100%, 2A — 67%, 15 -33%.

Поскольку цвет точки зависит от содержимого соответствующего ее коду DAC-регистра, то в дальнейшем, говоря о конкретном цвете, мы будем приводить коды трех базовых цветов (r, g, b), хранящихся в указанной последовательности в байтах DAC-регистра. Например, черному цвету соответствуют коды 0, 0, 0, а белому — коды 3F, 3F, 3F.

Таблица 4.2. *Названия и коды цветов палитры CGA*

Код точки	Название цвета	Коды базовых цветов		
		Красный	Зеленый	Синий
Цвета средней интенсивности				
0	Черный	00	00	00
1	Синий	00	00	2A
2	Зеленый	00	2A	00
3	Циан	00	2A	2A
4	Красный	2A	00	00
5	Фиолетовый	2A	00	2A
6	Коричневый	2A	15	00
7	Белый	2A	2A	2A
Цвета высокой интенсивности				
8	Серый	15	15	15
9	Синий	15	15	3F
A	Зеленый	15	3F	15
B	Циан	15	3F	3F
C	Красный	3F	15	15
D	Фиолетовый	3F	15	3F
E	Желтый	3F	3F	15
F	Белый	3F	3F	3F

Сравните вторую половину табл. 4.2 с табл. 4.1. Вы увидите, что одноименным цветам в них соответствуют разные значения базовых цветов. Напри-

мер, красному цвету высокой интенсивности в табл. 4.2 соответствует тройка кодов 3F, 15, 15, а в табл. 4.1 — 3F, 00, 00. Какой из них считать красным, а какой не совсем красным? Еще один пример. В таблицах редактора CorelDraw фиолетовому цвету средней интенсивности соответствуют коды 26, 00, 26, а высокой интенсивности — 3F, 26, 3F. В табл. 4.2 для этих же названий цветов указаны другие значения.

Эти примеры иллюстрируют тот факт, что наше восприятие цвета весьма субъективно. То, что одному кажется фиолетовым цветом, другой склонен считать пурпурным. Кроме того, наша способность различать близкие цвета весьма индивидуальна. Разница в интенсивности двух близких цветов с кодами 26, 00, 26 и 2A, 00, 2A составляет всего 4/64 = 6.25% и вполне возможно, что она будет неразличима для глаза.

4.3. Функции BIOS

При работе в режимах PPG перед выводом рисунков на экран, как правило, приходится выполнять те или иные действия, связанные с обращением к одному или нескольким DAC-регистрам. Способ доступа к ним зависит от технических особенностей конкретной видеокарты, поэтому расположенное на видеокарте расширение BIOS поддерживает набор функций, выполняющих все необходимые манипуляции с DAC-регистрами. Использование в вашей задаче этих функций обеспечивает ее независимость от технических особенностей установленной на ПК видеокарты.

Общая характеристика группы функций 10h. В состав прерывания BIOS int 10h входит группа функций с названием Set Palette Registers (установка регистров палитры), выполняющих разнообразные действия, связанные с обслуживанием внутренних регистров видеокарты. Набор функций, образующих эту группу, вместе с перечнем входных и выходных параметров был разработан фирмой IBM для стандарта VGA и с тех пор не изменялся. Появление стандарта VESA не добавило ничего нового в способы работы с DAC-регистрами, поскольку в этом не было необходимости.

В данном разделе рассмотрены только те функции, которые используются при программировании для режимов PPG. Полное описание всех функций вы найдете, например, в книгах [5, 8] или в Tech Help.

В англоязычной документации на BIOS, DAC-регистры называют еще регистрами цвета (color registers). Такое название вполне соответствует их назначению, и мы будем его использовать в дальнейшем. Кроме них в составе видеоконтроллера существуют регистры палитры (palette registers). Они не используются при работе в режимах VESA и не имеют никакого отношения к тем палитрам, о которых идет речь в настоящей книге. Их назначение будет описано ниже в данной главе.

Для запроса конкретной функции код группы (10h) помещается в регистр ah, а код запрашиваемой функции — в al. Расположение входных и выходных параметров в регистрах зависит от конкретной функции. BIOS не проверяет допустимость значений параметров. О корректности запроса должен заботиться программист. Вызов функций BIOS, как уже было сказано, выполняет прерывание int 10h. Обращаем внимание на то, что совпадение кодов группы и вектора прерывания является случайным.

Работа с одним регистром. В группу 10h прерывания int 10h включены два запроса, позволяющие записать или прочитать один регистр цвета.

Запрос **1010h "Set One Color Register"** записывает нужный код в один из регистров цвета. Перед его вызовом коды базовых цветов помещаются в регистры dh, ch, cl, соответственно красный, зеленый и синий, а номер регистра цвета указывается в bx.

Замечание

Не забывайте, что при работе с регистрами цвета используется только 6 младших разрядов каждого байта. Содержимое двух старших разрядов регистров dh, ch и cl BIOS просто игнорирует.

Запрос **1015h "Read One Color Register"** выполняет чтение содержимого регистра цвета. Перед вызовом в регистре bx указывается его номер, а содержимое после выполнения запроса находится в регистрах dh, ch, cl, соответственно красный, зеленый и синий. BIOS возвращает шестиразрядные коды базовых цветов.

Замечание

В запросах 1010h и 1015h для указания кода цвета используются одни и те же регистры общего назначения.

В примере 4.2 приведена группа команд, записывающих в последний DAC-регистр видеокарты (0FFh) код яркого белого цвета.

Пример 4.2. Установка содержимого последнего регистра цвета

```
mov    dh, 3Fh          ; интенсивность красного цвета
mov    ch, 3Fh          ; интенсивность зеленого цвета
mov    cl, 3Fh          ; интенсивность синего цвета
mov    bx, 255          ; номер регистра цвета
mov    ax, 1010h        ; код запрашиваемого действия
int    10h              ; выполнение запроса
```

Работа с блоком регистров. В группу 10h прерывания int 10h включены два запроса, позволяющие записать или прочитать сразу несколько (блок) регистров цвета.

Запрос `1012h "Set Block of Color Registers"` записывает коды базовых цветов в несколько (в блок) регистров цвета. Предварительно в оперативной памяти надо сформировать массив, содержащий `N` троек байтов, где `N` соответствует размеру блока. В байтах каждой тройки последовательно указываются шестиразрядные коды красного, зеленого и синего цветов. В литературе такой массив принято называть палитрой используемых цветов. Перед обращением к `BIOS` в регистрах задаются следующие величины: `bx` — номер первого изменяемого `DAC`-регистра, `cx` — количество изменяемых `DAC`-регистров (`N`), `es:dx` — адрес оперативной памяти, соответствующий началу массива кодов устанавливаемых цветов (палитры).

Запрос `1017h "Read Block of Color Registers"` предназначен для копирования содержимого блока регистров цвета в оперативную память. Входные параметры задаются так же, как для запроса `1012h`. В регистре `bx` указывается номер первого копируемого регистра цвета, в `cx` — количество копируемых регистров (`N`), а в `es:dx` — адрес начала массива, размером в `3N` байтов для размещения копируемых значений, где `N` — число, указанное в `cx`.

Подпрограммы сохранения и восстановления палитры. При выполнении графической задачи может возникнуть необходимость изменить уже установленную палитру, а спустя некоторое время восстановить ее. Чаще всего это делается при полном изменении находящейся на экране картинки, или при переходе к другой странице видеопамяти. В таких случаях перед изменением текущей палитры содержимое всех 256-ти регистров цвета надо сохранить в оперативной памяти. Для сохранения текущей палитры в памяти необходимо выделить пространство (буфер) размером в $3 \times 256 = 768$ байтов. Где именно будет расположен этот буфер, не имеет значения, но для возможности его использования в сегменте данных задачи надо выделить два слова и поместить в них смещение (адрес в сегменте) и значение сегмента, содержащего буфер. Зарезервировать эти два слова можно, например, так:

```
BuffPal  dw 0  ; для указания смещения буфера от начала сегмента
         dw 0  ; для указания значения сегмента, содержащего буфер
```

При выполнении задачи в эти слова вместо нулей должны быть записаны конкретные значения сегмента и смещения в нем. Распределение пространства оперативной памяти обычно производится в начале выполнения задачи, как это делается, описано в приложении Б данной книги. Если местонахождение буфера известно при составлении программы, то оно указывается в приведенных выше директивах вместо нулей.

В примерах 4.3 и 4.4 приведены подпрограммы для сохранения текущей и восстановления ранее сохраненной палитры. Обе подпрограммы используют описанную выше переменную `BuffPal` для загрузки сегмента и смещения буфера, выделенного для хранения палитры.

Пример 4.3. Сохранение текущей палитры в буфере

```
SavePal: pusha              ; сохранение "всех" регистров
         push  es           ; сохранение содержимого es
         les   dx, dword ptr BuffPal; сегмент и смещение буфера
         xor   bx, bx       ; номер первого регистра цвета
         mov   cx, 256      ; количество сохраняемых регистров
         mov   ax, 1017h    ; код запрашиваемой функции
         int   10h          ; обращение к функции BIOS
         pop   es           ; восстановление содержимого es
         popa               ; восстановление "всех" регистров
         ret                ; возврат из подпрограммы
```

Пример 4.4. Восстановление исходной палитры из буфера

```
RstPal:pusha                ; сохранение регистров
         push  es           ; сохранение содержимого es
         les   dx, dword ptr BuffPal; сегмент и смещение буфера
         xor   bx, bx       ; номер первого регистра цвета
         mov   cx, 256      ; количество записываемых регистров
         mov   ax, 1012h    ; код запрашиваемой функции
         int   10h          ; обращение к BIOS
         pop   es           ; восстановление содержимого es
         popa               ; восстановление "всех" регистров
         ret                ; возврат из подпрограммы
```

Тексты примеров 4.3 и 4.4 не требуют особых пояснений, напомним только, что команда `les` копирует содержимое первого слова переменной `BuffPal` в регистр `dx`, а второго слова — в регистр `es`. Тексты примеров различаются только кодом запроса, помещаемым в регистр `ax` командой `mov`. При желании, для сохранения или восстановления палитры можно использовать только одну подпрограмму. Из ее текста надо исключить указанную команду `mov`, а код запроса (`1012h` или `1017h`) задавать в регистре `ax` перед вызовом подпрограммы.

Что такое "регистры палитры". В заключение несколько замечаний о назначении регистров палитры. С ними можно работать только в режимах `EGA`, `VGA` и в 16-цветных режимах VESA, а в режимах `PPG` они не доступны. Если вас не интересуют особенности работы в перечисленных режимах, то можно пропустить оставшуюся часть данного раздела.

Появлению стандарта `EGA` сопутствовала разработка мониторов, которые позволяли выводить на экран 64 цвета. Однако в стандарте `EGA` код точки 4-разрядный и, соответственно, доступны только 16 регистров цвета. Для более полного использования возможностей `EGA`-мониторов количество ре-

гистров цвета на видеокартах было увеличено до 64-х. Одновременно работать со всеми регистрами было, по-прежнему, невозможно. Поэтому они делились на четыре одинаковые группы и были введены четыре специальных *регистра палитры*. Хранящееся в них число (от 0 до 3) указывает номер группы из 16-ти регистров цвета, доступной в данный момент времени.

С появлением VGA-мониторов количество регистров цвета на видеокартах увеличилось до 256-ти, и появилась возможность делить их на 4×64 или 16×16 групп. Размер кода точки в стандарте VGA IBM позволяет использовать любой из 256-ти регистров цвета. Тем не менее, для совместимости с режимом EGA и расширения его возможностей деление на группы сохранилось, а у видеоконтроллера появилось 16 регистров палитры.

При работе в режиме VGA IBM (но не VESA) восьмиразрядный код точки рассматривается как две независимые тетрады XY. Содержимое старшей тетрады X является номером регистра палитры (от 0 до F), в котором находится номер одной из 16-ти групп регистров цвета (тоже от 0 до F). Младшая тетрада Y является номером регистра в этой группе. При установке видеорежима в регистры палитры записываются их порядковые номера от 0 до 0Fh, в результате регистры цвета оказываются как бы пронумерованными от 0 до 255, и присутствие регистров палитры просто не заметно. Если же принудительно изменить содержимое регистров палитры, то изменится естественный порядок нумерации регистров цвета.

Дополнение к программе визуализации. В качестве примера приведем дополнение к примеру 4.1, позволяющее увидеть на экране эффект от изменения содержимого регистров палитры. Оно приведено в примере 4.5 и должно быть включено в текст примера 4.1 между командами `int 21h` и `mov ax, 03`.

Напомним, что программа примера 4.1 заполняет экран разноцветными вертикальными линиями, ждет ввод любого символа и прекращает свою работу. После указанных изменений вместо прекращения работы будет выполняться группа команд из примера 4.5, которая записывает в регистры палитры значения, противоположные исходным (0Fh, 0Eh, ..., 1, 0). В результате картинка на экране окажется повернутой вокруг вертикальной оси. Например, если в исходном варианте палитра CGA располагалась в первых 16-ти столбцах на экране, то теперь она будет находиться в столбцах с номерами 240—255, а расположение цветов в этих столбцах будет противоположно исходному.

После выполнения этих действий программа вновь ждет нажатия на одну из клавиш. Это сделано для того, чтобы оператор мог рассмотреть изменения, произошедшие на экране. После нажатия на клавишу выполнение программы будет закончено и произойдет возврат в DOS.

Пример 4.5. Дополнение (вставка) к программе примера 4.1

```
mov   cx, 16      ; количество изменяемых регистров
mov   bx, 0F00h   ; код для записи в нулевой регистр
```

```
lp_3:   mov   ax, 1000h  ; код запрашиваемой функции BIOS
        int   10h        ; установка регистра палитры
        inc   bl         ; номер следующего регистра
        dec   bh         ; код для следующего регистра
        loop  lp_3       ; управление повторами цикла
        mov   ax, 0C01h  ; код функции DOS
        int   21h        ; ждет нажатия на клавишу
```

В примере 4.5 использован запрос 1000h, который производит запись в регистр палитры. Перед его вызовом заполняется регистр bx. В старший байт bh записывается номер группы регистров цвета, а в младший bl — номер регистра палитры.

Содержимое регистра палитры можно прочитать с помощью запроса 1007h. Номер читаемого регистра палитры указывается в bl, а его содержимое после выполнения запроса будет помещено в bh.

В состав группы 10h входят два запроса, которые позволяют прочитать (код 1009h) или записать (код 1002h) сразу все регистры палитры. Кроме того, они считывают или записывают еще один специальный регистр, который называется Overscan или Border. О назначении этого регистра следует поговорить особо.

Регистр Overscan. При работе в любом видеорежиме на экране существует небольшое свободное пространство, расположенное за пределами рабочей области. Окраска этого пространства зависит от содержимого регистра цвета, номер которого хранится в регистре Overscan. Обычно не используемое пространство окрашено в черный цвет, а поскольку код черного цвета находится в нулевом регистре, то Overscan просто очищен.

Некоторые программы, например русификатор Keyrus, изменяют содержимое Overscan для окрашивания неиспользуемого пространства экрана в разные цвета. Это позволяет различать режимы работы программы, например ввод русских или латинских букв, не используя рабочую область экрана. Существуют два запроса — 1001h и 1008h, выполняющие запись и чтение регистра Overscan.

Замечание

В заключение еще раз подчеркнем, что в видеорежимах SVGA деление на страницы действует только при работе с палитрой 16 цветов. При работе с палитрой 256 цветов регистры палитры не используются.

4.4. Простая установка палитры

При подготовке образов точечных, или как их еще называют "растровых", рисунков либо используется палитра, либо цвет указывается непосредственно в коде каждой точки. В режимах PPG можно работать только с рисунками, подготовленными с применением палитры.

Коды точек образа рисунка, использующего палитру, являются порядковыми номерами содержащихся в ней цветов. Поэтому палитра должна быть установлена (записана в регистры цвета видеокарты) до построения рисунка на экране. Если этого не сделать, то цвета точек построенного рисунка будут соответствовать тем, которые находятся в регистрах цвета видеокарты, и вы можете увидеть совсем не ту картинку, которая должна быть.

Формат и место палитры в файле. Палитра или таблица цветов содержит коды цветов, использованных при создании рисунка. В большинстве стандартов код цвета занимает 3 байта, в которых указана интенсивность базовых цветов — красного, зеленого и синего. Если базовые цвета перечислены именно в такой последовательности, то мы будем говорить, что палитра имеет формат `r`, `g`, `b`. Если в такой палитре описано `N` цветов, то она занимает в файле `3N` байтов. Исключением является стандарт `BMP` (см. приложение А). В этом случае строка может содержать три или четыре байта, а палитра занимает в файле соответственно `3N` или `4N` байтов. В трех первых байтах хранятся коды синего, зеленого и красного цветов. Если в строке есть четвертый байт, то он очищен. В таких случаях мы будем говорить, что палитра имеет формат `b`, `g`, `r` или `b`, `g`, `r`, `0`.

Как правило, код базового цвета содержит восемь разрядов и заполняет весь байт. В технической документации принято говорить, что при таком размере кода цвет хранится в формате, независящем от устройства.

Местонахождение и размер палитры также зависит от стандарта, которому соответствует файл. Палитра может предшествовать образу рисунка или находиться после него. Ее размер может быть фиксированным (максимально возможным) или сокращенным за счет включения только тех цветов, которые реально использованы в рисунке.

Мы опишем установку палитры при работе с файлами, соответствующими стандарту `PCX`, аналогичные действия при работе с файлами стандарта `BMP` описаны в приложении А данной книги.

Расположение и варианты палитры PCX. Стандарт `PCX` создавался в то время, когда ПК поддерживали видеорежимы `CGA` и `EGA`. Поэтому в заголовке файла, начиная с адреса `10h`, было зарезервировано 48 байтов, что позволяет разместить 16-цветную палитру. С появлением видеорежима `VGA` возникла необходимость изменения или доработки стандарта `PCX`, поскольку размер заголовка не позволял разместить 256-цветную палитру. Для сохранения совместимости с уже существующими файлами было решено оставить без изменения все, что касалось режимов `CGA` и `EGA`, и расположить 256-цветную палитру в конце файла, после образа рисунка. В этом случае место, занимаемое палитрой `EGA`, не используется.

Первой это сделала фирма Genius Microprogramming в 1988 году. В соответствии с ее версией стандарта `PCX` сразу после образа рисунка располагается байт, содержащий код `0A`, а после него следует палитра формата `r`, `g`, `b`. Ко-

ды базовых цветов сокращены до 6-ти разрядов, т. е. формат палитры полностью соответствует стандарту VGA IBM и она может быть установлена с помощью уже описанного запроса 1012h прерывания int 10h.

В том же году разработчик стандарта PCX фирма ZSoft приняла аналогичные дополнения. Однако, в отличие от версии Genius, байт, расположенный перед палитрой, содержит код 0C, а коды базовых цветов содержат 8 разрядов (независящие от устройства коды). Такая палитра не полностью соответствует стандарту VGA IBM — перед ее установкой все байты надо сдвинуть на 2 разряда вправо.

Таким образом, на сегодняшний день существуют, по крайней мере, два способа хранения 256-цветной палитры в стандарте PCX. Различить их можно по коду, находящемуся в байте, расположенному после образа рисунка.

Как получить доступ к палитре. Для доступа к палитре файл надо открыть, прочитать его заголовок (первые 80h байтов) и извлечь оттуда данные о размерах рисунка и кода точки. Если последний равен восьми, то палитра находится в конце файла после образа рисунка. Расположение нужных величин в заголовке файла формата PCX описано в разделе 3.3.3.

Перед чтением палитры в оперативную память файл надо принудительно позиционировать так, чтобы сохраняемый DOS указатель находился на расстоянии 769 байтов от конца файла. Затем надо прочитать 769 байтов в буфер обмена. После чтения нулевой байт буфера обмена должен содержать код 0A или 0C, в противном случае произошла ошибка, которая означает, что файл не соответствует стандарту PCX, или был неверно позиционирован.

Если нулевой байт буфера обмена содержит код 0C, то перед установкой палитры надо сдвинуть содержимое ее байтов на два разряда вправо. В противном случае (если буфер обмена содержит код 0A) сдвиг не требуется. Теперь формат палитры, находящейся в буфере обмена, соответствует стандарту VGA IBM и ее можно устанавливать (копировать в регистры цвета).

Позиционирование файла. В процессе построения рисунка формата PCX приходится дважды принудительно изменять текущую позицию файла. Рассмотрим, как это делается.

При открытии файла по указанному имени DOS в своем разделе данных создает специальную структуру (или таблицу), в которой хранятся величины, необходимые для доступа к файлу. Порядковый номер этой структуры называется file handle (или просто ссылка), при успешном открытии файла DOS возвращает значение ссылки в регистре ax. Задача должна сохранить ссылку, например, в переменной Handler, и указывать ее при всех последующих обращениях к файлу.

В созданной DOS структуре, помимо прочих величин, хранится указатель позиции, в которой находится файл, сразу после открытия это число 0. При каждом чтении или записи значение указателя увеличивается на количество

прочитанных или записанных байтов. Поэтому если вы последовательно читаете или записываете данные, то дополнительные действия по изменению позиции файла не требуются. Однако в некоторых случаях может понадобиться принудительное изменение позиции (значения указателя) без чтения или записи данных. Для выполнения таких действий предназначена специальная функция DOS, которая называется `Lseek`.

Все функции DOS вызываются через прерывания `int 21h`, а код функции указывается в регистре `ah`. Код функции `Lseek` равен `42h`. В регистре `al` указывается точка, относительно которой выбирается новая позиция, это числа 0, 1 или 2. Соответственно эти числам точкой отсчета является начало, текущая позиция, или конец файла. В паре регистров `cx:dx` задается 32-разрядное число, для получения новой позиции, его значение прибавляется к точке отсчета. При работе с файлами в регистр `bx` всегда записывается ссылка (`file handle`). После выполнения функции DOS возвращает в паре регистров `dx:ax` новое значение указателя, если эта величина используется в задаче, то ее надо сохранить.

Подпрограмма установки палитры PCX. В примере 4.6 приведен текст подпрограммы, выполняющей простую установку палитры из файла формата `PCX`. Предполагается, что файл был открыт, и его заголовок обработан. После возврата из подпрограммы задача должна проверить состояние C-разряда. Если он очищен, то палитра успешно установлена. В противном случае подпрограмма обнаружила ошибку при позиционировании или в структуре файла.

Пример 4.6. Установка 256-цветной палитры из файла PCX

```
SetPpal: pusha                      ; сохранение содержимого регистров
;        Позиционирование на конец файла PCX
         mov   ax, 4202h            ; код запроса позиционирования файла
         mov   bx, Handler          ; указываем ссылку на файл
         mov   cx, -1               ; старшая часть числа -769
         mov   dx, -769             ; младшая часть числа -769
         int   21h                  ; обращение к функциям DOS
         jc    spexit               ; -> ошибка позиционирования
;        Чтение палитры в буфер и проверка признака
         mov   cx, 769              ; размер считываемой порции данных
         call  readf                ; чтение данных в буфер обмена
         xor   si, si               ; нулевой адрес в буфере обмена
         lods  byte ptr fs:[si]     ; чтение в al нулевого байта
         cmp   al, 0Ah              ; в байте находится код 0A ?
         je    spal                 ; -> да, исключаем масштабирование
         cmp   al, 0Ch              ; в байте находится код 0C ?
         je    @F                   ; -> да, выполняем масштабирование
         stc                        ; установка признака ошибки
```

```
        jmp   short spexit        ; -> на завершение подпрограммы
;       Масштабирование (сдвиг содержимого байтов) палитры
@@:     mov   cx, 768             ; указываем размер палитры
mpal:   shr   byte ptr fs:[si], 02; сдвиг очередного байта
        inc   si                  ; адрес следующего байта
        loop  mpal                ; управление повторами цикла
;       Установка палитры (копирование в регистры цвета видеокарты)
spal:   push  es                  ; сохраняем содержимое es
        push  fs                  ; сохраняем содержимое fs
        pop   es                  ; выталкиваем его в es
        mov   dx, 01              ; адрес начала палитры
        xor   bx, bx              ; номер первого регистра цвета
        mov   cx, 256             ; кол-во устанавливаемых регистров
        mov   ax, 1012h           ; код запроса на установку палитры
        int   10h                 ; обращение к функции BIOS
        pop   es                  ; восстанавливаем содержимое es
;       Позиционирование файла на начало рисунка
        mov   ax, 4200h           ; код запроса позиционирования файла
        mov   bx, Handler         ; указываем ссылку на файл
        xor   cx, cx              ; старшая часть величины смещения
        mov   dx, 80h             ; младшая часть величины смещения
        int   21h                 ; обращение к функциям DOS
spexit: popa                      ; восстановление регистров
        ret                       ; возврат из подпрограммы
```

Основные фрагменты подпрограммы отделены друг от друга комментарием. В примере 4.6 используются все регистры общего назначения, поэтому первая команда `pusha` сохраняет их содержимое в стеке. Следующие четыре команды формируют данные для запроса на позиционирование файла. В регистр `al` записывается код 2, являющийся признаком перемещения указателя в заданную позицию, а в пару регистров `cx:dx` — смещение этой позиции от конца файла (отрицательное число 769). Затем происходит обращение к DOS (`int 21h`) для исполнения запроса. Если при позиционировании зафиксирована ошибка, то команда `jc spexit` выполнит переход на метку `spexit` для завершения подпрограммы с установленным С-разрядом.

В случае успешного позиционирования в буфер обмена считываются 769 байтов, содержащих признак `0Ch` или `0Ah` и собственно палитру. Для чтения используется подпрограмма `readf`, текст которой был приведен в примере 3.23. После чтения нулевой байт буфера обмена содержит код типа палитры. Он помещается в регистр `al` и анализируется. Если код равен `0Ah`, то масштабирование исключается. Если код равен `0Ch`, то нужно выполнить масштабирование палитры. Если код отличается от `0Ah` и `0Ch`, то устанавливается С-разряд и происходит возврат из подпрограммы.

Для масштабирования палитры в регистр `cx` записывается ее размер и выполняется цикл сдвигов содержимого каждого байта на 2 разряда вправо.

Начиная с команды, имеющей метку `spal`, формируются исходные данные для функции `BIOS`, устанавливающей палитру. В регистр `es` надо записать значение сегмента буфера обмена (содержимое регистра `fs`), а в регистр `dx` — адрес начала палитры в этом сегменте (1). Номер первого устанавливаемого `DAC`-регистра равен нулю, а количество устанавливаемых `DAC`-регистров равно 256. В регистр `ax` помещается код запроса (`1012h`) и происходит обращение к `BIOS` (`int 10h`). После возвращения из `BIOS` надо восстановить исходное содержимое регистра `es`, сохраненное в стеке.

Палитра установлена и подпрограмму можно было бы завершить. Однако в процессе ее выполнения изменилась исходная позиция файла, и ее надо восстановить. В противном случае перед построением рисунка потребуются лишние действия, связанные с определением и изменением текущей позиции файла.

Образ рисунка отстоит на `80h` байтов от начала файла. Поэтому в примере в регистры `al` и `cx` записывается 0, а в регистр `dx` код `80h`. После позиционирования восстанавливается содержимое регистров (команда `popa`) и происходит возврат из подпрограммы. Если при позиционировании возникнет ошибка, то произойдет возврат с установленным С-разрядом.

При выходе из описанной подпрограммы С-разряд будет установлен не только при ошибках позиционирования файла, но и при отсутствии кодов `0Ah` или `0Ch` перед палитрой. В основной программе уже невозможно определить причину ошибки. Если она важна, то в подпрограмме можно сформировать дополнительные признаки, уточняющие характер ошибки.

После успешной установки палитры можно приступать к построению рисунка на экране. Выбор способа построения зависит от содержимого байта 2 заголовка файла. Если он очищен, то образ рисунка не сжат и для вывода на экран используется подпрограмма `BigDraw` (см. пример 3.22). Если же байт 2 заголовка содержит единицу, то образ рисунка упакован и для распаковки при выводе на экран применяется подпрограмма `PackDrw` (см. пример 3.26).

Таким образом, к этому моменту вы располагаете всей информацией, необходимой для работы с файлами стандарта `PCX`, содержащими рисунки, использующие палитру. О работе с файлами этого же стандарта, но не использующими палитру, мы поговорим в главе 7 (см. раздел 7.5.2).

Замечание

При установке палитры описанным способом она просто копируется в регистры цвета видеокарты. Это допустимо в тех случаях, когда на экране будет находиться только тот рисунок, для которого предназначена установленная палитра. Если же рисунок добавляется к уже существующим, то при смене палитры, почти наверняка, изменятся цвета той части изображения, которая не занята

новым рисунком. Поэтому, в общем случае, описанная процедура недопустима и при установке палитры должны быть приняты меры для сохранения исходной картинки, находящейся на экране.

4.5. Манипуляции с палитрой цветов

При разработке графических программ сравнительно часто приходится решать следующую задачу. На экране находится исходное изображение, при построении которого использованы некоторые регистры цвета видеокарты. К нему добавляется новый рисунок, закрывающий только часть исходной картинки. Цвета не затронутой новым рисунком части экрана должны сохраниться. Поэтому при установке палитры добавляемого рисунка нельзя изменять содержимое уже используемых регистров цвета видеокарты.

В данном разделе будет показан способ записи новых цветов в свободные регистры видеокарты с исключением повторения одинаковых цветов. Он может быть полезен, например, при оформлении "рабочего стола", когда задача выводит на экран заставку, различные рамки, линейки, элементы меню, значки ярлыков и прочие оформительские атрибуты. Необходимые для этого рисунки обычно используют общую палитру, содержащую ограниченное количество цветов. Например, для Windows 3.X и ее приложений создано много рисунков различных значков, хранящихся в виде `BMP`-файлов и пиктограмм, использующих стандартную (для Windows) 16-цветную палитру.

Системная палитра. Для того чтобы каждый раз при добавлении нового рисунка не считывать в оперативную память текущее содержимое регистров цвета видеокарты, имеет смысл постоянно хранить в оперативной памяти копию их текущего состояния. Эту копию мы будем называть *системной палитрой*.

При установке палитры добавляемого рисунка предпринимается попытка разместить ее в системной палитре и если она окажется успешной, то дополненная системная палитра, или только ее дополнение, копируется в регистры цвета видеокарты.

Для размещения точной копии содержимого регистров цвета требуется 768 байтов ($3 \cdot 256$). При работе с системной палитрой использовать трехбайтовый код цвета неудобно, поэтому к трем байтам каждого цвета лучше добавить четвертый пустой байт. Это увеличит размер занимаемой памяти до 1024 байтов, но существенно упростит работу с палитрой.

Большинство вспомогательных рисунков, применяемых для оформления рабочей области экрана, хранится в файлах форматов `BMP` и `ICO`. Поэтому мы выберем следующее расположение кодов базовых цветов в каждой строке системной палитры: синий, зеленый, красный, пустой байт (формат `b`, `g`, `r`, `0`). Такой вариант системной палитры не является точной копией содержимого

регистров цвета видеокарты. Однако хранящиеся в ней коды цветов строго соответствуют кодам, находящимся в регистрах видеокарты.

Учитывая достаточно большой размер палитры, ее не целесообразно хранить в разделе данных задачи. Лучше выделить отдельный сегмент, а в разделе данных зарезервировать следующие два слова:

```
Syspal   dw 0  ; смещение палитры от начала сегмента (обычно 0)
         dw 0  ; код сегмента памяти, в котором хранится палитра
```

Если память распределяет программист при подготовке исходного текста задачи, то место размещения палитры описывается в тексте программы и нули заменяются конкретными значениями. Если же память распределяется динамически, т. е. пространство для размещения палитры выделяется в процессе выполнения задачи, то нужные значения записываются в `Syspal` и `Syspal+2` после того, как будет определен код сегмента для хранения палитры.

В разделе данных задачи надо зарезервировать еще одну переменную, исходное значение которой устанавливается при инициализации палитры:

```
numcol   dw 0  ; количество цветов, хранящихся в системной палитре
```

Инициализация системной палитры. После выделения пространства для системной палитры в ее строки и в регистры цвета видеокарты надо записать коды цветов, которые не будут изменяться во время выполнения задачи. В Windows 3X таких цветов 20, они называются "статическими" и применяются для оформления окон (цвета рамок, фона, текста и пр.). Прикладные задачи могут использовать, но не могут изменять эти цвета.

Мы ограничимся двумя статическими цветами — черным и белым. Код черного цвета (`0`, `0`, `0`, `0`) надо записать в нулевую, а код белого (`3F`, `3F`, `3F`, `0`) в последнюю (255-ю) строку `Syspal`. Инициализация палитры производится в той части программы, где выполняются подготовительные действия. Туда надо вставить следующую группу команд:

```
lgs   di, dword ptr Syspal ; gs:di = адрес Syspal в памяти
xor   eax, eax             ; eax = код черного цвета
mov   gs:[di], eax         ; запись кода в начало Syspal
mov   eax, 003F3F3Fh       ; eax = код белого цвета
mov   gs:[di+1020], eax    ; запись кода в конец Syspal
mov   numcol, 01           ; количество цветов в палитре
; Сюда надо вставить команды из примера 4.2
```

В большинстве случаев нулевой регистр цвета видеокарты уже содержит код черного цвета (его байты очищены). Если это не так, то к перечисленным командам надо добавить очистку нулевого регистра цвета. Содержимое остальных регистров цвета не имеет значения, они считаются свободными, поскольку задача работает только с системной палитрой.

Внимание!

В системную палитру записано 2 цвета, а переменной `numcol` присвоено значение 1. Это сделано потому, что код белого цвета будет обрабатываться нестандартно. По мере заполнения палитры предельно допустимым значением `numcol` является 255, а не 256, т. к. последняя строка палитры занята кодом белого цвета.

Пополнение системной палитры. Вместо простой установки палитры добавляемого рисунка она размещается в системной палитре. При этом код каждого добавляемого цвета последовательно сравнивается с кодами всех цветов, находящихся в системной палитре. Отсутствующий цвет помещается в первую свободную строку системной палитры, а значение переменной `numcol` увеличивается на 1. В результате будут добавлены только те цвета, которых не было раньше. Если в добавляемой палитре повторится один и тот же цвет (такое бывает), то в системную палитру он будет включен только один раз.

Рассмотрим конкретный пример. Предположим, что после инициализации системной палитры на экран выводится рисунок, использующий стандартную для Windows 3X 16-цветную палитру. При ее добавлении в системной палитре появятся 14 новых цветов, поскольку черный и белый там уже находились. Теперь при добавлении на экран любого количества рисунков, использующих 16-цветную палитру Windows, системная палитра не изменяется. В этом случае эффективность описанного алгоритма очевидна.

Добавленные в системную палитру цвета надо записать в регистры цвета видеокарты, иначе их нельзя будет использовать. Запись в регистры видеокарты производится либо при добавлении каждого нового цвета, либо после обработки всех цветов добавляемой палитры. Вы можете выбрать любой из этих вариантов по своему усмотрению, очевидных преимуществ друг перед другом у них нет.

Новые номера цветов. Коды точек образа рисунка являются номерами строк палитры, хранящейся в том же файле. При использовании описанного алгоритма порядок расположения части или всех исходных цветов может измениться. Поэтому при обработке каждого цвета добавляемого рисунка надо запоминать его расположение в системной палитре, а при построении рисунка выполнять перекодировку точек.

Для хранения новых значений кодов точек рисунка в памяти резервируется пространство размером в 256 байтов (максимальное количество цветов палитры). Небольшой размер этого пространства позволяет расположить его в сегменте данных программы, например, так:

```
Index  db 256 dup (?); выделение 256 байтов для массива Index
```

Массив `Index` имеет следующую структуру. Номера байтов соответствуют номерам строк палитры добавляемого рисунка, а содержимое байтов являет-

ся номерами строк системной палитры. При построении рисунка по коду точки выбирается соответствующий байт массива `Index` и его содержимое записывается в видеопамять.

Подпрограмма установки палитры. Текст подпрограммы, выполняющей описанные действия, приведен в примере 4.7. Перед ее вызовом добавляемая палитра должна быть прочитана в буфер обмена, и адрес ее начала указан в регистрах `fs:di` (сегмент : смещение). Размер палитры (количество строк, или описанных в ней цветов) помещается в регистр `cx`. Как прочитать палитру и определить ее размер при работе с файлами формата `BMP`, описано в приложении А данной книги.

Если при добавлении новых цветов произойдет переполнение системной палитры, то перед возвратом из подпрограммы будет установлен С-разряд и восстановлено исходное значение переменной `numcol`.

Пример 4.7. Установка палитры добавляемого рисунка формата BMP

```
AnlsPal: pusha                  ; сохранение "всех" регистров
         PushReg <gs,numcol>    ; сохранение содержимого gs и numcol
         xor   bx, bx           ; исходный номер байта в Index
anls_1:  PushReg <cx,bx>        ; сохранение содержимого cx и bx
         mov   ebx, fs:[di]     ; !! очередной добавляемый цвет
         shr   ebx, 02          ; масштабирование кода этого цвета
         and   ebx, 03F3F3Fh    ; выделение 6-ти младших разрядов
         mov   dx, 0FFh         ; номер белого цвета в SysPal
         cmp   ebx, 03F3F3Fh    ; добавляется белый цвет ?
         jz    anls_4           ; -> да
         lgs   si, dword ptr Syspal; установка сегмента и смещения
         mov   cx, numcol       ; количество цветов в Syspal
         xor   dx, dx           ; очистка номера строки в Syspal
;        Цикл сравнения с цветами, записанными в Syspal
anls_2:  lods  dword ptr gs:[si]; чтение текущего цвета из Syspal
         cmp   eax, ebx         ; сравнение с добавляемым цветом
         jz    anls_4           ; -> цвета совпали
         inc   dx               ; номер следующей строки Syspal
         loop  anls_2           ; управление циклом сравнений
;        Добавляемый цвет отличается от хранящихся в Syspal
         cmp   numcol, 255      ; использованы все регистры цвета ?
         jnz   anls_3           ; => нет
         stc                    ; установка С-разряда
         PopReg <bx,cx,numcol>  ; восстановление bx, cx и numcol
         jmp   short anls_5     ; "короткий" переход на метку anls_5
;        Запись нового цвета в Syspal и в регистр цвета
anls_3:  mov   [si], ebx        ; добавляем новый цвет в Syspal
         inc   numcol           ; увеличиваем счетчик цветов в Syspal
```

```
        mov   cx, bx            ; копируем в cx коды цветов g и b
        shr   ebx, 08           ; сдвигаем код цвета r в регистр bh
        xchg  bx, dx            ; переставляем содержимое bx и dx
        mov   ax, 1010h         ; код запроса "запись в регистр цвета"
        int   10h               ; обращение к BIOS
        xchg  bx, dx            ; восстанавливаем содержимое bx и dx
;       Запись очередного номера в массив Index и управление циклом
anls_4: pop   bx                ; номер текущего байта массива Index
        mov   Index[bx], dl     ; сохраняем номер строки в Syspal
        inc   bx                ; номер следующего байта в Index
        add   di, 04            ; !! адрес следующей строки палитры
        pop   cx                ; восстанавливаем счетчик
        loop  anls_1            ; управление повторами внешнего цикла
        pop   ax                ; выталкиваем значение numcol из стека
anls_5: pop   gs                ; восстанавливаем содержимое gs
        popa                    ; восстанавливаем "все" регистры
        ret                     ; возврат из подпрограммы
```

Выполнение примера 4.7 начинается с сохранения в стеке содержимого всех используемых регистров. Команда `pusha` не работает с сегментными регистрами, поэтому содержимое `gs` сохраняет макрос `PushReg`. Он же помещает в стек исходное значение переменной `numcol` для того, чтобы его можно было восстановить в случае переполнения системной палитры. Третья команда очищает регистр `bx`, который используется как счетчик байтов массива `Index`.

Основные действия выполняются в двух вложенных циклах. Внешний имеет метку `anls_1`, а внутренний — `anls_2`.

Внешний цикл повторяется столько раз, сколько цветов (строк) содержит палитра добавляемого рисунка. Эта величина указывается в регистре `cx` при вызове подпрограммы. Оба цикла используют общий регистр `cx` для хранения счетчика повторов, поэтому выполнение внешнего цикла начинается с сохранения в стеке его содержимого. Кроме того, в стеке сохраняется содержимое `bx`, поскольку его изменяет следующая команда. Обе величины сохраняет макрос `PushReg`.

После сохранения содержимого `cx` и `bx` в регистр `ebx` считывается код очередной строки палитры добавляемого рисунка. Коды базовых цветов в ней сокращаются до шести разрядов путем сдвига содержимого регистра `ebx` на два разряда вправо и выделения шести младших разрядов каждого байта.

Код белого цвета расположен в конце системной палитры, поэтому во внешнем цикле надо проверить, не является ли анализируемый цвет белым. Если это так, то выполняется переход на метку `anls_4` для записи номера белого цвета `FF` в очередной байт массива `Index`. В противном случае проис-

ходит подготовка к выполнению цикла сравнений. Для этого в регистры `gs:si` загружается адрес системной палитры, в регистр `cx` помещается количество хранящихся в ней цветов, а регистр `dx` очищается.

Первые пять команд внутреннего цикла выполняют сравнение кода цвета, находящегося в регистре `ebx`, с кодами строк системной палитры. При совпадении происходит переход на метку `anls_4` для записи текущего номера строки системной палитры в очередной байт массива `Index`. В противном случае сравнение продолжается, пока не будут проверены все коды.

После завершения цикла сравнений надо проверить, осталось ли в `Syspal` место для записи нового цвета. Если осталось, то выполняется переход на метку `anls_3`. В противном случае устанавливается признак переноса, восстанавливаются сохраненные в стеке величины и происходит возврат на вызывающий модуль.

Если в `Syspal` есть свободное место, то код добавляемого цвета записывается в первую свободную строку, значение `numcol` увеличивается на 1 и происходит обращение к `BIOS` для записи кода в регистр цвета видеокарты, номер которого совпадает с номером строки системной палитры. Затем выполняется фрагмент, первая команда которого имеет метку `anls_4`.

Из стека восстанавливается содержимое регистра `bx` и в указанный в нем байт массива `Index` записывается номер строки системной палитры, содержащей анализируемый цвет. Находящийся в регистре `di` адрес увеличивается на 4 так, чтобы он указывал начало следующей строки добавляемой палитры. Из стека восстанавливается содержимое регистра `cx` и команда `loop anls_1` повторяет выполнение внешнего цикла до тех пор, пока не будут обработаны все строки (цвета) палитры добавляемого рисунка.

Если добавление палитры закончилось успешно, то изменять значение переменной `numcol` нельзя, поэтому сохраненное в стеке значение выталкивается в регистр `ax`. Затем восстанавливается содержимое регистра `gs` и всех остальных регистров и происходит возврат на вызывающий модуль.

Если для установки добавляемой палитры недостаточно места, то при возврате из подпрограммы `AnlsPal` будет установлен С-разряд регистра флагов и сохранено исходное значение `numcol`. Поэтому при неудачной попытке установки палитры цвета, записанные в системную палитру и регистры видеокарты, не учитываются при последующих обращениях к подпрограмме `AnlsPal`.

Изменения в тексте подпрограммы. Как уже говорилось, пример 4.7 составлен исходя из предположения, что палитра добавляемого рисунка хранится в файле в формате `b, g, r, 0`. Если по каким-то соображениям вам нужно работать с палитрами формата `r, g, b`, то в тексте примера 4.7 надо изменить две команды. Комментарий к заменяемым командам начинается с двух восклицательных знаков.

Для размещения в регистре ebx трех базовых цветов текущей строки палитры добавляемого рисунка команда mov ebx, fs:[di] заменяется приведенной в примере 4.8 группой, состоящей из пяти команд.

Пример 4.8. Преобразование формата r, g, b в формат b, g, r, 0

```
xor     ebx, ebx          ; очистка регистра ebx
mov     bh,  fs:[di]      ; чтение в bh кода красного цвета
mov     bl,  fs:[di+1]    ; чтение в bl кода зеленого цвета
shl     ebx, 08           ; сдвиг ebx на 8 разрядов влево
mov     bl,  fs:[di+2]    ; чтение в bl кода синего цвета
```

И второе изменение. Строка палитры формата r, g, b занимает в памяти три байта, поэтому команду переадресации add di, 04 надо заменить командой add di, 03.

Подпрограмма построения строки. В случае успешной установки палитры можно выводить рисунок на экран, используя для перекодировки его точек сформированный массив Index. Для этого нужна подпрограмма, выполняющая запись в видеопамять точек строки с их перекодировкой.

Текст подпрограммы приведен в примере 4.9. Входные параметры для нее указываются так же, как и для всех подпрограмм, описанных в разделе 3.3.1. Пара регистров fs:si содержит адрес начала образа строки в оперативной памяти, регистр di задает адрес видеопамяти, начиная с которого записываются коды точек. Как обычно, должно быть установлено окно видеопамяти, содержащее адрес, указанный в регистре di, а в регистр es помещен код видеосегмента. Количество точек в строке помещается в регистр cx. Предполагается, что массив Index находится в разделе данных.

Пример 4.9. Вывод строки рисунка с изменением кодов точек

```
drawline: push  bx            ; сохранение содержимого bx
          lea   bx, Index     ; bx = адрес массива Index
drwln_1:  lods  byte ptr fs:[si]; чтение исходного кода точки
          xlat                ; перекодировка al = [bx + al]
          stosb               ; запись кода точки в видеопамять
          or    di, di        ; начало нового сегмента ?
          jne   @F            ; -> нет, обход команды call NxtWin
          call  NxtWin        ; установка следующего окна
@@:       loop  drwln_1       ; управление повторами цикла
          pop   bx            ; восстановление содержимого bx
          ret                 ; возврат из подпрограммы
```

"Изюминкой" примера 4.9 является перекодировка точек рисунка с помощью команды xlat. При ее выполнении суммируется содержимое регистров bx

и `al` и в регистр `al` копируется содержимое байта, расположенного по вычисленному адресу. Остальные действия, выполняемые в примере, уже неоднократно обсуждались, поэтому мы не будем повторяться.

Недостаточно места в системной палитре. Ограниченное количество цветов, которые можно одновременно вывести на экран, является "Ахиллесовой пятой" видеорежимов `PPG`. Вместе с тем, именно по этой причине код точки занимает всего 1 байт, и манипуляции с графическими объектами выполняются достаточно просто и быстро.

В общем случае задача одновременного вывода на экран двух рисунков, использующих разные палитры цветов, неразрешима. Ее можно решить только при определенных ограничениях на размеры палитр и разнообразие описанных в них цветов.

Например, при разработке простых компьютерных игр все рисунки для конкретной игры подготавливаются с использованием цветов единой палитры, которая устанавливается один раз, и в дальнейшем не изменяется.

В семействе Windows 3X применяется комбинированное решение. Рисунки, предназначенные для оформления рабочего стола, как уже говорилось, используют стандартную 16-цветную палитру, которая мало чем отличается от палитры `CGA`, приведенной в табл. 4.2. Для хранения этих (статических) цветов выделено 20 строк системной палитры и столько же регистров цвета видеокарты.

Остальные 236 строк системной палитры отведены для цветов заставок и для нужд прикладных задач. У многоцветных рисунков часть цветов просто изменяется. Разумеется, при сокращении цветов рисунок несколько отличается от оригинала, но что остается делать, если в вашем распоряжении есть только ограниченное число регистров цвета.

ГЛАВА 5

Работа с текстом

При выполнении графических задач на экран выводятся различные текстовые сообщения. Это могут быть названия окон, пояснения к выбранным значкам, информационные строки различного назначения, подсказки оператору и т. п. Программирование вывода текста при работе в графических режимах имеет свои специфические особенности, которые описаны в данной главе.

Все видеорежимы делятся на текстовые и графические. Первые предельно упрощают работу с текстом, но исключают возможность работы с рисунками. Вторые позволяют работать только с отдельными точками, из которых, как известно, складываются любые рисунки, в том числе и изображения символов текста. В соответствии с этим данная глава делится на две основные части, в первой описана работа в текстовых режимах, а во второй — в графических.

В связи с непрерывным совершенствованием технических характеристик мониторов и видеокарт и широким распространением Windows и ее приложений текстовые режимы отошли на второй план и потеряли свою былую значимость. Однако они входят в число стандартных режимов VESA, поэтому автор счел целесообразным описать их в первом разделе главы.

5.1. Текстовые режимы

Текстовые режимы отличаются от графических следующими особенностями:

- видеобуфер расположен в сегменте `B800h` (а не `A000h`);
- в видеобуфере хранятся коды символов в стандарте `ASCII` и их атрибуты;
- преобразование кодов символов в рисунки выполняет видеоконтроллер;
- видеоконтроллер формирует изображение специального текстового курсора.

Стандартом VESA предусмотрено пять текстовых режимов высокого разрешения, имеющих коды от 108h до 10Ch (см. табл. 1.1). Большинство видеокарт поддерживает только два из них — 109h и 10Ah. Поэтому, так же как при работе с графикой, до или после установки режима надо проверить, поддерживает его видеокарта или нет.

Если в соответствии с рекомендациями, приведенными в главе 2, ваша задача выбрала из массива Info значения переменных, перечисленные в примере 2.11, то переменная Vbuff будет содержать код видеосегмента B800h, переменная Horsize — количество символов в строке, а Versize — количество строк на экране. При выполнении подготовительных действий необходимо проверить состояние второго разряда нулевого байта массива Info (разряды пронумерованы начиная с нуля). Если он содержит 1, то BIOS и DOS поддерживают работу в выбранном режиме. Поэтому все приемы программирования, изложенные, например, в книге [8], остаются в силе. В текстовых режимах VESA изменяются только размер и количество строк на экране. В данном разделе описано программирование вывода текста на экран монитора в режимах VESA.

5.1.1. Русский текст на экране

Прежде всего, разберемся с тем, как появляются символы на экране. Задача самостоятельно или с помощью функций BIOS помещает выводимый текст в видеопамять. Коды символов должны соответствовать стандарту ASCII (American Standard Code for Information Interchange — американский стандартный код для обмена информацией). После кода каждого символа в видеопамять записывается атрибут, назначение которого будет описано ниже.

Таблицы знакогенератора. При отображении содержимого видеопамяти в текстовых режимах видеоконтроллер последовательно выбирает коды символов из видеобуфера. По коду символа вычисляется адрес начала его рисунка, который и выводится на экран. Знакогенератором называется *область видеопамяти*, в которой размещаются таблицы символов.

Таким образом, изображение символов, которое мы видим на экране при работе с текстом, зависит от используемой таблицы знакогенератора. Его размеры позволяют загрузить 8 разных таблиц, содержащих по 256 рисунков символов (в режиме EGA только 4 таблицы).

При установке видеорежима после включения компьютера или при переходе от одного видеорежима к другому BIOS загружает в знакогенератор свои таблицы символов из ROM BIOS. По понятным причинам в них отсутствуют рисунки русских букв. Для поддержки работы с русским текстом в процессе загрузки DOS вызывается программа-русификатор, которая остается резидентной в памяти компьютера. Она выполняет множество функций, в том числе отслеживает все прерывания int 10h, и при смене текущего видеоре-

жима загружает в знакогенератор таблицы с рисунками русских букв. Пока русификатор находится в оперативной памяти, русский текст будет выводиться при установке любого стандартного видеорежима. Одним из распространенных русификаторов является программа Keyrus Д. Гуртяка.

Установка режимов VESA происходит без использования стандартной команды "Set Video Mode" прерывания int 10h, поэтому русификатор не "замечает" смены видеорежима и не заменяет русскоязычными базовые шрифты, загруженные из ROM BIOS. Такую замену должна произвести задача, работающая с русским текстом в режимах VESA. При этом она может загрузить собственную таблицу или "заставить" русификатор загрузить одну из его таблиц. В любом случае задача обращается к одной из функций BIOS.

Группа функций 11h. В состав прерывания int 10h входит группа, состоящая из 15-ти функций. Они выполняют загрузку знакогенератора, изменение количества строк и столбцов на экране, установку таблиц символов для текстового и графического режимов и получение информации об используемых таблицах символов. Группа создавалась для обслуживания мониторов EGA и была несколько расширена для мониторов VGA. Расширение заключалось в том, что добавились шрифты высотой в 16 строк и возможность загружать в знакогенератор до 8 (вместо 4) разных таблиц. При запросах функций код группы (11h) указывается в регистре ah, а код нужной функции — в регистре al, его значение может изменяться от 0 до 30h. Реально используются не все значения из этого интервала. Описание всей группы можно найти в книгах [5, 8], в TECH HELP и в других руководствах. Ниже мы рассмотрим только те функции, которые нужны для установки русифицированных таблиц.

Загрузка собственной таблицы. Функции BIOS позволяют загрузить в знакогенератор собственные таблицы выполняемой задачи или таблицы, принадлежащие русификатору. В первом случае используется запрос 1100h.

Запрос 1100h "Load User-defined Font" предназначен для загрузки в знакогенератор таблицы, указанной при обращении к BIOS. Перед изданием запроса в регистры помещаются следующие данные: es:bp — адрес начала загружаемой таблицы в оперативной памяти, cx — количество загружаемых символов, dx — порядковый номер первого символа (начиная с 0), bh — размер рисунка символа в байтах, bl — порядковый номер таблицы в знакогенераторе, который может изменяться от 0 до 7. По умолчанию доступна нулевая таблица, поэтому если вы выберете значение bl отличное от нуля, то для работы с таблицей придется принимать специальные меры.

Размеры символов таблицы должны соответствовать характеристикам установленного видеорежима, например, для режима VESA 109h они составляют 8×16 точек, а для режима 10Ah — 8×8 точек. Здесь первая цифра указывает ширину, а вторая высоту символа. Для большинства текстовых режимов ширина символов составляет 8 точек.

Таблица должна располагаться в выделенном для задачи пространстве оперативной памяти. Если при выполнении задачи таблица загружается в знакогенератор один раз, то постоянно держать ее в оперативной памяти не целесообразно. Ее можно хранить в отдельном файле и в нужный момент прочитать, например, в буфер обмена.

В любом случае при загрузке надо знать полный адрес таблицы (сегмент и смещение в нем). Для хранения этих величин в разделе данных программы выделяются два слова, например:

```
RsFnt16: dw  0 ; Смещение начала таблицы в указанном ниже сегменте
         dw  0 ; Значение сегмента, в котором расположена таблица
```

Если расположение таблицы известно при составлении программы, то вместо нулей в директивах указываются конкретные значения сегмента и смещения. В противном случае содержимое обоих слов формируется при выполнении задачи.

Пример загрузки таблицы. Фрагмент программы, приведенный в примере 5.1, составлен с учетом того, что адрес таблицы указан в переменной `RsFnt16`, а размер символов таблицы составляет 8×16 точек.

Пример 5.1. Загрузка таблицы символов 8×16 в знакогенератор

```
push    es              ; сохранение содержимого es
les     bp, dword ptr RsFnt16; es:bp = адрес начала таблицы
mov     cx, 256         ; количество символов в таблице
xor     dx, dx          ; смещение первого символа в таблице
mov     bh, 16          ; количество байтов на символ
xor     bl, bl          ; номер таблицы в знакогенераторе
mov     ax, 1100h       ; код запроса "загрузить таблицу"
int     10h             ; выполнение запроса
pop     es              ; восстановление содержимого es
```

В приведенном примере таблица загружается полностью, что, вообще говоря, делать не обязательно. Первая половина любой таблицы содержит латинский алфавит, цифры, знаки препинания и пр., она является стандартной и уже находится в знакогенераторе. Поэтому можно хранить (в файле или в памяти) и загружать в знакогенератор только вторую половину таблицы, содержащую русские буквы, символы псевдографики и несколько специальных символов. Для загрузки второй половины таблицы в примере 5.1 в регистры `cx` и `dx` надо помещать значение 128.

Использование таблиц русификатора. Собственная таблица делает выполнение задачи независимым от наличия русификатора на компьютере. Отсутствие русификатора не такая уж редкость, если пользователи не работают в среде DOS, то он просто не нужен.

Если же русификатор установлен, то можно "заставить" его загружать свои таблицы. Русификатор контролирует все запросы, имеющие отношение к таблицам символов. В частности, при попытке загрузить одну из таблиц ROM BIOS, он загружает свою (хранящуюся в ОЗУ) таблицу с русскими шрифтами. BIOS позволяет загрузить из ROM три таблицы со шрифтами следующих размеров:

- запрос 1101h "Load ROM 8x14 Character Font";
- запрос 1102h "Load ROM 8x8 Character Font";
- запрос 1104h "Load ROM 8x16 Character Font".

Перед изданием этих запросов заполняется только регистр bl, содержащий номер таблицы в знакогенераторе (обычно нуль), все остальные величины функция BIOS формирует самостоятельно. В примере 5.2 приведен фрагмент программы, загружающий таблицу шрифтов 8×16 из ROM.

Пример 5.2. Загрузка таблицы символов 8×16 точек из ROM BIOS

```
xor     bl, bl          ; номер таблицы в знакогенераторе
mov     ax, 1104h       ; запрос "загрузка таблицы ROM 8x16"
int     10h             ; выполнение запроса
```

Данный пример проще примера 5.1 и исключает необходимость хранения таблицы в теле задачи, но при отсутствии установленного русификатора в знакогенераторе окажется англоязычная таблица из ROM BIOS.

5.1.2. Общая характеристика процесса вывода текста

При выводе текста, для записи кодов символов в видеопамять, задача может использовать поддержку BIOS и DOS или делать это самостоятельно. Мы опишем оба способа вывода текста на экран, но предварительно обсудим общие особенности программирования работы с текстом.

Из личного опыта работы с компьютером вы знаете, что символы и окружающий их фон могут иметь разные цвета. Например, после загрузки DOS текстовые сообщения выводятся белыми символами на черном фоне, а на обеих панелях оболочки Norton Commander фон имеет синий цвет (если разрешена работа с цветом). Таблицы шрифтов не содержат никакой информации о цветах символов и фона, их формирует задача в процессе вывода текста. Покажем, как это делается.

Атрибуты символов предназначены для раскрашивания выводимого на экран текста. В видеопамяти код атрибута располагается после кода символа, т. е. четные байты видеопамяти содержат коды символов, а нечетные — ко-

ды их атрибутов. Если слово видеопамяти прочитать в один из регистров общего назначения, например в ax, то код символа окажется в младшем байте регистра (al), а код атрибута — в старшем байте (ah).

Для размещения символа на экране всегда выделяется прямоугольная область, которую в литературе принято называть "знакоместо". Размер знакоместа (количество точек по горизонтали и вертикали) зависит от видеорежима. Для стандартных текстовых режимов его размеры составляют 8×8, 8×14 или 8×16 точек. Собственно изображение символа занимает часть знакоместа, в англоязычной документации ее принято называть передним планом (foreground). Свободная часть знакоместа, не занятая рисунком символа, называется задним планом (background), мы будем называть ее фоном. Например, изображение символа "пробел" состоит только из фона.

Байт атрибута рассматривается как три группы независимых разрядов. Младшая тетрада (разряды 0—3) содержит код цвета точек изображения символа (foreground). Следующие три разряда (4—6) содержат код цвета фона (background). Старший (седьмой) разряд байта атрибута управляет миганием точек символа (foreground flashes). Его установка разрешает, а очистка запрещает мигание символа на экране.

Коды цветов точек изображения символа могут принимать значения от 0 до F, а фона от 0 до 7. Если задача не изменяла установленную по умолчанию палитру, то указанным кодам соответствуют цвета стандартной палитры CGA (см. табл. 4.2). Следовательно, точки фона могут иметь следующие цвета: черный, синий, зеленый, циан, красный, фиолетовый, коричневый или белый. Для раскрашивания точек символа можно дополнительно использовать цвета второй половины табл. 4.2.

Вы можете выбрать другие цвета для раскраски изображения символов и фона, для этого задача должна установить собственную палитру в 16-ти младших регистрах цвета видеокарты. Способы установки палитры не зависят от видеорежима. Используемые для этого функции BIOS описаны в предыдущей главе.

На практике наиболее часто используются атрибуты, имеющие следующие коды:

- 07 — белые символы на черном фоне;
- 0F — яркие белые символы на черном фоне;
- 1F — яркие белые символы на голубом фоне.

В процессе загрузки ПК BIOS заполняет видеобуфер кодами символа "пробел" (20h) и кодами атрибутов 07, поэтому мы видим белые символы на черном фоне.

Таким образом, для раскрашивания выводимого текста вы должны выбрать подходящие коды атрибутов и предусмотреть их запись в видеопамять после

кодов символов, к которым они относятся. Если задача записывает в видеопамять только коды символов, то сохраняются те значения атрибутов, которые уже находятся в нечетных байтах.

Управление текстовым курсором. Текстовый курсор — это мигающий символ прямоугольной формы, формируемый видеоконтроллером при работе в текстовых режимах. После загрузки DOS он имеет форму горизонтальной черты, расположенной в двух нижних строчках знакоместа, в которое будет помещен очередной введенный символ. Прикладные задачи могут изменять количество строк в изображении курсора и их расположение в прямоугольнике, но ширину курсора изменить невозможно, она зависит от установленного видеорежима. Кроме того, задачи могут гасить (выключать) курсор, но не могут изменить частоту его миганий.

В соответствии со стандартом VGA IBM в состав видеоконтроллера входят четыре однобайтовых регистра, содержащие следующие характеристики курсора:

- регистр 10 (0Ah) — первая строка рисунка курсора в прямоугольнике;
- регистр 11 (0Bh) — последняя строка рисунка курсора в прямоугольнике;
- регистр 14 (0Eh) — старший байт адреса курсора в видеопамяти;
- регистр 15 (0Fh) — младший байт адреса курсора в видеопамяти.

Доступ к этим регистрам производится через порт с адресом 3D4h, если монитор цветной, или с адресом 3B4h, если монитор черно-белый.

В регистрах 10 и 11 номер строки располагается в 5-ти младших разрядах (0—4) и может изменяться от 0 до 31. Отсчет строк ведется сверху вниз. Если, например, высота символов равна 16-ти точкам (стандартная для VGA) и рисунок курсора занимает две последние строчки прямоугольника, то в регистрах 10 и 11 находятся коды 0Eh и 0Fh.

Вам не обязательно программировать работу с портами, хотя в данном случае это не сложно. Функция 01 прерывания int 10h предназначена для записи данных в регистры 10 и 11. Перед ее использованием в регистрах ch и cl указываются номера первой и последней строк рисунка курсора. В регистр ah помещается код запроса (01) и выполняется команда int 10h. При выполнении этой функции новая форма курсора запоминается в слове 0460 области данных BIOS (см. пример 5.3).

Расположение (позицию) курсора или символа на экране удобнее задавать не в виде адреса, а в виде номера строки и столбца, на пересечении которых он должен находиться. Если строки и столбцы пронумерованы начиная с нуля, то для вычисления *адреса слова* видеопамяти надо умножить номер строки на количество символов в строке и к произведению прибавить номер столбца. Если нужен номер байта (для записи кода символа), то полученный результат умножается на 2. Количество символов в строке является одной из

характеристик видеорежима. BIOS сохраняет эту величину в своей области данных, и в нужных случаях ее используют функции прерывания int 10h. Эту величину можно прочитать из массива Info, описанного в главе 2, или из слова 044Ah области данных BIOS (см. пример 5.3).

Задача может самостоятельно пересчитывать координаты в адрес и записывать его в регистры видеокарты (14 и 15). Однако если при выводе текста используется поддержка BIOS, то лучше обратиться к специальной функции 02 прерывания int 10h. Она не только выполняет указанные вычисления, но и сохраняет значения строки и столбца в одном из 8-ми слов, расположенных в области данных BIOS (см. пример 5.3). Сохраненные координаты курсора используются процедурами, выводящими текст на экран. Для вызова функции 02 код запроса (02) записывается в регистр ah, номера строки и столбца, на пересечении которых должен располагаться рисунок курсора указываются в регистрах dh и dl, а в регистр bh помещается номер страницы видеопамяти, на которой располагается курсор (см. ниже). После этого выполняется команда int 10h.

Изображение курсора можно удалить с экрана тремя разными способами:

1. Запретить его вывод, указав в регистре 10 код 20h.
2. Записать в регистр 10 код равный высоте символов.
3. Переместить рисунок курсора за пределы рабочей области экрана.

Для перемещения курсора за пределы рабочей области его помещают в строку, которая не выводится на экран, но этого лучше не делать по причинам, описанным ниже.

Текстовый курсор является самостоятельным рисунком, а его увязка с процессами ввода и вывода символов производится программно. При вводе и редактировании текста курсор нужен для привлечения внимания оператора. Он указывает позицию на экране, в которую будет помещен очередной символ при вводе с клавиатуры.

Если процесс вывода текста не связан с процессом ввода, то изменение позиции курсора на экране не несет никакой смысловой нагрузки. Он может указывать конец выведенной строки, но едва ли такая информация пригодится тому, кто читает выводимый текст.

Разработчики BIOS использовали курсор для указания позиции, в которую *выводится* каждый символ текста. Это значит, что при вызове процедур BIOS координаты выводимого символа не указываются явно. В качестве координат процедуры используют текущую позицию курсора, сохраняемую в области данных BIOS. Если же курсор был перемещен за пределы рабочей области, то и выводимый текст окажется там же и не будет виден.

Сейчас трудно судить, почему был выбран такой способ позиционирования при выводе текста. Возможно, в свое время, это решение было оправдано,

но постепенно оно превратилось в серьезное препятствие, ограничивающее возможности применения поддержки BIOS.

Страницы видеопамяти. Если не принять специальные меры, то в текстовых режимах задача работает только с младшей частью сегмента видеопамяти, размер которой зависит от установленного видеорежима. Для доступа ко всему пространству видеобуфера оно делится на страницы, которые могут использоваться независимо друг от друга.

Создание страниц и их отображение на экране при работе в графических режимах VESA обсуждалось в разделе 2.5. В текстовых режимах (неважно IBM или VESA) страницы располагаются в одном сегменте видеопамяти, что существенно упрощает их создание и отображение на экране.

При делении видеопамяти на страницы полный адрес для записи символа состоит из двух частей — точки расположения символа на странице и смещения страницы от начала сегмента видеобуфера. Кроме того, записанный в видеопамять символ будет виден на экране, только в том случае, если он расположен на активной, т. е. отображаемой на экране в данный момент времени, странице.

При установке текстовых видеорежимов BIOS вычисляет необходимый для работы объем видеопамяти. Если режим позволяет выводить на экран M строк, содержащих по N символов каждая, то для хранения кодов символов вместе с атрибутами требуется M*N*2 байта видеопамяти. Для определения размера страницы это произведение увеличивается до ближайшего значения, кратного числу 4096, и сохраняется в слове 044Ch области данных BIOS. В зависимости от видеорежима страница может занимать 4096, 8192 или 16384 байта. Соответственно, в сегменте видеопамяти может располагаться 16, 8 или 4 страницы.

Для хранения позиции курсора на каждой странице в области данных BIOS (см. пример 5.3) выделяется слово, содержащее номер строки (старший байт) и столбца (младший байт), на пересечении которых он расположен. Всего в области данных зарезервировано восемь слов, в соответствии с количеством страниц, поддерживаемых BIOS. В исходном состоянии эти слова очищены, что соответствует расположению курсора в начале каждой страницы.

Номер страницы указывается при вызове всех функций прерывания int 10h, выполняющих вывод символов или строк, но *фактически* функции 09, 0Ah и 0Eh прерывания int 10h размещают выводимые символы на нулевой странице. Реально номер страницы использует только функция 13h. По нему она определяет текущее положение курсора на странице и пересчитывает его в относительный адрес (адрес на странице). Затем номер страницы умножается на ее размер и получается смещение начала страницы в видеобуфере. Сумма этих двух величин и является адресом, начиная с которого записываются выводимые символы.

Страница является активной, если ее содержимое отображается на экране. В соответствии со стандартом VGA IBM в состав видеоконтроллера входят два однобайтовых регистра, содержащих адрес видеопамяти, начиная с которого ее содержимое выводится на экран. В регистре 12 (0Ch) хранится старший байт этого адреса, а в регистре 13 (0Dh) — младший. Доступ к этим регистрам осуществляется через порт 3D4h, если монитор цветной, или 3B4h, если он черно-белый.

После включения или перезагрузки компьютера BIOS очищает указанные регистры, поэтому активной является нулевая страница, расположенная в начале сегмента видеобуфера. Для изменения отображаемой области надо записать в регистры 12 и 13 новый адрес видеопамяти.

Задача может самостоятельно изменять адрес начала отображаемой области, но если при выводе текста используется поддержка BIOS, то для смены активной страницы лучше вызывать функцию 05 прерывания int 10h. Перед ее вызовом в регистре al указывается номер новой активной страницы, а в регистре ah — код 05. Эта функция извлекает размер страницы из слова 044Ch области данных BIOS, умножает его на указанный номер и результат помещает в регистры 12 и 13. На экране появляется содержимое новой страницы. Кроме того, функция записывает в слово 044Eh (пример 5.3) смещение активной страницы от начала сегмента видеобуфера.

Фрагмент области данных BIOS. При общей характеристике BIOS в разделе 1.3.2 говорилось, что ее процедуры используют пространство оперативной памяти, называемое областью данных BIOS. Оно расположено в нулевом сегменте оперативной памяти ПК, начиная с адреса 400h. В частности, при выводе символов на экран функции прерывания int 10h используют следующие слова и байты, расположенные в области данных.

Пример 5.3. Фрагмент области данных BIOS

```
0000:0449 — (байт)   установленный видеорежим;
0000:044A — (слово)  количество столбцов (размер строки);
0000:044C — (слово)  размер страницы в байтах;
0000:044E — (слово)  смещение активной страницы от начала видеобуфера;
0000:0450 — (8 слов) позиции курсора на восьми страницах видеопамяти;
0000:0460 — (слово)  номера первой и последней строк рисунка курсора;
0000:0462 — (байт)   номер активной страницы;
0000:0463 — (байт)   базовый адрес видеоконтроллера (3D4h или 3B4h);
0000:0484 — (байт)   номер последней строки на экране;
0000:0485 — (слово)  размер рисунка символа в байтах.
```

Перечисленные в этом примере величины могут пригодиться в тех случаях, когда вы программируете вывод текста без использования функций BIOS.

5.1.3. Вывод текста с использованием поддержки DOS и BIOS

Вывод с помощью функции DOS. Проще всего вывести строку текста на экран с помощью специальной функции DOS, имеющей код 09, поэтому этот способ часто встречается на практике. Перед обращением к DOS адрес начала строки помещается в регистры ds:dx, а код запрашиваемой функции (09) — в регистр ah, после чего вызывается программное прерывание int 21h, которое выполняет обращение к DOS.

Предположим, для определенности, что выводимый текст хранится в разделе данных и оформлен одним из следующих способов:

```
commun  db 'Проверка возможности вывода текста в режиме SVGA$'
commun  db 'Проверка возможности вывода текста в режиме SVGA', 24h
```

Для функции 09 признаком конца строки является код 24h, которому обычно соответствует изображение знака доллара $. В первой из двух приведенных строк он записан в виде символа, а во второй — в виде кода. Если вы забудете указать признак конца строки, то поведение компьютера при выводе текста будет непредсказуемо.

Для вывода строки commun в нужное место текста задачи включаются три команды, приведенные в примере 5.4.

Пример 5.4. Вывод строки на экран с помощью функции DOS

```
lea     dx, commun     ; помещаем в dx адрес начала строки
mov     ah, 09         ; указываем код функции DOS
int     21h            ; обращаемся к DOS
```

Функция DOS записывает только коды символов в четные байты видеопамяти, поэтому цвет символов и фона зависит от значений атрибутов, уже находящихся в нечетных байтах видеобуфера.

Текст будет обязательно виден на экране, поскольку он помещается на активную страницу, начиная с позиции, в которой находится курсор. После вывода курсор перемещается в конец текста.

Замечание

Текст направляется на стандартное устройство вывода, которым по умолчанию является дисплей. Задача может изменить установленное по умолчанию устройство, выбрав, например, принтер или дисковод. В таком случае текст будет напечатан на принтере или записан в файл.

Описанная функция лучше всего подходит для вывода на экран заранее заготовленного текста. Заготовки располагают в разделе данных программы и

оформляют с помощью директивы db, перед которой указывается метка, подобно тому, как оформлена приведенная выше строка commun. Текст заключается в одинарные или двойные кавычки, коды управляющих символов указываются явно и отделяются друг от друга и от заключенного в кавычки текста запятыми. В качестве управляющих символов могут использоваться "возврат каретки" (0Dh), "перевод строки" (0Ah), "табуляция" (09) и др. Например, для привлечения внимания оператора в выводимую строку можно включить код звукового сигнала (07), при его исполнении встроенный динамик ПК издаст гудок. Ограничения на размер выводимого текста нет, если для его размещения на экране не хватит одной строки, то продолжение будет перенесено на следующую.

Поддержка вывода BIOS. Для более гибкого управления процессом вывода текста на экран предназначены функции BIOS, входящие в группу "Video Services" (int 10h) и выполняющие следующие действия:

- 09h — вывод символа и атрибута без перемещения курсора, страница 0;
- 0Ah — вывод символа без атрибута без перемещения курсора, страница 0;
- 0Eh — вывод символа без атрибута с перемещением курсора, страница 0;
- 13h — вывод строки символов с атрибутами на указанную страницу.

Перед вызовом функции 09 и 0Ah в регистры записываются следующие величины: в ah — код функции (09 или 0Ah); в al — код выводимого символа (ASCII); bh — не используется; в cx — количество повторов символа; в bl — код атрибута, который нужен только для функции 09.

Вывод повторяющихся символов. Функции 09 и 0Ah хорошо подходят для вывода повторяющихся символов. В примере 5.5 приведен фрагмент программы, рисующий горизонтальную линию, в которой 132 раза повторяется один из символов псевдографики, имеющий код 0C4h (или 196).

Пример 5.5. Построение горизонтальной линии с помощью функции 0Ah

```
mov     ah, 0Ah         ; код запрашиваемой функции BIOS
mov     al, 0C4h        ; код ASCII символа "-"
mov     cx, 132         ; число повторений символа
int     10h             ; обращение к группе "Video Services"
```

При выполнении функции 0Ah записываются только коды символов в четные байты видеопамяти, поэтому цвет линии и фона, на котором она нарисована, зависит от значений ранее записанных атрибутов.

Если вам надо вывести символы вместе с атрибутами, то измените в примере 5.5 код функции на 09 и добавьте команду, записывающую в регистр bl

нужный код атрибута. Например, для того чтобы на голубом фоне нарисовать белую линию, атрибут должен иметь код 1Fh.

Функции 09 и 0Ah не изменяют позицию курсора, поэтому их неудобно использовать при выводе строки текста. В этом случае вам придется хранить в области данных номера строки и столбца, соответствующие текущим координатам курсора, корректировать их после вывода каждого символа, а перед выводом символа обращаться к BIOS для перемещения курсора в позицию, соответствующую этим координатам.

Вывод строки текста. Для вывода последовательности символов лучше использовать функцию 0Eh, которая после записи кода символа в четный байт видеопамяти перемещает курсор вперед на следующую позицию на экране и корректирует сохраняемые в области данных BIOS текущие координаты курсора на используемой странице (см. пример 5.3).

Перед вызовом функции 0Eh в регистры записываются следующие величины: в ah — код функции (0Eh); в al — код выводимого символа (ASCII); bh — не используется, в bl — атрибут (только для графических режимов).

Для вывода строки организуется цикл обращений к функции 0Eh. Управлять его повторами можно с помощью счетчика или повторять процесс вывода до обнаружения в строке специального признака, например символа $, или пустого байта (строка формата ASCIIZ).

В примере 5.6 цикл организован с использованием счетчика, в который перед входом в цикл помещается размер строки commun.

Пример 5.6. Вывод строки с использованием функции BIOS 0Eh

```
      lea   si, commun ; указываем адрес начала строки
      mov   cx, 48     ; задаем количество символов в строке
lp:   lodsb            ; читаем в al очередной символ строки
      mov   ah, 0Eh    ; код запрашиваемой функции
      int   10h        ; вывод очередного символа
      loop  lp         ; управление циклом
```

Раскрашивание текста. BIOS не содержит специальных функций, изменяющих только атрибуты символов, но сочетание функций 09 и 0Eh иногда позволяет раскрашивать символы и фон в нужные цвета. Рассмотрим два примера, иллюстрирующих сказанное.

При однократном обращении к функции 09 можно очистить и окрасить выбранным вами цветом все рабочее пространство экрана монитора. Для этого надо переместить курсор в начало активной страницы и заполнить ее отображаемую часть кодами символа "пробел" и атрибута, соответствующего цвету символов и фона. Как это можно сделать, показано в примере 5.7, рассчитанном на выполнение при установленном режиме VESA 109h.

Пример 5.7. "Заливка" экрана синим цветом

```
mov     ax, 920h        ; ah = код функции, al = код  символа "пробел"
mov     bx, 1Fh         ; bh = 0, bl = код атрибута
mov     cx, 132*25      ; cx = количество символов 132*25 = 3300
int     10h             ; обращение к BIOS
```

При выполнении примера 5.7 в нулевую страницу видеопамяти, начиная с позиции, соответствующей текущим координатам курсора, будет записано 3300 слов, каждое из которых содержит код `1F20h`. Если нулевая страница видеопамяти является активной, а курсор находится в ее левом верхнем углу, то все рабочее пространство экрана будет очищено от находившихся там символов и окрашено в синий цвет. После этого символы, выводимые на экран с помощью функции `0Eh`, будут окрашены в белый цвет (напомним, что код атрибута `1Fh` соответствует белым символам и синему фону). Исходные координаты курсора не изменяются.

В том случае, когда надо выделить заданным цветом одно слово или фразу, можно использовать подпрограмму, приведенную в примере 5.8. Перед обращением к ней в регистрах `ds:si` указывается адрес начала строки в оперативной памяти, а в `cx` — количество символов в строке. В регистре `bl` помещается код атрибута, `bh` не используется.

Пример 5.8. Подпрограмма для вывода символов и атрибутов

```
Colortxt: mov   ax, 920h  ; ah = код функции, al = код  символа "пробел"
          int   10h       ; закрашивание нужного пространства
Outsym:   lodsb           ; al = код очередного символа, si = si + 1
          mov   ah, 0Eh   ; код функции BIOS
          int   10h       ; вывод очередного символа
          loop  Outsym    ; управление циклом
          ret             ; возврат из подпрограммы
```

В примере 5.8 используется тот факт, что для функции `09` количество повторов указывается в регистре `cx`, а при ее выполнении позиция курсора не изменяется. При выполнении функции `09` в видеопамять записывается код символа "пробел" вместе с указанным в регистре `bl` атрибутом. Затем в цикле, имеющем метку `Outsym`, на то же место выводится текст, который будет окрашен в соответствии с атрибутом, уже записанным в видеопамять.

Позиционирование текста. Для расположения текста в нужном месте экрана можно использовать описанную в разделе 5.1.2 функцию прерывания `int 10h`, имеющую код `02`. В примере 5.9 приведен фрагмент программы, при выполнении которого курсор будет перемещен в 42-й столбец 12-й строки.

Пример 5.9. Позиционирование курсора с помощью функции BIOS 02

```
xor     bh, bh          ; номер страницы 0
mov     dh, 12          ; номер строки 12
mov     dl, 42          ; номер столбца 42
mov     ah, 2           ; код функции BIOS
int     10h             ; позиционирование курсора
```

Если после выполнения команд примера 5.9 в текущую позицию вывести строку `commun`, описанную в начале раздела, то при работе в режиме VESA `109h` она окажется расположенной в центре экрана.

Напомним, что функции, выполняющие вывод текста на экран, вычисляют адрес видеопамяти по тем значениям координат, которые хранятся в одном из слов области данных `BIOS` (см. пример 5.3). Фактическое расположение курсора на экране будет соответствовать этим координатам, только если его перемещают функции `BIOS`.

Использование функции 13h. Эта функция предназначена для вывода строки текста с явным указанием координат ее начала на экране и возможностью раскрашивания текста. В процессе вывода в видеопамять записываются коды символов и атрибутов. Атрибут может быть общим для всех символов строки или индивидуальным для каждого символа. Во втором случае выводимая строка должна содержать не только коды символов, но и их атрибуты. После вывода текста функция может переместить курсор в текущую позицию или не изменять его исходную позицию.

Для реализации перечисленных возможностей перед вызовом функции `13h` в регистре `al` указывается код режима вывода, который изменяется от 0 до 3:

- в режимах 0 и 1 код атрибута выбирается из регистра `bl`;
- в режимах 2 и 3 коды атрибутов выбираются из выводимой строки;
- в режимах 1 и 3 курсор перемещается после вывода текста;
- в режимах 0 и 2 курсор остается на исходной позиции.

Кроме указания режима, перед обращением к `BIOS` должны быть заполнены следующие регистры: `es:bp` — адрес начала строки в оперативной памяти; `cx` — количество символов в строке; `bh` — номер страницы, который используется при выводе; `dh`, `dl` — номера строки и столбца.

При выполнении примера 5.10 строка `commun`, описанная в начале раздела, выводится в центр экрана. Символы строки будет расположены на синем фоне и окрашены в белый цвет. Курсор будет перемещен в позицию, расположенную после выведенного текста.

Пример 5.10. Вывод текста с использованием функции BIOS 13h

```
push  es          ; сохранение содержимого es
push  ds          ; помещаем содержимое ds в стек
pop   es          ; и выталкиваем его в регистр es
lea   bp, commun  ; bp = адрес строки в сегменте ds
mov   cx, 48      ; указываем в cx размер строки
mov   bx, 1Fh     ; bh = 0, bl = 1Fh
mov   dh, 12      ; dh = номер исходной строки
mov   dl, 42      ; dl = номер исходного столбца
mov   ax, 1301h   ; ah = код функции, al = режим вывода
int   10h         ; BIOS выводит строку
pop   es          ; восстанавливаем содержимое es
```

Из текста примера 5.10 видно, что перед обращением к функции `13h` приходится выполнять достаточно много вспомогательных действий. Поэтому при программировании конкретной задачи вам придется выбирать, что лучше — составить собственную подпрограмму или использовать описанную функцию. Автор предпочитает работать с собственными подпрограммами.

Замечание

Функции `0Eh` и `13h` анализируют установленный видеорежим, поэтому их можно использовать для вывода текста при работе во всех графических видеорежимах, соответствующих стандартам IBM (но не стандартам VESA).

5.1.4. Непосредственная работа с видеобуфером

Если отвлечься от вспомогательных действий, то функции `09` и `0Ah` вычисляют адрес видеобуфера, используя номера страницы, строки и столбца, и записывают по этому адресу либо код символа (`0Ah`), либо код символа и атрибут (`09`). Эти действия достаточно просты и могут выполняться задачей без обращения к функциям `BIOS`. В таком случае существенно сокращается время, затрачиваемое на обмен с буфером, и появляется возможность более гибкого управления процессом вывода текста на экран. По этой причине в большинстве руководств по программированию на языке ассемблера подробно рассматриваются способы прямой работы с видеобуфером и курсором без обращения к `BIOS`.

Следует также подчеркнуть, что существует определенная категория задач, которые по тем или иным причинам не должны использовать поддержку DOS или `BIOS`. В частности, если задача работает со страницами видеопамяти, то для вывода символов нельзя использовать функции `09`, `0Ah` и `0Eh` прерывания `int 10h`.

Преимущества непосредственной работы с видеопамятью по сравнению с использованием функций BIOS заключаются в следующем:

1. При выводе текста вычисляется адрес только первого символа. Все последующие адреса на 2 больше предыдущих.
2. Существует возможность раскрашивания уже находящегося на экране текста. Для этого надо просто записать новые значения атрибутов в нечетные байты видеопамяти, не изменяя коды символов текста.
3. Возможны ввод, вывод и редактирование текста, находящегося на любой странице видеопамяти.

Вычисление адреса по координатам. Расположение текста на экране удобно задавать в виде номеров строк и столбцов. Хранить значения строки и столбца можно в словах и байтах области данных BIOS (см. пример 5.3) или в области данных задачи. Мы выберем первый вариант, поскольку в таком случае приведенные ниже примеры применимы при работе в текстовых режимах как VESA, так и IBM.

Кроме того, мы будем считать, что задача поддерживает работу со страницами видеопамяти и при преобразовании координат в адрес надо учитывать смещение страницы от начала сегмента видеопамяти.

В примере 5.11 приведена подпрограмма, вычисляющая адрес видеопамяти по текущему значению координат. Перед обращением в регистре bx указывается номер страницы, вычисленный адрес помещается в регистр di. Все нужные величины выбираются из области данных BIOS.

Пример 5.11. Вычисление адреса на указанной странице

```
GetAdr:  PushReg <ds, bx>       ; сохранение регистров
         mov   ds, NulSeg       ; очистка регистра ds
         shl   bx, 01           ; удвоение номера страницы
         mov   bx, [bx + 450h]  ; bl = столбец, bh = строка
         mov   ax, [44Ah]       ; количество символов в строке
         mul   bh               ; ax = размер строки * номер строки
         xor   bh, bh           ; очистка байта bh
         add   ax, bx           ; прибавляем к ax номер столбца
         shl   ax, 01           ; удваиваем полученный результат
         mov   di, ax           ; и сохраняем его в dx
         mov   ax, [44Ch]       ; ax = размер страницы
         pop   bx               ; восстанавливаем номер страницы
         mul   bl               ; ax = смещение страницы в буфере
         add   di, ax           ; вычисляем полный адрес
         pop   ds               ; восстановление ds
         ret                    ; возврат из подпрограммы
```

Поясним способ доступа к области данных BIOS. Она расположена в нулевом сегменте оперативной памяти. Для доступа к нулевому сегменту надо очистить один из сегментных регистров, лучше, если это регистр ds. В разделе данных задачи надо зарезервировать пустое слово с именем NulSeg и при выполнении подпрограммы копировать его в ds.

Перед вызовом подпрограммы GetAdr значения координат должны быть указаны в слове BIOS, соответствующем нужной странице. Если задача не работает со страницами, точнее работает только с нулевой страницей, то координаты курсора хранятся в слове 450h. При этом из текста примера 5.11 надо исключить вычисление адреса слова и смещения страницы от начала сегмента видеопамяти.

Запись текста в видеопамять. Мы приведем пример подпрограммы, которая записывает в видеопамять коды символов строки вместе с атрибутом, общим для всех символов, а затем покажем, как ее надо изменить для записи только кодов символов или только кодов атрибутов.

Замечание

Напомним, что регистр es должен содержать код видеосегмента, который в текстовых режимах равен B800h.

Текст подпрограммы показан в примере 5.12. Перед обращением к ней надо вычислить адрес видеопамяти и поместить его в регистр di, например, с помощью подпрограммы примера 5.11. Адрес начала выводимого текста указывается в регистрах ds:si, количество выводимых символов помещается в регистр cx, а код общего для всех символов атрибута — в регистр bl.

Пример 5.12. Запись символов строки с одинаковым атрибутом

```
OutLine: push  ax       ; сохраняем содержимое ax
         mov   ah, bl   ; помещаем атрибут в ah
wrt:     lodsb          ; читаем в al очередной символ
         stosw          ; пишем ax в видеобуфер
         loop  wrt      ; управление повторами цикла
         pop   ax       ; восстанавливаем содержимое ax
         ret            ; возврат из подпрограммы
```

При выполнении примера 5.12 указанный в bl атрибут копируется в регистр ah. Далее в цикле wrt каждый символ строки копируется в регистр al и содержимое регистра ax записывается в видеопамять.

Замечание

Напоминаем, что команды lodsb и stosw корректируют содержимое индексного регистра.

В примерах 5.13 и 5.14 показано, как изменится подпрограмма OutLine, если в видеопамять записываются только коды символов или атрибуты.

Пример 5.13. Запись символов строки без атрибутов

```
OutSym: movsb          ; копирование символа в четный байт
        inc   di       ; пропуск нечетного байта
        loop  outsym   ; управление повторами цикла
        ret            ; возврат из подпрограммы
```

Пример 5.14. Раскрашивание символов, находящихся в видеопамяти

```
OutAtr: push  ax       ; сохраняем содержимое ax
        mov   al, bl   ; помещаем атрибут в al
wrtatr: inc   di       ; пропускаем четный байт
        stosb          ; записываем код атрибута
        loop  wrtatr   ; управление повторами цикла
        pop   ax       ; восстанавливаем содержимое ax
        ret            ; возврат из подпрограммы
```

При обращении к подпрограмме OutAtr в регистре di указывается адрес видеопамяти, в регистре bl — атрибут, а в cx — сколько раз его надо записать в видеопамять (количество раскрашиваемых символов).

Перемещение курсора. Если задача использует текстовый курсор, то для его перемещения можно использовать функцию 02 прерывания int 10h, или составить свою подпрограмму. Такая подпрограмма полезна, например, в тех случаях, когда недопустимо использование поддержки BIOS.

В примере 5.15 приведена подпрограмма перемещения курсора, к которой можно обратиться по двум именам. При обращении по имени PosCur происходит обращение к описанной в примере 5.11 подпрограмме GetAdr, которая пересчитывает координаты в адрес и помещает его в регистр di. В этом случае номер страницы указывается в регистре bx, а значения координат выбираются из области данных BIOS. При обращении по имени MovCur адрес байта должен находиться в регистре di.

Замечание

Не забывайте, что курсор будет виден только в том случае, когда его рисунок расположен на активной странице.

Пример 5.15. Перемещение курсора по адресу, указанному в регистре di

```
PosCur: call  GetAdr           ; пересчет координат в адрес
MovCur: PushReg <ds,ax,bx,dx>  ; сохранение используемых регистров
        mov   ds, NulSeg       ; очистка сегментного регистра
```

```
mov   bx, di            ; bx = адрес позиции в байтах
shr   bx, 01            ; преобразование байтов в слова
mov   ah, bh            ; ah = старший байт адреса
mov   al, 0Eh           ; al = номер регистра видеоконтроллера
mov   dx, [0463h]       ; dx = базовый адрес видеоконтроллера
out   dx, ax            ; запись старшего байта в регистр 0Eh
mov   ah, bl            ; ah = младший байт адреса
inc   al                ; al = номер следующего регистра
out   dx, ax            ; запись младшего байта в регистр 0Fh
PopReg <dx,bx,ax,ds>    ; восстановление регистров
ret                     ; возврат из подпрограммы
```

В примере 5.15 основные действия выполняют команды, первая из которых имеет метку `MovCur`. Для записи данных в регистры нужен базовый адрес (порт) видеоконтроллера, который хранится в слове `463h` области данных `BIOS`. Для чтения содержимого этого слова в сегментный регистр `ds` копируется пустое слово `NulSeg`, хранящееся в разделе данных задачи.

В регистре `di` должен находиться адрес байта, он копируется в регистр `bx` и уменьшается в два раза, в результате получается адрес слова, в котором должен быть расположен рисунок курсора. Этот адрес надо записать в регистры видеоконтроллера, имеющие коды `0Eh` и `0Fh` (14 и 15).

Для записи адреса в регистры видеокарты выполняются следующие действия. В регистр `dx` записывается адрес порта видеоконтроллера из слова `463h` области данных `BIOS`. В `al` помещается код регистра (`0Eh` или `0Fh`), в который надо записать один из байтов адреса, а сам байт помещается в регистр `ah`. После этого команда `out` записывает байт в регистр видеокарты. Сначала записывается старший байт адреса в регистр `0Eh`, а затем младший в регистр `0Fh`.

После выполнения описанных действий восстанавливается сохраненное в стеке содержимое регистров и происходит возврат на вызывающий модуль.

Для перемещения курсора в конец выведенного текста после выполнения подпрограмм примеров 5.12 и 5.13 вызывается подпрограмма `MovCur`. После выполнения указанных подпрограмм регистр `di` содержит нужный адрес, и дополнительное обращение к подпрограмме `GetAdr` не требуется.

Установка активной страницы. Если по каким-то причинам вы не можете использовать функцию `05` прерывания `int 10h`, то в текст программы надо включить собственную подпрограмму аналогичного назначения.

Текст подпрограммы для смены активной страницы приведен в примере 5.16. Перед ее вызовом номер активной страницы указывается в регистре `bx`. Содержимое этой страницы появится на экране. Для вывода рисунка курсора на новую страницу используйте подпрограмму `PosCur`.

Пример 5.16. Установка активной страницы, указанной в регистре bx

```
SelPag:  PushReg <ds,ax,bx,dx>  ; сохранение используемых регистров
         mov   ds, NulSeg       ; очистка регистра ds
         mov   ax, [44Ch]       ; ax = размер страницы в байтах
         mul   bl               ; умножаем его на номер страницы
         mov   bx, ax           : сохраняем результат в bx
         mov   al, 0Ch          ; al = номер регистра видеоконтроллера
         mov   dx, [463h]       ; dx = базовый адрес видеоконтроллера
         out   dx, ax           ; запись старшего байта адреса
         mov   ah, bl           ; ah = младший байт адреса
         inc   al               ; номер второго регистра видеокарты
         out   dx, ax           ; запись младшего байта адреса
         PopReg <dx,bx,ax,ds>   ; восстановление регистров из стека
         ret                    ; возврат из подпрограммы
```

Для установки активной страницы надо старший и младший байты адреса ее начала записать в регистры 0Ch и 0Dh (12 и 13). В примере 5.16 адрес вычисляется так же, как и в примере 5.11. Размер страницы выбирается из слова 44Ch области данных BIOS, помещается в регистр ax и умножается на номер страницы, указанный в регистре bx. Полученный результат сохраняется в регистре bx, а его старший и младший байты записываются в регистры 0Ch и 0Dh. Способ записи такой же, как в примере 5.15, поэтому мы не будем повторяться.

Замечание о переносимости. При разработке видеоадаптера VGA IBM стандартизировала как состав регистров видеоконтроллера, так и способы их программирования. Стандарт распространяется на все режимы работы контроллера. Со временем более совершенные графические режимы SVGA потеснили режимы VGA, но текстовые режимы остались без изменений. Все современные видеокарты соответствуют стандартам как VESA, так и VGA IBM (одно не противоречит другому). Поэтому при работе в текстовых режимах задача может самостоятельно манипулировать с регистрами видеокарты без использования функций BIOS и оставаться при этом переносимой с одного компьютера на другой.

В случае частичного или полного отказа от использования функций BIOS вам придется самостоятельно разработать подпрограммы аналогичные тем, которые были описаны в данном разделе. При этом вы получите возможность более гибкого управления процессами ввода и вывода текста. Именно по этой причине текстовые редакторы, предназначенные для выполнения в среде DOS, используют собственные процедуры для работы с текстом.

5.2. Графические режимы

Основные особенности графических режимов, имеющие непосредственное отношение к работе с текстом, заключаются в следующем:

- видеобуфер располагается в сегменте `A000h` (а не `B800h`);
- в видеопамяти находятся коды цветов точек, а не символов;
- выключен знакогенератор, преобразующий коды символов в рисунки;
- выключена аппаратная поддержка работы с текстовым курсором.

Из этого перечня следует, что после установки графических режимов (как VESA, так и IBM) изображения символов и курсора на экране должны рисовать специальные подпрограммы. При работе с текстом в графических режимах IBM можно использовать поддержку `BIOS`, но в режимах VESA задача должна самостоятельно выполнять все необходимые действия.

5.2.1. Таблицы символов

Для вывода текста на экран нужен набор заготовок рисунков всех используемых символов. Обычно эти заготовки хранятся в специальных таблицах символов. Структура стандартных таблиц не зависит от режима, в котором они будут использоваться. При описании текстовых видеорежимов (см. раздел 5.1.1) нас не интересовала структура таблиц, поскольку изображения символов рисовал видеоконтроллер. В данном случае нам необходимо знать способ хранения данных в таблицах символов, для того чтобы составить подпрограмму, выполняющую функции знакогенератора.

Структура стандартных таблиц. В стандартных таблицах ширина символа составляет восемь точек, т. е. код одной строки (линии) рисунка символа занимает один байт. В зависимости от высоты символа (количества строк в знакоместе) заготовка полного рисунка занимает 8, 14 или 16 байтов, расположенных последовательно друг за другом. Полная таблица содержит 256 заготовок символов и занимает в оперативной памяти пространство 256×8=2048, 256×14=3584 или 256×16=4096 байтов.

Заготовка строки кодируется следующим образом. Старший бит (разряд) байта соответствует крайней левой позиции в строке, а младший бит — крайней правой позиции. Если текущая позиция содержит одну из точек рисунка символа, то соответствующий ей бит установлен (содержит 1), в противном случае он очищен (содержит 0).

На рис. 5.1 схематически изображено расположение русской буквы `Б` в стандартном прямоугольнике (знакоместе) размером 8×16 точек. Первый столбец рисунка содержит коды байтов каждой строки. В остальных восьми столбцах расположен рисунок буквы `Б`. Если клетка содержит точку изображения

символа, то в ней записана буква X, пустые клетки соответствуют фону, окружающему изображение символа.

00								
00								
FE	X	X	X	X	X	X	X	
62		X	X				X	
62		X	X				X	
60		X	X					
7C		X	X	X	X	X		
66		X	X			X	X	
66		X	X			X	X	
66		X	X			X	X	
66		X	X			X	X	
FC	X	X	X	X	X	X		
00								
00								
00								
00								

Рис. 5.1. Буква B из таблицы символов размером 8×16

Доступ к таблице символов. Для доступа к таблице символов надо знать, где она расположена. В текстовых режимах таблицы располагались в видеопамяти. В графических режимах они находятся в оперативной памяти. Адрес текущей таблицы хранится в векторе прерывания 43h, состоящем из двух слов с адресами 0000:010C и 0000:010E. В первом слове находится смещение начала таблицы в сегменте, а во втором — сам сегмент. При установке режимов VESA в вектор 43h записывается адрес англоязычной таблицы, находящейся в ROM BIOS. Нам, обычно, нужны таблицы с русскими символами, поэтому исходное содержимое вектора 43h не представляет интереса.

Как и при работе в текстовых режимах, задача может использовать собственную таблицу символов или одну из таблиц русификатора.

В первом случае место расположения таблицы в оперативной памяти выбирает программист по своему усмотрению. При работе с собственной таблицей выполнение задачи не зависит от наличия русификатора на конкретном компьютере и возможно использование символов произвольного размера и начертания. Графические видеорежимы не накладывают никаких ограничений на размеры и начертание символов — их выбирает программист, разрабатывающий конкретную прикладную задачу.

Если на компьютере установлен русификатор, например Keyrus, то можно использовать его таблицы со стандартными шрифтами трех размеров: 8×8, 8×14 и 8×16 точек. Их адреса определяются при выполнении задачи.

Установка таблицы символов. Как уже говорилось в разделе 5.1.1, в состав прерывания `int 10h` входит функция `11h`, обрабатывающая запросы, относящиеся к знакогенератору. Одним из них является следующий.

Запрос `1130h "Get Video Font Information"` возвращает сведения о таблицах шрифтов. Перед его изданием в регистре `bh` указывается код таблицы, сведения о которой надо получить. `BIOS` возвращает в регистрах следующие величины: `es:bp` — полный адрес таблицы, `cx` — размер заготовки символа в байтах, `dl` — количество строк на экране.

Код таблицы указывается в регистре `bh`, он может иметь значения от 0 до 7. Нас будут интересовать значения 2, 3 и 6, соответствующие таблицам 8×14, 8×8 и 8×16 точек. Запрос возвращает адреса англоязычных таблиц, расположенных в `ROM BIOS`. Русификатор перехватывает запрос и при обнаружении указанных кодов возвращает адрес одной из своих таблиц, находящихся в оперативной памяти.

Если используется только одна таблица, то ее адрес определяется в начале выполнения задачи, сохраняется в специально выделенном двойном слове и используется по мере необходимости. Адрес таблицы желательно хранить в разделе данных программы. Полный адрес состоит из сегмента и смещения, для их размещения нужно двойное слово. Оно описывается с помощью директивы ассемблера `dd`, перед которой располагается метка, например:

```
ftaddr   dd    00; поле для размещения адреса таблицы символов
```

Фрагмент программы, определяющий адрес таблицы символов размером 8×16 точек и сохраняющий его в `ftaddr`, приведен в примере 5.17.

Пример 5.17. Определение адреса таблицы с размером символов 8×16

```
mov     bh, 06                  ; код таблицы символов
mov     ax, 1130h               ; код запроса на получение информации
int     10h                     ; выполнение запроса
mov     word ptr ftaddr, bp     ; сохранение смещения в сегменте
mov     ftaddr+2, es            ; сохранение сегмента адреса таблицы
```

При пересылке смещения указатель типа `word ptr` нужен потому, что поле `ftaddr` является двойным словом, а регистр `bp` имеет размер слова. Благодаря явному указанию типа смещение будет записано в первое из двух слов `ftaddr`. `Ftaddr+2` является словом и при пересылке сегмента явное указание типа не требуется.

При выполнении запроса в регистр `cx` помещается высота символа (размер заготовки в байтах). Эта величина нужна для дальнейшей работы, но она

известна заранее и равна 16 байтам. Напомним, что записанный в bh код 06 означает, что мы запрашиваем адрес таблицы, содержащей символы размером 8×16 точек.

Кроме того, запрос возвращает в регистр dl количество строк на экране, но при работе в графических режимах VESA эта величина нас не интересует.

5.2.2. Программный знакогенератор

В данном разделе будет описана подпрограмма, которая по коду ASCII выбирает из таблицы соответствующую заготовку и рисует изображение символа на экране. В технической литературе для обозначения подобных подпрограмм используется термин "программный знакогенератор". Подпрограмма будет предназначена для выполнения в видеорежимах PPG. Однако, учитывая ее значимость, мы специально обсудим те изменения, которые позволят выводить текст при работе в видеорежимах direct color.

Общая характеристика знакогенератора. Структура заготовки символа (см. рис. 5.1) ничем не отличается от структуры упакованного двухцветного рисунка. Способ построения строки такого рисунка был показан в примере 3.18 раздела 3.3.1. Особенности построения небольших рисунков обсуждались в разделе 3.3.2, а соответствующая подпрограмма приведена в примере 3.21. Нам остается объединить эти примеры и учесть следующие обстоятельства.

В отличие от рисунка заготовка символа хранится не в файле, а в таблице символов и знакогенератор должен самостоятельно вычислять адрес ее начала в оперативной памяти. Для этого ему нужны следующие величины: адрес начала таблицы, ширина и высота символа (размер знакоместа для размещения символа) и код ASCII. Мы ограничимся случаями, когда ширина символов составляет 8 точек, т. е. подпрограмма рассчитана на стандартные таблицы символов. Адрес таблицы и высота символов будут находиться в специальных переменных, расположенных в разделе данных задачи.

К рисунку прилагается палитра, содержащая описание использованных в нем цветов. Таблица символов не содержит палитры. В зависимости от состояния текущего бита, знакогенератор выбирает код одного из двух заранее задаваемых цветов, они могут различаться для каждого символа.

При работе в текстовых режимах аппаратный знакогенератор выбирает коды цветов точек изображения символа (foreground) и фона (background) из байта атрибута, находящегося в видеопамяти (см. раздел 5.1.2). В нем можно закодировать 16 разных цветов foreground и 8 цветов background, коды которых соответствуют стандартной палитре CGA (см. табл. 4.2).

При работе в графических режимах для каждого выводимого символа указываются коды цветов точек его изображения и фона. В режимах PPG кодами являются номера регистров цвета видеокарты, содержащих нужные цве-

та. В режимах direct color это коды самих цветов, имеющие размер слова (Hi-Color) или двойного слова (True Color). В зависимости от используемого видеорежима в разделе данных задачи надо зарезервировать два байта, два слова или два двойных слова, содержащие цвета для раскрашивания изображения символов и окружающего их фона.

Новые переменные. Для того чтобы знакогенератор мог выбрать заготовку рисунка символа из таблицы и раскрасить ее в нужные цвета, в разделе данных задачи должны располагаться переменные, приведенные в примере 5.18.

Пример 5.18. Аргументы программного знакогенератора

```
ftaddr   dd    00    ; полный адрес таблицы символов
hsymb    dw    16    ; высота символа (размер заготовки в байтах)
augment  dw    00    ; !! константа переадресации строк рисунка символа
grndcol  db    0FFh  ; !! код цвета точек фона, окружающего символ
symbcol  db    00    ; !! код цвета точек контура символа
```

Если адрес таблицы известен при составлении программы, то его значение указывается в исходном тексте вместо нулей. Если используется одна из таблиц русификатора, то ее адрес можно определить так, как это показано в примере 5.17.

Высота символов обычно известна заранее и указывается в исходном тексте программы. Напомним, что стандартные таблицы содержат символы высотой 8, 14 или 16 строк. В тех случаях, когда планируется работа с символами разной высоты, вам придется предусмотреть запись ее значения в переменную hsymb.

Переменная augment содержит величину, которая добавляется к текущему адресу видеопамяти для перехода в начало следующей строки рисунка символа. Напомним, что в режимах PPG она выражается в точках и вычисляется как разность между шириной рабочего поля экрана и шириной рисунка. Если установлен режим с разрешением 640×480 точек, то при ширине символа в 8 точек значение augment = horsize − 8 = 640 − 8 = 632.

В примере 5.18 значения переменных grndcol и symbcol выбраны исходя из предположения, что символы изображаются черным цветом на белом фоне и что коды черного и белого цветов находятся в DAC-регистрах видеокарты с номерами 00 и 0FFh. Например, именно их используют Windows и ее приложения при работе с текстом. В общем же случае значения указанных переменных зависят от того, в каких регистрах видеокарты расположены коды нужных вам цветов.

Подпрограмма знакогенератора. Текст подпрограммы знакогенератора приведен в примере 5.19. Перед обращением к ней код ASCII выводимого на экран символа помещается в регистр al. В di записывается адрес видеопа-

мяти для размещения кода точки левого верхнего угла изображения символа. Переменная `Cur_win` содержит окно видеопамяти, которому принадлежит адрес, указанный в `di`. Предварительная установка окна не требуется. Как обычно при работе с графикой, в регистре `es` должен находиться сегмент видеобуфера (значение переменной `Vbuff`). Остальные параметры задаются неявно, это переменные примера 5.18. После выполнения подпрограммы регистр `di` содержит адрес начала следующего символа, а переменная `Cur_win` — окно, к которому относится этот адрес.

Пример 5.19. Подпрограмма рисования символов шириной в 8 точек

```
outsgn:  PushReg <ax,bx,cx,fs,si,di> ; сохранение используемых регистров
         call  Setwin        ; установка исходного окна
         push  Cur_win       ; сохранение номера исходного окна
         lfs   si, ftaddr; fs:si = адрес таблицы символов
         mov   cx, hsymb; cx = количество строк рисунка
         xor   ah, ah        ; очистка байта ah
         mul   cl            ; смещение рисунка в таблице (ax*cl)
         add   si, ax        ; полный адрес рисунка символа
         ; Построение изображения символа (внешний цикл)
out_ext: mov   bh, fs:[si]; bh = код текущей строки таблицы
         inc   si            ; адрес следующей строки таблицы
         mov   bl, 80h       ; константа выделения (и счетчик)
         ; Построение текущей строки рисунка (внутренний цикл)
out_int: mov   al, grndcol; !! al = цвет точки окружающего фона
         test  bh, bl        ; текущий бит установлен ?
         jz    @F            ; => нет, на запись кода точки
         mov   al, symbcol; !! al = цвет точки контура символа
@@:      stosb               ; !! запись кода в видеопамять
         or    di, di        ; достигнута граница сегмента ?
         jne   @F            ; => нет
         call  Nxtwin        ; установка следующего окна
@@:      shr   bl, 01        ; сдвиг константы выделения
         jne   out_int       ; управление внутренним циклом
         add   di, augment   ; адрес следующей строки рисунка
         jnc   @F            ; => адрес в пределах окна
         call  Nxtwin        ; установка следующего окна
@@:      loop  out_ext       ; управление внешним циклом
         pop   Cur_win       ; исходный номер окна
         pop   di            ; восстановление исходного адреса
         add   di, 08        ; !! адрес для следующего символа
         jnc   @F            ; => адрес в пределах окна
         mov   ax, GrUnit    ; константа для коррекции окна
         add   Cur_win, ax   ; вычисляем значение нового окна
@@:      PopReg <si,fs,cx,bx,ax> ; восстанавливаем регистры
         ret                 ; конец подпрограммы
```

Выполнение примера 5.19 начинается с сохранения в стеке содержимого используемых регистров. После этого устанавливается окно видеопамяти, в которое будет выводиться символ, и значение `Cur_win` сохраняется в стеке, т. к. оно может измениться в процессе рисования символа.

Команда `lfs` загружает младшее слово `ftaddr` в регистр `si`, а старшее — в сегментный регистр `fs`, в результате пара регистров `fs:si` содержит адрес таблицы символов в оперативной памяти. После этого в `cx` записывается количество строк в рисунке символа. Эта величина определяет количество повторов внешнего цикла, она же используется при вычислении адреса начала заготовки символа в таблице. При умножении кода символа `ASCII` на высоту рисунка (`cl`) в регистре `ax` получается смещение заготовки символа, оно прибавляется к адресу начала таблицы, в результате чего в регистре `si` получается адрес первого байта заготовки рисунка символа.

Основные действия выполняют два вложенных цикла. Внешний имеет имя `out_ext` и начинается с чтения в регистр `bl` кода очередного байта заготовки изображения символа. После чтения адрес, находящийся в регистре `si`, увеличивается на 1. В `bh` записывается константа выделения разрядов кода (`80h`) и начинается выполнение внутреннего цикла.

Внутренний цикл имеет имя `out_int`. В нем, начиная со старшего, последовательно выделяются биты строки, хранящейся в регистре `bl`. В зависимости от состояния текущего бита в `al` записывается значение переменной `grndcol` или `symbcol`, затем оно копируется в видеопамять командой `stosb`. Константа выделения смещается на разряд вправо и если она не равна нулю, то внутренний цикл повторяется.

После построения текущей строки продолжается выполнение внешнего цикла. При этом формируется адрес байта видеопамяти, соответствующий началу следующей строки, и команда `loop` повторяет выполнение внешнего цикла, пока не будут нарисованы все строки.

Как обычно, при любых изменениях адреса видеопамяти проверяется принадлежность нового значения текущему сегменту. Если оно выходит за пределы сегмента, то устанавливается следующее окно видеопамяти.

После выхода из внешнего цикла из стека восстанавливаются значение исходного окна и адрес видеопамяти, который увеличивается на ширину символа. Если при этом произойдет выход за границу сегмента, то увеличивается номер окна. Для этого к нему прибавляется значение переменной `GrUnit` (см. раздел 2.2). Перед возвратом из подпрограммы восстанавливается сохраненное в стеке исходное содержимое регистров.

Отметим, что при однократном обращении к знакогенератору не обязательно вычислять позицию следующего символа. Однако при выводе связного текста такие вычисления необходимы, и удобнее их делать именно в знакогенераторе.

Описанная подпрограмма рассчитана на выполнение в режимах PPG. Для того чтобы при описании режимов direct color не возвращаться к программированию знакогенератора, покажем, что надо изменить для его использования в этих видеорежимах. В примерах 5.18 и 5.19 комментарий к изменяемым командам отмечен двумя восклицательными знаками.

Изменения для режимов Hi-Color. В режимах Hi-Color код точки занимает слово (два байта), а код цвета содержит 15 или 16 разрядов этого слова.

В пример 5.18 вносятся следующие изменения. Значение переменной augment надо вычислять по формуле (horsize – 8)*2. Переменные grndcol и symbcol описываются директивой dw как слова (а не как байты), а их содержимое (цвет) кодируется так, как описано в главе 7.

В тексте примера 5.19 изменяемые команды будут выглядеть так:

```
out_int: mov   ax, grndcol ; !! ax = цвет точки окружающего фона
         mov   ax, symbcol ; !! ax = цвет точки контура символа
         stosw             ; !! запись кода в видеопамять
         add   di, 16      ; !! адрес для следующего символа
```

Изменения для режимов True Color. В этих режимах код точки занимает двойное слово (четыре байта), а код цвета — 24 разряда этого слова.

В примере 5.18 значение переменной augment вычисляется по формуле (horsize – 8)*4. Переменные grndcol и symbcol описываются директивой dd как двойные слова, а их содержимое (цвет) кодируется так, как описано в главе 7.

В тексте примера 5.19 изменяемые команды будут выглядеть так:

```
out_1:   mov   eax, grndcol   ; !! eax = цвет точки окружающего фона
         mov   eax, symbcol   ; !! eax = цвет точки контура символа
out_2:   stosd                ; !! запись кода в видеопамять
         add   di, 32         ; !! позиция для следующего символа
```

Подчеркнем, что в примере 5.19 заменять надо *только те команды*, комментарий для которых начинается с двух восклицательных знаков, не нарушая общей последовательности команд.

Рассмотренный вариант знакогенератора работает с символами, ширина которых составляет 8 точек. Для того чтобы подпрограмма примера 5.19 выводила символы шириной в 16 точек, код строки можно считывать в регистр bx, а константу выделения хранить в dx, ее исходное значение 8000h. Наконец, можно усложнить знакогенератор так, чтобы он выводил символы переменной ширины. Об этом следует поговорить особо.

Пропорциональные шрифты. В рукописном тексте символы имеют разную ширину. Такая форма записи текста привычна для человеческого глаза, по-

этому она применяется при оформлении печатной продукции различного назначения. Все текстовые редакторы позволяют использовать при подготовке документов пропорциональные шрифты. Свое название они получили потому, что при размещении на экране (или на бумаге) выделяется место, пропорциональное ширине символа. Например, для размещения символа ! достаточно четырех столбцов, а для буквы щ их надо намного больше.

К таблице пропорциональных шрифтов обязательно прилагается массив, содержащий ширину каждого символа в точках. При подготовке таблицы рисунки символов располагают в левой части знакоместа. Широкие символы заполняют все или почти все знакоместо, а узкие только его левую часть. Соответственно в строках таблицы заполняются старшие биты, а младшие могут не использоваться. Размер строки должен быть удобным для вычисления адреса начала заготовки символа. У большинства экранных шрифтов строка занимает одно слово.

Для вывода пропорциональных символов вам придется составить специальный знакогенератор. Его основные отличия от описанного в примере 5.19 заключаются в следующем:

- во внешнем цикле, кроме адреса начала заготовки символа, надо определять его ширину, которая хранится в отдельном массиве, прилагаемом к таблице;
- количество повторов внутреннего цикла равно ширине символа, поэтому для управления его повторами придется использовать счетчик;
- исходный код константы выделения зависит от размера строки, например если она занимает 1 слово, то код константы равен `8000h`;
- после построения рисунка при вычислении адреса позиции следующего символа, содержимое регистра `di` надо увеличивать не на восемь, а на ширину нарисованного символа.

Таким образом, при работе с текстом в графических режимах появляется возможность выводить на экран символы разной высоты и постоянной или переменной ширины, что было невозможно в текстовых режимах. В заключение несколько слов о другой, весьма распространенной категории шрифтов, используемых Windows и ее приложениями.

Масштабируемые шрифты. Область применения таблиц ограничена тем, что в них хранятся готовые точечные (растровые) рисунки символов. Изменить размер такого рисунка, а тем более повернуть его без ущерба для качества изображения, достаточно сложно. Кроме того, рисунки рассчитаны на определенное разрешение устройства вывода: чем оно выше, тем больше точек должен содержать рисунок при том же размере символа. Разрешение современных принтеров превышает 1000 точек на дюйм, поэтому размеры таблиц, используемых при печати текста, будут очень большими. Пожалуй,

именно стремление получать высококачественную печать текста стимулировало разработку масштабируемых шрифтов.

Идея заключается в том, чтобы использовать безразмерную заготовку, которую при выводе можно преобразовать в точечный рисунок конкретного размера, расположенный под заданным углом. Масштабируемые шрифты различаются по способу описания заготовки символа. В настоящее время наибольшее распространение получили шрифты форматов `PostScript` и `True Type`. `PostScript` — это язык программирования печатающих устройств. Первый интерпретатор этого языка для лазерных принтеров был разработан Adobe Systems inc. Позже появилась возможность вывода символов шрифтов `PostScript` на экран. `True Type` — это масштабируемые шрифты, стандарт на которые был разработан Microsoft для Windows и ее приложений. В настоящее время существует множество шрифтов, подготовленных в формате `True Type` и содержащих символы различного начертания (`Typeface`) и/или специальные значки. Однако при компьютерной верстке предпочтение отдается языку `PostScript`. В этом случае можно создать файл (а не распечатку) готового документа, структура которого не зависит от разрешающей способности принтера. Его можно преобразовать в нужную для размножения документа форму на специализированном типографском оборудовании.

Рассмотрение программирования вывода масштабируемых шрифтов выходит за рамки данной книги, поэтому мы заканчиваем их краткую характеристику и переходим к следующему разделу.

5.2.3. Вывод информационных строк

В процессе выполнения графических программ на экран могут выводиться информационные строки, которые делятся на две основные категории.

К первой категории относятся напоминания о назначении различных значков, находящихся на экране, не требующие конкретной реакции оператора. Например, одно из подобных сообщений, выдаваемых Windows 9X, выглядит так:

```
"Начните работу с нажатия этой кнопки".
```

Обычно такие строки через некоторое время удаляются с экрана.

К другой категории относятся подсказки, требующие от оператора выполнения конкретных действий, например ввода числовых величин, спецификаций файлов и т. п. Такие строки остаются на экране до тех пор, пока оператор не выполнит требуемое действие.

В данном разделе мы рассмотрим общую схему вывода информационных строк, а в следующих — программирование ввода текста в ответ на подсказку оператору. При изложении материала нас будет интересовать вывод отдельных строк, а не больших объемов текста, именно поэтому в заголовке

раздела использовано выражение "информационная строка". При работе с текстами большего объема применяются совершенно другие приемы.

Расположение и адрес строки. Прежде чем выводить символ, надо решить, в каком месте экрана он будет располагаться, и определить, какому участку видеопамяти соответствует это место. Способы указания координат точек и вычисления их адресов при работе в графических видеорежимах описаны в разделе 3.1.3. Здесь мы рассмотрим конкретные примеры.

Где бы ни располагалась информационная строка для работы, надо знать адрес ее начала в видеопамяти и соответствующее ему окно. При построении изображения символа знакогенератор выводит точки на экран в том порядке, в котором их коды хранятся в таблице, а именно слева направо и сверху вниз. Поэтому для вывода текста надо знать адрес верхнего левого угла первого символа информационной строки. Для его хранения в разделе данных программы надо описать две следующие переменные:

```
Inflino dw 0; для хранения адреса начала информационной строки
Inflinw dw 0; для хранения окна, к которому относится этот адрес
```

Если информационная строка расположена в начале рабочей области экрана, то надо просто очистить указанные переменные при их описании.

Если расположение строки связано с расположением курсора на экране, например, курсор указывает на значок, назначение которого надо пояснить, то в описанные переменные просто копируется текущий, или несколько измененный, адрес изображения курсора. Текущие координаты курсора и адрес его изображения на экране нужны во многих случаях, поэтому обычно они хранятся в переменных, зарезервированных в разделе данных задачи.

Подсказки о необходимости ввода данных удобно располагать в последних строках рабочей области экрана. Количество и размер строк на экране зависят от установленного видеорежима, поэтому номера и адреса последних строк рабочей области экрана вычисляются при выполнении задачи.

Если верхнюю линию изображения текста поместить в строку с номером (`Versize – hsymb`), то его нижняя линия совпадет с нижней границей рабочей области экрана.

Замечание

Напомним, что переменные `Horsize` и `Versize` содержат соответственно размер строк и их количество на экране, а переменная `hsymb` — высоту символа (см. пример 5.18), ее значение зависит от используемой таблицы.

Предположим, что левый край информационной строки расположен в нулевом столбце рабочей области экрана. В таком случае нам нужен адрес видеопамяти, соответствующий точке, расположенной в рабочей области экрана на пересечении нулевого столбца и строки с номером (`Versize – hsymb`). Его можно вычислить, например, как приведено в примере 5.20.

Пример 5.20. Вычисление адреса начала информационной строки

```
mov     ax, versize      ; ax = количество строк на экране
sub     ax, hsymb        ; уменьшаем его на высоту символа
mul     horsize          ; разность умножаем на размер строки
mov     Inflino, ax      ; сохраняем адрес в Inflino
mov     ax, dx           ; копируем содержимое dx в ax
mul     GrUnit           ; вычисляем номер окна
mov     Inflinw          ; и сохраняем его в Inflinw
```

Если выводимый текст смещен относительно левого края информационной строки, то вычисленное в примере 5.20 значение переменной `Inflino` надо увеличить на соответствующее число столбцов.

Манипуляции с исходным фоном. Информационная строка, как правило, располагается не на пустом экране. Поэтому надо сохранять изображение в той части рабочей области экрана, которую займет текст, и восстанавливать его при удалении текста с экрана. Сохраняемые коды точек изображения копируются из видеопамяти в оперативную память, так чтобы потом их можно было вернуть на прежнее место.

Подпрограммы копирования строки видеопамяти в оперативную память приведены в примерах 3.19 и 3.20, нам остается применить их для пересылки нескольких строк. Восстановление сохраненного фона ничем не отличается от построения рисунка, полностью помещающегося в оперативной памяти. Соответствующая подпрограмма описана в примере 3.21.

Для использования указанных или аналогичных подпрограмм пересылки надо знать исходные адреса видео и оперативной памяти и количество пересылаемых байтов. Начнем с последнего.

Размер и размещение фона. Высота информационной строки нам известна, точнее мы знаем, что ее значение хранится в переменной `hsymb`. Ширина строки равна произведению ширины символов на их количество, но последнее является переменной величиной. Ее значение зависит как от размера текста сообщения, так и от ответа оператора, если таковой подразумевается. Поэтому лучше выбрать ширину информационной строки равную ширине рабочей области экрана (значению переменной `Horsize`). При использовании стандартных таблиц в такой строке можно разместить 80, 100, 128 или 160 символов, в зависимости от установленного видеорежима. Чем выше разрешение, тем мельче изображение символов на экране и тем труднее читать текст, поэтому при работе с высоким разрешением вам могут понадобиться таблицы с более крупными символами.

Если информационная строка занимает всю ширину экрана, то количество сохраняемых байтов и размер буфера для их размещения вычисляются как произведение значений переменных `Horsize` и `hsymb`. Если `hsymb` = 16, то,

в зависимости от видеорежима PPG, информационная строка содержит следующее количество байтов: 101h — 7680, 103h — 9600, 105h — 12288, 107h — 16384. При работе в режимах Hi-Color указанные числа увеличатся в два раза, а в режимах True Color — в четыре раза. Очевидно, что буфер таких размеров нецелесообразно располагать в разделе данных задачи, для него надо выделить отдельное место в оперативной памяти ПК.

Способы резервирования пространства оперативной памяти описаны в приложении Б данной книги. При программировании задачи надо следить за тем, чтобы это пространство не использовалось по другому назначению. Для этого надо имя переменной, содержащей адрес выделенного сегмента, использовать только в подпрограммах сохранения и восстановления исходного фона информационной строки, но есть и другой способ защиты.

Нужный нам буфер можно расположить в начале сегмента общего назначения, выделенного в разделе 3.3.3 для временного хранения распакованной строки рисунка. Код сегмента хранится в переменной GenSeg, а начало свободного в нем пространства в переменной GenOffs. Если GenOffs присвоить значение horsize*hsymb, то соответствующая часть сегмента будет недоступна другим подпрограммам, при условии, что они используют адрес, хранящийся в GenOffs, и не уменьшают его.

Таким образом, адрес начала информационной строки в видеопамяти хранится в переменных Inflino и Inflinw, а буфер для ее размещения расположен в начале сегмента, указанного в переменных GenOffs и GenSeg.

Подпрограмма Savinfo. Текст подпрограммы, выполняющей сохранение исходного фона, приведен в примере 5.21. Перед ее вызовом надо сохранить содержимое переменной Cur_win и поместить в нее номер окна из переменной Inflinw, предварительная установка этого окна не требуется.

Пример 5.21. Сохранение фона на месте информационной строки

```
Savinfo: PushReg <Cur_win,ax,cx,si,di,fs,es>; сохранение в стеке
         call  setwin         ; установка исходного окна
         mov   fs, Vbuff      ; fs = сегмент видеобуфера
         mov   si, Inflino    ; si = адрес начала информ. строки
         mov   es, GenSeg     ; es = сегмент общего назначения
         xor   di, di         ; di = 0 — смещение в GenSeg
         mov   ax, horsize    ; ax = ширина экрана
         mul   byte ptr hsymb; умножаем ее на высоту символа
         mov   cx, ax         ; копируем результат в cx
         shr   cx, 02         ; уменьшаем его в 4 раза
Savlp:   movs  dword ptr [di], fs:[si]; копирование двойного слова
         or    si, si         ; адрес в пределах видеосегмента ?
         jnc   @F             ; -> да, переход на команду loop
         call  nxtwin         ; установка следующего окна
```

```
@@:     loop  savlp          ; управление циклом копирования
        PopReg <es,fs,di,si,cx,ax,Cur_win>; восстановление из стека
        call  setwin         ; восстановление исходного окна
        ret                  ; возврат из подпрограммы
```

В примере 5.21 копирование содержимого видеопамяти в оперативную выполняет строковая операция `movs`, у которой приемник находится в регистрах `es:di`, а источник в `fs:si`. Содержимое этих регистров формируется перед циклом пересылки. Затем вычисляется размер информационной области в байтах, и результат преобразуется в количество двойных слов. Строки рабочей области экрана копируются полностью, поэтому нужен только один цикл, в котором пересылается сразу по 4 байта. Цикл пересылки повторяет аналогичный цикл из примера 3.20 с той разницей, что копируются не байты, а двойные слова.

После пересылки восстанавливается содержимое сохраненных в стеке регистров и значение переменной `Cur_win`, восстанавливается исходное окно видеопамяти и происходит возврат на вызывающий модуль.

Подпрограмма Delinfo. Текст подпрограммы, восстанавливающей исходный фон из оперативной памяти, приведен в примере 5.22. Перед ее вызовом надо сохранить содержимое переменной `Cur_win` и поместить в нее номер окна из переменной `Inflinw`, предварительная установка этого окна не требуется.

Пример 5.22. Восстановление фона на месте информационной строки

```
Delinfo: PushReg <Cur_win,ax,cx,si,di,fs>; сохранение в стеке
        call  setwin         ; установка исходного окна
        mov   di, Inflino    ; di = адрес начала информ. строки
        mov   fs, GenSeg     ; fs = сегмент общего назначения
        xor   si, si         ; si = смещение в GenSeg (0)
        mov   ax, horsize    ; ax = ширина экрана
        mul   byte ptr hsymb; умножаем ее на высоту символа
        mov   cx, ax         ; копируем результат в cx
        shr   cx, 02         ; и уменьшаем его в 4 раза
Dellp:  movs  dword ptr [di],fs:[si]; копирование двойного слова
        or    di, di         ; адрес в пределах видеосегмента ?
        jne   @F             ; -> да, переход на команду loop
        call  nxtwin         ; установка следующего окна
@@:     loop  dellp          ; управление циклом копирования
        PopReg <fs,di,si,cx,Cur_win>; восстановление из стека
        call  setwin         ; восстановление исходного окна
        ret                  ; возврат из подпрограммы
```

В примере 5.22 данные пересылаются из оперативной в видеопамять, что и объясняет все различия текстов примеров 5.21 и 5.22. Цикл пересылки,

практически, повторяет аналогичный цикл из примера 3.15 с той разницей, что копируются не байты, а двойные слова. Подразумевается, что регистр `es` содержит код видеосегмента, указанный в переменной `Vbuff`.

Замечание

При компиляции инструкции `movs` в том виде, как она записана в примерах 5.21 и 5.22, Макроассемблер MASM 5.1 выдает предупреждающее сообщение, но генерирует правильный код. На это сообщение можно не обращать внимание.

Подпрограммы примеров 5.21 и 5.22 рассчитаны на общий случай, когда информационная строка занимает всю ширину рабочей области экрана, но может начинаться с любой строки и располагаться в двух смежных окнах видеопамяти. В частных случаях их можно упростить и ускорить пересылку.

Упрощение подпрограмм. Во всех режимах `PPG` при выбранном нами размере и расположении информационная строка полностью помещается в последнем окне видеопамяти. Поэтому в цикле пересылки проверять значение текущего адреса видеопамяти не имеет смысла. В примерах 5.21 и 5.22 циклы пересылки состоят из пяти команд и имеют метки `Savlp` и `Dellp`. Все пять команд надо исключить из текста примеров, а вместо них записать одну команду, одинаковую в обоих случаях:

```
rep movs dword ptr [di],fs:[si];цикл пересылки для примеров 5.21, 5.22.
```

Кроме этого, из списка параметров макровызовов `PushReg` и `PopReg` надо исключить имя переменной `Cur_win` и удалить команду `call setwin` перед `ret`.

Упрощенный вариант подпрограмм можно использовать, только если информационная строка полностью расположена в одном окне. Кроме рассмотренного нами случая, это условие выполняется, если строка расположена вверху рабочей области экрана (коды ее точек расположены в нулевом окне видеопамяти).

Вывод информационной строки. Мы описали сопутствующие действия и можем, наконец, рассмотреть конкретную подпрограмму для вывода строки на экран. Ее текст приведен в примере 5.23. Перед обращением адрес начала строки помещается в регистр `si`, предполагается, что строка расположена в разделе данных задачи. Признаком конца строки является пустой байт, т. е. строка подготавливается в формате `ASCIIZ`.

Пример 5.23. Вывод текста информационной строки

```
OutInf:  push  Cur_win       ; сохранение исходного значения Cur_win
         mov   ax, Inflinw   ; ax = номер окна информационной строки
         mov   Cur_win, ax   ; Cur_win = ax
         call  Savinfo       ; сохранение исходного фона
         jmp   short outstr  ; переход на выборку первого символа
```

```
out1:   call  outsgn        ; вывод на экран очередного символа
outstr: lodsb               ; al = код очередного символа (al = ds:si)
        or    al, al        ; конец выводимого текста ?
        jne   out1          ; -> нет, переход на метку out1
;       Здесь могут выполняться сопутствующие действия
        pop   Cur_win       ; восстановление исходного значения Cur_win
        call  setwin        ; восстановление исходного окна
        ret                 ; возврат из подпрограммы
```

Собственно вывод текста в примере 5.23 выполняется в цикле, состоящем из четырех команд. Первая из них имеет имя `out1`, но точкой входа является следующая команда, имеющая метку `outstr`. Код очередного символа строки считывается в регистр `al`, и если он не равен нулю, то происходит возврат на метку `out1` для обращения к подпрограмме `outsgn` (см. пример 5.19). Цикл повторяется, пока в строке не будет обнаружен пустой байт.

При желании вы можете изменить цикл так, чтобы использовался другой признак конца строки или задавалось количество символов в строке. Однако формат `ASCIIZ` является наиболее удобным.

После цикла вывода текста в примере вставлен комментарий, указывающий на возможность выполнения других действий, например выдержки паузы или ввода ответа оператора с клавиатуры. В разделе 5.2.5 мы опишем вставку в это место примера.

Если другие действия не нужны, то восстанавливается исходное значение переменной `Cur_win`, устанавливается исходное окно и происходит возврат на вызывающую программу. Выведенный текст остается на экране, а сохраненный фон — в буфере `GenSeg`.

Замечание

При планировании текстового оформления задачи имеет смысл выделить неизменяемые фрагменты текста, расположенные в различных окнах и заставках. Их можно заранее включить в рисунки окон или заставок с помощью графического редактора. При этом текст становится частью соответствующего рисунка, выводится, перемещается или удаляется вместе с ним. В некоторых случаях это удобно, а современные графические редакторы позволяют включать в рисунки текст, состоящий из символов различных размеров, начертаний и цветов.

5.2.4. Текстовый курсор в графическом режиме

При вводе и редактировании текста, для указания позиции вводимого или изменяемого символа, традиционно используется курсор. В графических режимах аппаратная поддержка текстового курсора выключена, поэтому применяется "программный" курсор. В данном разделе мы рассмотрим способ построения мигающего текстового курсора, а его использование при вводе символов с клавиатуры будет описано в следующем разделе.

Предварительные замечания. При работе в графических режимах на экране могут находиться рисунки двух курсоров, один из которых указывает текущее положение манипулятора "мышь", а второй — место вводимого или изменяемого символа. Главную роль играет "указатель мыши", он нужен для управления процессом выполнения задачи и, в частности, для изменения позиции текстового курсора. Указатель мыши может перемещаться по всей рабочей области экрана. В отличие от него текстовый курсор появляется только в определенных местах, например в диалоговых окнах.

Для работы с текстом в графических режимах на экране выделяются специальные строки или окна, размеры которых зависят от их назначения. Как правило, они не велики и предназначены для ввода различных установочных данных — числовых величин, зарезервированных (ключевых) слов, спецификаций файлов и т. п. Только у специализированных редакторов текстовые окна занимают большую часть экрана или весь экран.

Windows и ее приложения работают с текстом в черно-белом режиме — на белом фоне изображаются черные буквы. Текстовый курсор обычно имеет форму мигающей вертикальной черты, цвет которой совпадает с цветом букв. Высота черты зависит от высоты шрифта, а ширина составляет одну или две точки. Рассмотрим, как программируется подобный рисунок текстового курсора (далее по тексту просто "курсора").

Способ построения курсора. В процессе редактирования текста изображение курсора может перемещаться по строкам и располагаться на месте уже выведенных символов. Поэтому надо принять специальные меры, для того чтобы при перемещении курсор не искажал изображение символов текста, например сохранять рисунок расположенного под курсором символа и восстанавливать его после перемещения курсора. При работе с текстом обычно используют более простой способ, при котором для записи кодов точек рисунка курсора в видеопамять используется команда `xor`.

Двухадресная команда `xor` вычисляет логическую функцию `exclusive OR` (исключающее ИЛИ), ее операндами являются источник и приемник, результат помещается в приемник. При выполнении инструкции `xor` микропроцессор запрещает перенос единиц переполнения из младших разрядов в старшие и производит поразрядное сложение операндов. Результат выполнения операции для одного бита показан в табл. 5.1.

Таблица 5.1. *Схема выполнения операции* `xor`

Состояние бита источника	0	0	1	1
Состояние бита приемника	0	1	0	1
Состояние бита результата	0	1	1	0

Обратите внимание на последний столбец таблицы. Если состояние *всех* разрядов у приемника и источника одинаково, то в результате получится

нуль, т. е. приемник будет очищен. Это свойство команды `xor` мы неоднократно использовали в примерах для очистки регистров. Здесь нас интересует ее другое свойство.

Если у источника установлены *все* разряды, то установленные разряды приемника будут очищены, а очищенные — установлены, т. е. код приемника будет инвертирован. При повторной инверсии произойдет восстановление исходного кода приемника. Таким образом, `xor` позволяет инвертировать код приемника, а затем вернуть его первоначальное значение.

Использовать это свойство для построения текстового курсора можно только при определенных ограничениях. Напомним, что в режимах `PPG` код точки является номером регистра цвета видеокарты. Поэтому, инвертируя код точки, мы изменяем номер регистра видеокарты, а получаемый при этом цвет зависит от того, что записано в этом регистре, т. е. от установленной палитры цветов. При описании системной палитры в разделе 4.5 мы рекомендовали размещать код черного цвета в регистре 0, а белого — в регистре `0FFh`. В таком случае при инверсии кода точки будет инвертирован и ее цвет.

Точки изображения символа имеют черный цвет, если рисунок курсора затрагивает эти точки, то они станут белыми. Для уменьшения наложения рисунку курсора придают форму узкой вертикальной черты, расположенной в начале или в конце знакоместа. Например, у редактора Microsoft Word ширина курсора составляет 2 точки. Первая из них расположена в конце одного знакоместа, а вторая — в начале следующего, в которое будет помещен введенный с клавиатуры символ. При таком расположении курсор не закрывает основной рисунок символа.

Подпрограмма TglCrsr. В примере 5.24 приведен текст подпрограммы, изменяющей состояние курсора на противоположное. При первом обращении изображение курсора появляется на экране, а при втором — удаляется с экрана. Оно имеет форму вертикальной линии, расположенной в двух левых столбцах знакоместа. Высота линии равна высоте символа, а ширина составляет две точки.

Подпрограмма рассчитана на то, что для вывода символов на экран используется знакогенератор из примера 5.19. Поэтому адрес видеопамяти, соответствующий верхнему левому углу рисунка курсора, выбирается из регистра `di`, а код цвета фона и высоту символов задают переменные `grndcol` и `hsymb` (см. пример 5.18). При коррекции адреса видеопамяти используется переменная `bperline`, описанная в примере 2.11, она равна значению `Horsize`, умноженному на размер кода точки в байтах (1—4).

Пример 5.24. Подпрограмма изменения состояния текстового курсора

```
TglCrsr: PushReg <ax,cx,di,Cur_win>; сохранение в стеке
         call  Setwin        ; установка исходного окна
```

```
        mov   al, grndcol    ; !! al = код цвета фона
        mov   cx, hsymb      ; cx = высота символов
Tcrsr:  xor   es:[di], al    ; !! изменяем код первой точки
        xor   es:[di+1], al  ; !! изменяем код второй точки
        add   di, bperline   ; коррекция видеоадреса
        jnc   @F             ; => адрес в пределах сегмента
        call  Nxtwin         ; установка следующего окна
@@:     loop  Tcrsr          ; управление циклом
        PopReg <Cur_win,di,cx,ax>  ; восстановление из стека
        ret                  ; завершение работы подпрограммы
```

Выполнение подпрограммы примера 5.24 начинается с сохранения в стеке тех величин, значения которых могут измениться, и установки исходного окна. После этого в регистр `al` копируется код цвета фона, а в `cx` — количество строк в символе.

Цикл изменения состояния курсора имеет метку `Tcrsr`. Его первые две команды изменяют состояние двух первых точек очередной строки рисунка. Затем вычисляется адрес следующей строки, если при этом вырабатывается признак переполнения, то производится смена окна видеопамяти. Команда `loop` повторяет выполнение цикла, пока не будут изменены все строки. После этого восстанавливаются сохраненные в стеке величины и происходит возврат на вызывающий модуль.

Исходный текст примера 5.24 рассчитан на выполнение в видеорежимах `PPG`. Комментарий к командам, зависящим от видеорежима, начинается с двух восклицательных знаков. При работе в режимах `direct color` используйте варианты переменных команд, приведенные в табл. 5.2.

Таблица 5.2. *Варианты переменных команд для примера 5.24*

Режимы PPG	Режимы Hi-Color	Режимы True Color
mov al, grndcol	mov ax, grndcol	mov eax, grndcol
xor es:[di], al	xor es:[di], ax	xor es:[di], eax
xor es:[di+1], al	xor es:[di+2], ax	xor es:[di+4], eax

Мигающий курсор. Текстовый курсор обычно мигает, т. е. его изображение периодически появляется и исчезает. Для получения эффекта мигания надо вызывать подпрограмму `TglCrsr` через равные промежутки времени, например через 0,5 сек, как это делают Windows и ее приложения.

При управлении курсором высокая точность измерения времени не требуется, поэтому можно использовать таймер, который "тикает" через каждые 55 миллисекунд, или 18,2 раза в секунду. Для выдержки паузы надо дождаться пока от таймера поступит нужное количество тиков с момента начала паузы. Вопрос лишь в том, как узнать, что таймер "тикнул".

В области данных BIOS, начиная с адреса 0000:046C, зарезервировано 4 байта, содержащих 32-разрядный счетчик количества тиков. BIOS очищает счетчик при первоначальной загрузке, после чего его значение увеличивается на 1 с каждым тиком таймера. Для выдержки паузы надо запомнить исходное значение счетчика, в начале паузы, а затем время от времени сравнивать текущее значение с исходным. Пауза закончится, когда их разность достигнет нужного значения.

Необходимость периодически опрашивать состояние счетчика тиков является недостатком такого способа, поэтому он применяется только в тех случаях, когда задача ничего не делает во время паузы. В общем случае работа с таймером происходит в режиме прерываний. Для этого вам надо составить подпрограмму, которая будет выполняться при каждом тике таймера. О том, как ее составить, мы поговорим особо, а сначала рассмотрим, как сделать, чтобы она выполнялась при каждом тике таймера.

Перехват прерываний от таймера. Каждый тик таймера вызывает так называемое аппаратное прерывание. Текущий процесс вычислений приостанавливается и выполняется специальная процедура BIOS. Она обслуживает процессы, синхронизированные с таймером, и вызывает подпрограмму, адрес начала которой указан в векторе 1Ch. Этот вектор специально выделен для нужд прикладных задач. Сразу после загрузки ПК в нем находится адрес команды iret, расположенной в ROM BIOS. Если в прикладной задаче есть подпрограмма, выполнение которой должно быть синхронизировано с таймером, то ее адрес указывается в векторе 1Ch.

Исходное значение вектора 1Ch надо сохранить в теле задачи. Оно нужно для того, чтобы после вашей подпрограммы можно было начать выполнение той программы (внешней по отношению к задаче), адрес которой находился в векторе 1Ch до изменения его содержимого. Такой трюк в литературе называется "перехват вектора прерывания", он широко используется при работе прикладных задач с внешними устройствами.

Для изменения содержимого вектора надо знать его адрес. Векторы пронумерованы начиная с нуля, каждый из них занимает 4 байта оперативной памяти, а нулевой вектор расположен по адресу 0000:0000. Следовательно, умножив номер вектора на 4, мы получим его адрес в оперативной памяти. В нашем случае 4*1Ch = 70h ($4 \cdot 28 = 7 \cdot 16 = 112$).

В примере 5.25 приведена группа команд, выполняющих перехват вектора 1Ch, неизвестные вам имена переменных описаны ниже в примере 5.27.

Пример 5.25. Сохранение и изменение содержимого вектора 1Ch

```
xor     ax, ax              ; очистка регистра ax
mov     CurStat, al         ; запрет построения рисунка курсора
mov     fs, ax              ; очистка сегментного регистра fs
```

```
lea     ax, cs:Timeint    ; ax=адрес прерывающей подпрограммы
mov     bx, cs            ; bx=сегмент прерывающей подпрограммы
cli                       ; запрещаем прерывания
xchg    fs:[70h], ax      ; перестановка содержимого ax и 70h
xchg    fs:[72h], bx      ; перестановка содержимого bx и 72h
mov     cs:Vec1C, ax      ; Vec1C=исходное значение слова 70h
mov     cs:Vec1C+2, bx    ; Vec1C+2=исходное значение слова 72h
sti                       ; разрешаем прерывания
```

Перехват вектора `1Ch` производится в начале выполнения задачи, но после того, как подготовлено все необходимое для корректной работы прерывающей подпрограммы. В нашем случае имя прерывающей подпрограммы `Timeint`, а для ее корректной работы надо запретить построение рисунка текстового курсора. Для этого вторая команда примера очищает переменную `CurStat`.

Для доступа к словам вектора прерывания очищается один из сегментных регистров, в примере 5.25 это регистр `fs`, его очищает третья команда. Затем в регистры `ax` и `bx` записываются адрес прерывающей подпрограммы и сегмент, в котором она находится. Прежде чем изменять содержимое вектора, надо запретить прерывания. Это делается потому, что выполнение задачи никак не синхронизировано с таймером и прерывание от последнего может произойти в тот момент, когда задача начала, но еще не завершила изменение и запоминание содержимого вектора `1Ch`.

Прерывание запрещает команда `cli`, после нее производится обмен содержимого (`xchg`) слов вектора и регистров `ax` и `bx`. В результате в словах вектора `1Ch` окажется новый, а в регистрах `ax` и `bx` старый адрес, который надо запомнить. Следующие две команды пересылают старый адрес в слова `Vec1C` и `Vec1C+2`, после чего команда `sti` разрешает прерывания.

Восстановление вектора прерывания. Перед завершением задачи восстанавливается исходное значение вектора `1Ch`, т. е. подпрограмма `Timeint` исключается из списка заданий таймеру. Если это не сделать, то при первом же тике таймера произойдет обращение к области памяти, в которой уже нет прерывающей подпрограммы, что приведет к аварийной ситуации.

В примере 5.26 приведена группа команд, выполняющих восстановление исходного значения вектора `1Ch`. Эти команды могут быть выполнены непосредственно перед завершением задачи, т. е. перед возвратом в DOS.

Пример 5.26. Восстановление исходного содержимого вектора 1Ch

```
xor     ax, ax            ; очистка регистра ax
mov     fs, ax            ; очистка сегментного регистра fs
mov     ax, cs:Vec1C      ; ax = содержимое Vec1C
```

```
mov       bx, cs:Vec1C+2   ; bx = содержимое Vec1C+2
cli                        ; запрещаем прерывания
mov       fs:[70h],ax      ; восстановление 1-го слова вектора
mov       fs:[72h],bx      ; восстановление 2-го слова вектора
sti                        ; разрешаем прерывания
```

В примере 5.26 для доступа к словам вектора используется регистр `fs`, поэтому его содержимое предварительно очищается. Затем в регистры `ax` и `bx` копируются первое и второе слово сохраненного ранее вектора `1Ch`. После запрещения прерываний содержимое регистров `ax` и `bx` копируется в слова `70h` и `72h`, и разрешаются прерывания. Вектор восстановлен, и можно завершать выполнение задачи.

Замечание

Для работы с векторами прерываний предназначены две специальные функции DOS (прерывания `int 21h`). Функция `Get Vector` (код `35h`) читает содержимое вектора, а `Set Vector` (код `25h`) записывает в вектор новое содержимое. Однако их применение просто не оправдано, в чем вы можете убедиться самостоятельно.

Прерывающая подпрограмма подсчитывает количество тиков таймера и, как только оно будет равно 9 (примерно через каждые 0,5 сек), изменяет текущее состояние курсора. Рисунок курсора находится на экране, только если задача работает с текстом. Поэтому необходим специальный признак, разрешающий или запрещающий изменение состояния курсора. Кроме того, нужен признак, позволяющий узнать, в каком состоянии находится курсор — включен или погашен (виден или не виден на экране).

Для хранения счетчика тиков и признаков в разделе данных задачи надо зарезервировать две однобайтовые переменные, имеющие следующие имена:

```
Ntick   db    9 ; счетчик тиков таймера изменяется от 0 до 9
CurStat db    0 ; текущее состояние курсора изменяется от 0 до 3
```

В байте `CurStat` используются только два младших бита. Нулевой бит разрешает (1) или запрещает (0) изменение состояния (мигание) курсора, он устанавливается в основной программе. Первый бит отражает текущее состояние курсора на экране: 0 — выключен, 1 — включен. Им управляет прерывающая подпрограмма, текст которой приведен в примере 5.27.

Пример 5.27. Подпрограмма, создающая эффект мигающего курсора

```
Timeint:  PushReg <ax,ds,es>; !! сохранение регистров ax, ds и es
          mov   ax, data    ; !! ax = значение сегмента данных
          mov   ds, ax      ; !! ds = код сегмента данных
          mov   es, VBuff   ; !! es = код сегмента видеобуфера
```

```
          test  CurStat, 01  ; работа с курсором разрешена ?
          jz    GoV1C        ; -> нет, завершение подпрограммы
          dec   Ntick        ; Ntick = Ntick – 1
          jnz   GoV1C        ; -> пауза продолжается
          mov   Ntick, 09    ; Ntick = 9 (примерно 0,5 сек)
          xor   CurStat, 02  : изменение признака состояния курсора
          call  TglCrsr      : изменение состояния рисунка курсора
GoV1C:    PopReg <es,ds,ax> ; !! восстановление es, ds и ax
          db    0EAh         ; код инструкции jmp, на удаленный адрес
Vec1C     dw    00, 00       ; старое содержимое вектора 1Ch
```

Прежде всего отметим, что подпрограмму примера 5.27 *нельзя* вызывать командой `call Timeint`, поскольку ее выполнение завершается не командой `ret`, а безусловным переходом на тот адрес, который раньше находился в векторе `1Ch`. Подпрограмму вызывает процедура `BIOS`, обрабатывающая задания, адресованные таймеру. Действия, которые надо выполнить для разрешения или запрещения вызовов данной подпрограммы при каждом тике таймера, показаны в примерах 5.25 и 5.26.

При вызове прерывающей подпрограммы в стеке сохраняется содержимое регистров `ax`, `ds` и `es`. В общем случае при входе содержимое регистра `ds` не определено, и в него надо записать код сегмента данных. Имя сегменту данных присваивается при его описании (см. пример 2.11), обычно это `data`. Если вы выбрали другое имя, то измените команду `mov ax, data`. Регистр `es` используется в подпрограмме `TglCrsr`, изменяющей текущее состояние рисунка курсора, поэтому он должен содержать код сегмента видеобуфера, который хранится в переменной `Vbuff`.

Основные действия, выполняемые в примере 5.27, достаточно просты. Проверяется состояние младшего разряда байта `CurStat`, если он очищен, то команда `jz GoV1C` завершает выполнение подпрограммы, в противном случае работа с рисунком курсора разрешена. Содержимое счетчика тиков уменьшается на 1 и если разность больше нуля, то пауза не закончена и происходит выход из подпрограммы без изменения рисунка курсора. Наконец, если разность равна нулю, то в счетчик тиков записывается число 9, и инвертируются рисунок курсора и признак его состояния.

Перед выходом из подпрограммы восстанавливается сохраненное в стеке содержимое регистров `es`, `ds` и `ax`, а затем выполняется команда `jmp`, которая передает управление на адрес, сохраненный в двух словах переменной `Vec1C` (при выполнении примера 5.25).

В тексте примера 5.27 использован следующий трюк. Имя команды `jmp` не записано явно, вместо этого в байте, расположенном перед переменной `Vec1C`, указан код операции (`0EAh`). При ее выполнении микропроцессор интерпретирует содержимое следующих двух слов как адрес, на который про-

изводится переход. Это наиболее простой, но не единственно возможный способ вернуться на сохраненное значение вектора прерывания.

Замечание

Если при выполнении вашей задачи содержимое сегментных регистров `ds` и `es` не изменяется после их первоначальной установки, то из текста примера 5.27 можно исключить 5 команд, комментарий к которым начинается с двух восклицательных знаков.

5.2.5. Ввод символов с клавиатуры

Для иллюстрации способов работы с описанными подпрограммами мы рассмотрим ввод текста в специально выделенный буфер строки. Такой буфер нужен для того, чтобы оператор мог исправить допущенные им ошибки до того, как задача начнет обрабатывать введенную строку. Обычно содержимое буфера становится доступным для дальнейшей обработки после того, как оператор нажмет клавишу <Enter> ("возврат каретки").

Работа клавиатуры никак не связана с текущим видеорежимом, но от него зависит способ отображения вводимых символов на экране (эхо-печать).

Чтение введенного символа. При нажатии и отпускании любой клавиши контроллер клавиатуры генерирует аппаратное прерывание, при этом прекращается выполнение текущего вычислительного процесса и вызывается специальный драйвер, расположенный в `ROM BIOS`. Он считывает код введенного символа (`scan code`), преобразует его в код `ASCII` и помещает оба кода в специальный буфер, расположенный в области данных `BIOS`, начиная с адреса `0000:041E`. Последующая обработка введенного символа, включая вывод его изображения на экран, находится вне компетенции драйвера.

`Scan code` — это просто порядковый номер нажатой или отпущенной клавиши, для его преобразования в код `ASCII` надо учитывать состояние одной или нескольких функциональных клавишей. Например, `scan code 1Eh` соответствует сразу четырем буквам — латинским "A", "a" и русским "Ф", "ф", в то время как `ASCII` коды этих букв равны, соответственно, `41h`, `61h`, `94h` и `0E4h`.

`BIOS` работает только с латинским алфавитом, для ввода с русских букв нужны специальные программы русификаторы, например Keyrus, или драйверы, входящие в состав русифицированных версий Windows. Они перехватывают аппаратное прерывание от клавиатуры и формируют коды русских букв. Причем коды русских букв зависят от используемой кодовой таблицы, а их не так уж мало.

Задача может считывать введенные с клавиатуры символы как в режиме прерываний, так и путем опроса состояния буфера, расположенного в области данных `BIOS`. В первом случае задача должна перехватывать вектор

прерывания 09, подобно тому, как это делалось для вектора 1Ch. Нас интересует режим опроса. В этом режиме задача может самостоятельно работать с буфером клавиатуры, или вызывать специальную функцию BIOS.

Функции, обслуживающие клавиатуру, сгруппированы в прерывание int 16h (Keyboard Services). Нам нужна функция с кодом 0 (ah=0), которая опрашивает буфер клавиатуры до тех пор, пока в него не будет записан код введенного символа. После этого происходит возврат в задачу, scan code находится в регистре ah, а ASCII код — в регистре al. Заметим, что при вводе русских букв scan code может отсутствовать (ah=0), это зависит от установленного русификатора.

Управление курсором. Изменение состояния курсора можно разрешить перед началом ввода строки и запретить в конце ввода. Однако в таком случае при редактировании вводимого текста придется следить за текущим состоянием курсора, неоднократно удалять его с одного места и выводить в другом, т. е. выполнять много вспомогательных действий.

Мы выберем другой способ, при котором курсор виден (и мигает) только во время ожидания ввода символа с клавиатуры.

Для разрешения работы с курсором в байт Ntick записывается число 9, что соответствует паузе примерно в 0,5 сек, а в байт CurStat — число 3. Напомним, что 1 в байте CurStat разрешает прерывающей подпрограмме выполнять отсчет времени и изменять состояние курсора, а 2 указывает на то, что курсор нарисован. После этого надо вызвать подпрограмму TglCrsr, которая нарисует курсор. Теперь при каждом тике таймера состояние курсора будет изменяться на противоположное.

После ввода символа очищается байт CurStat, а рисунок курсора удаляется с экрана, если он там находился.

Таким образом, мы "привязали" рисунок курсора к тому знакоместу, в которое помещается вводимый с клавиатуры символ и никаких других действий для управления курсором не требуется.

Обработка служебных символов. При вводе с клавиатуры BIOS не отображает символы на экране, это должна делать подпрограмма, используемая для записи кодов символов в буфер строки. Перед выводом изображения символа на экран надо убедиться в том, введен ли код ASCII. Он генерируется только при нажатии на определенные клавиши (их примерно половина), а в остальных случаях если код и существует, то не имеет практического смысла.

Вопрос о существовании кода ASCII решается на основании анализа scan code, если его значение меньше чем 36h, то код ASCII существует, в противном случае нажата одна из служебных клавиш. Если значение кода ASCII больше или равно 20h (код символа "пробел"), то изображение символа можно выводить на экран, в противном случае прочитан один из управ-

ляющих символов. По традиции управляющими называют те символы, у которых код ASCII имеет значения от 0 до 1Fh.

В тех случаях, когда код ASCII меньше, чем 20h или scan code больше чем 35h, нужен дополнительный анализ введенного кода для выбора вспомогательных действий. При редактировании строки текста достаточно использовать следующие коды: <левая стрелка> (4Bh), <правая стрелка> (4Dh), <Delete> (53h), <возврат на шаг> (0Eh), <Enter> (1Ch); в скобках указано значение scan code. Дополнительно могут использоваться <стрелка вверх> (48h), <стрелка вниз> (50h), <табуляция> (0Fh) и другие коды.

Подпрограмма Inline. Текст подпрограммы, выполняющей ввод символов с клавиатуры и простые функции редактирования, приведен в примере 5.28. В разделе данных задачи надо выделить буфер, имеющий метку Linbuf, его размер должен быть достаточен для размещения вводимого текста (не более 80 байтов). В конце текста подпрограмма записывает пустой байт.

Пример 5.28. Ввод символов текста в буфер строки (Linbuf)

```
Inline: lea   si, Linbuf     ; si = адрес буфера строки
;       Ввод очередного символа и управление курсором
Gs:     cli                  ; запрещаем прерывания
        mov   CurStat, 03    ; CurStat = 3
        mov   Ntick, 09      ; Ntick = 9 (0,5 сек)
        sti                  ; разрешаем прерывание
        call  TglCrsr        ; рисуем изображение курсора
        mov   ah, 00         ; код запроса ввода символа
        int   16h            ; ожидание ввода символа
        cli                  ; запрещаем прерывания
        test  CurStat, 02    ; курсор нарисован ?
        mov   CurStat, 00    ; CurStat = 0
        sti                  ; разрешаем прерывания
        je    Prevanl        ; -> нет, курсор погашен
        call  TglCrsr        ; гашение курсора
;       Предварительный анализ введенного кода
Prevanl: cmp  ah, 35h        ; код ASCII существует ?
        ja    Detail         ; -> нет, это служебный символ
        cmp   al, 20h        ; это управляющий символ ?
        jb    Detail         ; -> да
        mov   ds:[si], al    ; запись ASCII кода в буфер строки
        inc   si             ; коррекция адреса
        call  outsgn         ; вывод символа на экран
        jmp   SHORT Gs       ; переход на продолжение ввода
;       Детальный анализ служебных кодов
Detail: cmp   ah, 1Ch        ; это символ "Enter" ?
        jne   Cont_1         ; -> нет, продолжение анализа
```

```
        mov   byte ptr ds:[di], 00; запись признака конца текста
        ret                 ; возврат из подпрограммы
Cont_1: cmp   ah, 0Eh       ; это символ "возврат на шаг" ?
        jne   Cont_2        ; -> нет, продолжение анализа
        call  prevpos       ; адрес предыдущего знакоместа
        mov   al, ' '       ; al = код символа "пробел"
        call  outsgn        ; стираем предыдущий символ
        dec   si            ; и удаляем его из буфера строки
        call  prevpos       ; адрес предыдущего знакоместа
        jmp   short Gs      ; переход на продолжение ввода
Cont_2: jmp   short Gs      ; здесь можно продолжить анализ
;       Вычисление позиции предыдущего символа
Prevpos:sub   di, 08        ; уменьшение адреса видеопамяти
        jnc   @F            ; -> смена окна не нужна
        mov   ax, GrUnit    ; ax = единица приращения памяти
        sub   Cur_win, ax   ; Cur_win = Cur_win — GrUnit
@@:     ret                 ; возврат из подпрограммы
```

В текст примера 5.28 вставлены строки комментария, поясняющего назначение основных групп команд. В каждой из этих групп, кроме подпрограммы `Prevpos`, выполняются действия, смысл которых описан выше, поэтому здесь мы уточним некоторые особенности реализации.

Прерывания от таймера надо запрещать на время работы с переменными `Ntick` и `CurStat`. Напомним, что команда `cli` запрещает, а `sti` разрешает маскируемые прерывания. В примере 5.28 они используются дважды при разрешении и запрещении изменения состояния рисунка курсора. Во втором случае между `test CurStat, 02` и `je Prevanl` расположены две команды, выполнение которых не изменяет состояние регистра флагов (признаков). Это просто трюк, упрощающий проверку состояния и очистку байта `CurStat`.

Коды введенного с клавиатуры символа находятся в регистрах `ah` (`scan`) и `al` (`ASCII`). Если код `ASCII` существует и его значение больше, чем `1Fh`, то изображение символа выводится на экран и продолжается ввод.

Если обнаружен код служебного символа, то производится его детальный анализ. В примере 5.28 обрабатываются коды только двух символов — <Enter> и <возврат на шаг>. В других случаях подпрограмма просто игнорирует введенный код и ждет ввода с клавиатуры очередного символа. Если вы захотите дополнить подпрограмму, то вместо команды `jmp short Gs`, имеющей метку `Cont_2`, вставьте команды, выполняющие анализ и обработку нужных вам кодов.

При обнаружении кода символа <Enter> в буфер строки записывается пустой байт и происходит возврат из подпрограммы.

Символ <возврат на шаг> обрабатывается так. Устанавливается адрес предыдущего знакоместа, в него выводится символ <пробел>, последний введенный символ удаляется из буфера строки и повторно устанавливается адрес предыдущего знакоместа. Установку адреса предыдущего знакоместа выполняет подпрограмма `Prevpos`, которая понадобится, если вы добавите обработку клавиши <левая стрелка> для перемещения влево по строке.

Пример вызова Inline. Для иллюстрации использования описанной подпрограммы мы перепишем текст примера 5.23, заменив в нем строку комментария двумя командами. Результат показан в примере 5.29.

Пример 5.29. Вывод текста информационной строки

```
OutInf:  push  Cur_win       ; сохранение исходного значения Cur_win
         mov   ax, Inflinw   ; ax = номер окна информационной строки
         mov   Cur_win, ax   ; Cur_win = ax
         call  Savinfo       ; сохранение исходного фона
         jmp   short outstr  ; переход на выборку первого символа
out1:    call  outsgn        ; вывод на экран очередного символа
outstr:  lodsb               ; al = код очередного символа (al = ds:si)
         or    al, al        ; конец выводимого текста ?
         jne   out1          ; -> нет, переход на метку out1
         call  Inline        ; ввод строки теста с клавиатуры
         call  Delinfo       ; удаление информационной строки с экрана
         pop   Cur_win       ; восстановление исходного значения Cur_win
         call  setwin        ; восстановление исходного окна
         ret                 ; возврат из подпрограммы
```

При выполнении примера 5.29 исходный фон будет сохранен на месте информационной строки, на экран будет выведен текст подсказки оператору, введен его ответ с эхо-печатью и записью введенных кодов в массив `Linbuf` и, после нажатия оператором на клавишу <Enter>, восстановлен исходный фон на месте информационной строки. Перечисленные действия выполняются с помощью подпрограмм, описанных в данном разделе. Использование данного примера для ввода спецификации файла описано в приложении А данной книги.

В этой главе автор стремился ответить на основные вопросы, которые приходится решать при программировании вывода текста на экран в графических режимах VESA. Насколько ему это удалось — судить читателю, а мы переходим к рассмотрению следующей, не менее важной темы, связанной с управлением процессом вычислений, выполняемых в задачах.

ГЛАВА 6

Курсор и мышь

Манипулятор "мышь" (далее просто мышь) является основным инструментом для поддержки диалога пользователя с задачей при работе в графических видеорежимах. С помощью мыши выбираются и активизируются диалоговые окна, меню или значки на панелях инструментов, выполняются различные манипуляции с рисунками и прочие действия.

На экране монитора текущее расположение мыши указывает специальный рисунок, который принято называть графическим курсором (`graphics cursor`) или указателем мыши (`mouse pointer`). Он удаляется с одного места и появляется на другом при каждом перемещении мыши. Текущие координаты курсора нужны задаче для выполнения различных действий.

В данной главе мы рассмотрим наиболее распространенные варианты построения рисунка курсора и обсудим способы организации взаимодействия задачи с манипулятором "мышь".

6.1. Построение рисунка курсора

При работе в текстовых или графических режимах IBM драйвер мыши самостоятельно определяет установленный видеорежим и в зависимости от этого выбирает способ построения или удаления рисунка курсора, задача только разрешает или запрещает ему выполнять эти действия. Драйверы мыши предназначены для работы в среде DOS, они различают только стандартные режимы IBM. Поэтому после установки режимов VESA строить и перемещать рисунок курсора должна задача. В отличие от DOS, операционные системы семейства Windows и OS/2 поддерживают управление курсором, что упрощает действия прикладных задач.

Замечание

Напомним, что код текущего режима хранится в байте, расположенном в области данных `BIOS`, по адресу `0000:0449`. Трехзначные коды режимов VESA не

помещаются в байте, и их заменяют кодами OEM, которые уникальны для каждой модели видеокарты. Именно отсутствие стандартов на коды OEM не позволяет разрабатывать драйверы, выполняющие построение рисунка курсора во всех без исключения видеорежимах.

Изображение курсора отличается от обычных рисунков тем, что постоянно перемещается по экрану, следуя за перемещениями манипулятора "мышь". При этом оно должно быть четко видно на любом окружающем фоне и не должно оставлять следов от своего перемещения, за исключением тех случаев, когда такой след создается специально. На видимость и расположение курсора не должны влиять вывод новых рисунков на экран или удаление существующих. В некоторых случаях форма рисунка курсора может изменяться в зависимости от его местонахождения на экране или действий, выполняемых задачей в данный момент времени.

Поэтому при работе с изображением курсора выполняются специфические действия, которые не требовались при построении обычных рисунков. Прежде чем рассматривать эти действия, давайте разберемся, где можно взять и как подготовить рисунок курсора для его использования в задаче.

6.1.1. Курсоры для Windows

Наиболее доступными являются файлы, содержащие рисунки курсоров, подготовленные в стандарте Windows. Операционные системы Windows используют курсоры различной формы: стрелка, вертикальная черта, рука, песочные часы и пр. Конкретный рисунок курсора зависит от выполняемых действий и выбирается системой автоматически. Windows 9X позволяет изменять рисунки курсора при выборе "темы рабочего стола".

Windows 3.X работают с черно-белыми курсорами, заготовки рисунков которых хранятся в специальном файле и извлечь их из него не так просто. Однако существует специальное приложение MouseWarp, которое позволяет оператору выбирать рисунок курсора по своему усмотрению. В комплект этого приложения входит 19 файлов с заготовками рисунков курсоров, которые можно использовать для наших целей.

Windows 9X не только сама изменяет форму курсора, но и позволяет это сделать оператору. Прилагаемые к ней заготовки рисунков курсоров хранятся в отдельном каталоге (Cursors) и вы можете их использовать.

Структура файлов Icon. Семейство Windows использует один общий стандарт Icon для хранения файлов с заготовками рисунков курсоров и пиктограмм (значков). Спецификации файлов имеют тип (расширение) cur для курсоров и ico для пиктограмм.

К сожалению, автор не встречал точного описания структуры таких файлов, даже в справочнике Борна [4] содержатся явные неточности. Если вам попадется описание стандарта Icon для Windows, то ему не следует слепо доверять. Обязательно распечатайте дамп одного из доступных вам файлов и

сравните распечатку с вариантом описания. В качестве эталона можно взять файл `nc.ico`, входящий в комплект Norton Commander. Для версии NC 5.0 он содержит заготовку рисунка капитанской фуражки с красными цифрами 5.0.

Стандартный файл формата `Icon` состоит из четырех основных частей: заголовка, палитры цветов, заготовки рисунка и маски.

Первые восемь байтов заголовка содержат следующие данные:

- слово со смещением 0 всегда очищено (пустое), это признак формата `Icon`;
- слово со смещением 2 содержит тип рисунка: 1 — пиктограмма, 2 — курсор;
- слово со смещением 4 содержит количество хранящихся в файле рисунков (обычно 1);
- байт со смещением 6 содержит количество точек в строке (обычно `20h`);
- байт со смещением 7 содержит количество строк в рисунке (обычно `20h`).

Из других полей заголовка следует отметить слово с адресом 36 (`24h`), содержащее размер точки рисунка, выраженный в битах. Он равен 1 для черно-белых и 4 для цветных рисунков. Эта величина указывает способ распаковки рисунка и размер палитры.

Палитра используемых цветов располагается в файле, начиная с адреса `3Eh`. Она содержит 2 или 16 строк, в которых хранятся коды цветов в формате `b`, `g`, `r`, `0`. Заметим, что в таком формате хранится палитра в `BMP`-файлах для Windows (см. приложение А). В зависимости от количества цветов палитра занимает 8 (2 цвета) или 64 (16 цветов) байта.

Сразу после палитры размещается образ рисунка. Адрес его начала зависит от размера палитры и равен `46h` для черно-белых рисунков или `7Eh` для 16-цветных. Количество точек в рисунке фиксировано и составляет $32 \cdot 32 = 1024$ точки. Черно-белые рисунки упакованы по 8 точек в байте, а цветные — по 2 точки в байте. Соответственно, образ рисунка занимает в файле 128 или 512 байтов.

После образа рисунка располагается маска. Адрес ее начала `C6h` для черно-белых рисунков или `27Eh` для цветных. Маска — это черно-белый рисунок, упакованный по 8 точек в байте и занимающий 128 байтов. Адрес ее начала отстоит от конца файла на 128 байтов.

Образ рисунка и маска хранятся в перевернутом виде: первой записана последняя строка, второй — предпоследняя и т. д., последней в файле хранится первая строка рисунка или маски. Такой способ хранения данных используется в файлах формата `BMP` (см. приложение А).

Дамп файла с рисунком курсора. В примере 6.1 приведена распечатка (дамп) файла `left_00.cur`, входящего в комплект MouseWarp. Он содержит рисунок

стрелки, наклоненной вправо (обычно стрелка наклонена влево). Распечатка приведена в общепринятой шестнадцатеричной форме, каждой строке предшествует адрес ее начала в файле.

Пример 6.1. Распечатка (dump) файла Left_00.cur

Заголовок файла

```
000     00 00 02 00 01 00 20 20 00 00 0E 00 04 00 30 01
010     00 00 16 00 00 00 28 00 00 00 20 00 00 00 40 00
020     00 00 01 00 01 00 00 00 00 00 00 01 00 00 00 00
030     00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Палитра, содержащая описание черного и белого цветов

```
03E     00 00 00 00 FF FF FF 00
```

Рисунок курсора, упакованный по 8 точек в байте

```
046     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
056     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
066     06 00 00 00 06 00 00 00 03 00 00 00 03 00 00 00
076     01 80 00 00 01 84 00 00 00 CC 00 00 00 DC 00 00
086     00 FC 00 00 07 FC 00 00 03 FC 00 00 01 FC 00 00
096     00 FC 00 00 00 7C 00 00 00 3C 00 00 00 1C 00 00
0A6     00 0C 00 00 00 04 00 00 00 00 00 00 00 00 00 00
0B6     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Маска курсора, упакованная по 8 точек в байте

```
0C6     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0D6     FF FF FF FF FF FF FF FF FF FF FF FF F9 FF FF FF
0E6     F0 FF FF FF F0 FF FF FF F8 7F FF FF F8 7D FF FF
0F6     FC 39 FF FF FC 31 FF FF fE 01 FF FF FE 01 FF FF
106     E0 01 FF FF F0 01 FF FF F8 01 FF FF FC 01 FF FF
116     FE 01 FF FF FF 01 FF FF FF 81 FF FF FF C1 FF FF
126     FF E1 FF FF FF F1 FF FF FF F9 FF FF FF FD FF FF
136     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

Для того чтобы лучше понять, как хранятся и кодируются рисунок и маска, советуем вам нарисовать их на бумаге в клетку. В рассматриваемом примере они упакованы одинаково, по восемь точек в байте. Единица в разряде означает наличие точки (заштрихованная клетка на бумаге), а нуль — ее отсутствие (пустая клетка на бумаге). После построения вы увидите, что маска похожа на негативное изображение рисунка, но если их совместить, то окажется, что рисунок стрелки в маске оконтурен белой линией.

6.1.2. Предварительная подготовка рисунка

В исходном виде рисунок курсора и маска не удобны для многократного использования. Их надо распаковать, перевернуть, по возможности сократить и хранить в оперативной памяти до конца выполнения задачи.

Перед построением изображения курсора, как и любого рисунка, должна быть установлена палитра используемых цветов (см. главу 4). В данном случае палитра хранится в формате BMP для Windows, который описан в приложении А.

Замечание

Напомним, что в зависимости от способа установки палитры могут измениться коды точек рисунка курсора (но не маски).

Поворот рисунка и маски. Рисунок на экране проще строить в естественном порядке, т. е. в направлении слева направо и сверху вниз. В исходном виде рисунок и маска хранятся "вверх ногами", поэтому перед распаковкой их надо повернуть. При повороте переставляют 16 пар строк: первую строку с последней, вторую — с предпоследней и т. д.

В примере 6.2 показано, как можно программно переставить строки маски или черно-белого рисунка. Перед выполнением примера исходный файл должен быть прочитан в буфер, сегмент которого указывается в регистре fs, а смещение (адрес) рисунка или маски в буфере помещается в регистр di. Перевернутое изображение записывается на место исходного.

Пример 6.2. Поворот черно-белого рисунка или маски

```
        mov   si, di         ; копируем адрес первой строки
        add   si, 124        ; получаем адрес последней строки
        mov   cx, 16         ; количество пар строк
turn:   mov   eax, fs:[di]   ; eax = строка 1
        xchg  eax, fs:[si]   ; eax = строка 2; fs:[si] = строка 1
        mov   fs:[di], eax   ; fs:[di] = строка 2
        add   di, 04         ; адрес следующей строки
        sub   si, 04         ; адрес предыдущей строки
        loop  turn           ; управление повторами цикла
```

Выполнение примера 6.2 начинается с подготовки адреса последней строки и задания количества переставляемых пар. Перестановку выполняет цикл, его первая команда имеет метку turn. Она копирует в регистр eax первую строку пары. Следующая команда переставляет содержимое eax и второй строки пары. Третья команда копирует содержимое eax в первую строку. В результате переставлена пара строк. Затем корректируются адреса строк, и команда loop повторяет выполнение цикла 16 раз.

При перестановке расположение байтов в строке не изменяется, поскольку пересылку выполняет одна команда. Для переворота цветного рисунка этот пример не подходит, т. к. для перестановки 16-ти байтов пары строк надо организовать внутренний цикл, а во внешнем подготавливать адреса переставляемой пары.

Распаковка рисунка и маски. Повернутые рисунок и маску надо распаковать и сохранить в оперативной памяти. Для их хранения выделяется два массива, размером по 1024 байта (один байт на точку). В дальнейшем мы будем называть их `pntimage` и `pntmask`, первый содержит распакованный рисунок курсора, а второй — маску.

При распаковке черно-белого рисунка содержимое каждого байта обрабатывается, начиная со старшего разряда, и значение каждого бита (0 или 1) помещается в соответствующий байт массива `pntimage`.

При распаковке 16-цветного рисунка в байты массива `pntimage` записываются сначала старшая, а затем младшая тетрада каждого байта упакованного рисунка. Коды распакованных точек изменяются от `0` до `0Fh`.

Подпрограммы распаковки строк 16- и 2-цветных рисунков приведены в примерах 3.17 и 3.18, но они записывают результат в видеопамять. Применительно к данному случаю их надо изменить так, чтобы результат записывался в оперативную память, и организовать цикл для распаковки всего рисунка или маски.

После распаковки рисунка будут получены коды цветов, являющиеся адресами строк прилагаемой палитры. Маловероятно, чтобы они совпали с кодами цветов системной палитры, с которой работает задача. Едва ли в ней код белого цвета будет равен 1 или `0Fh`, как в прилагаемой к рисунку палитре. Поэтому у вас есть две возможности: либо преобразовать распакованные коды рисунка так, чтобы они соответствовали системной палитре, либо в системной палитре зарезервировать 2 или 16 первых регистров цвета для работы с курсором. Второй способ используется в Windows при работе в режимах `PPG`.

Последовательность действий при распаковке маски та же, что и при распаковке черно-белого рисунка. Если текущий бит маски содержит 0, то соответствующий байт массива `pntmask` очищается, но если текущий бит маски содержит 1, то устанавливаются *все разряды* соответствующего байта массива `pntmask` (в него записывается код `0FFh`). Это объясняется специфическим назначением маски — при ее наложении байты видеопамяти либо полностью очищаются, либо остаются без изменения.

Сокращение рисунка и маски. При выполнении графических задач курсор перемещается достаточно часто, поэтому желательно сократить до минимума действия, связанные с его построением и перемещением. Для этого, в частности, можно исключить из исходного рисунка не используемые (пустые) строки и столбцы.

Как правило, размеры рисунка меньше стандартного поля 32×32 точки. Например, изображение стрелки, хранящейся в файле `Left_00.cur` (см. пример 6.1) помещается в прямоугольнике шириной в 14 и высотой в 21 точку. Следовательно, для его хранения в памяти достаточно выделить не 1024, а

всего 294 байта. Очевидно, что при сокращении рисунка не только уменьшается занимаемое им пространство оперативной памяти, но и ускоряется процесс его построения и удаления. Рисунок и маска взаимосвязаны, поэтому при исключении строки или столбца рисунка надо исключить соответствующую строку или столбец маски.

Пример описания рисунка и маски. В примере 6.3 заготовка рисунка и маска описаны на языке ассемблера. Это распакованный файл из примера 6.1, в котором переставлены не только строки, но и столбцы, для того чтобы стрелка курсора была наклонена влево, а не вправо.

Пример 6.3. Описание рисунка и маски курсора

```
Pntimage db 00,00,00,00,00,00,00,00,00,00,00,00,00,00
         db 00,00,00,00,00,00,00,00,00,00,00,00,00,00
         db 00,00,FF,00,00,00,00,00,00,00,00,00,00,00
         db 00,00,FF,FF,00,00,00,00,00,00,00,00,00,00
         db 00,00,FF,FF,FF,00,00,00,00,00,00,00,00,00
         db 00,00,FF,FF,FF,FF,00,00,00,00,00,00,00,00
         db 00,00,FF,FF,FF,FF,FF,00,00,00,00,00,00,00
         db 00,00,FF,FF,FF,FF,FF,FF,00,00,00,00,00,00
         db 00,00,FF,FF,FF,FF,FF,FF,FF,00,00,00,00,00
         db 00,00,FF,FF,FF,FF,FF,FF,FF,FF,00,00,00,00
         db 00,00,FF,FF,FF,FF,FF,FF,FF,FF,FF,00,00,00
         db 00,00,FF,FF,FF,FF,FF,FF,00,00,00,00,00,00
         db 00,00,FF,FF,FF,00,FF,FF,00,00,00,00,00,00
         db 00,00,FF,FF,00,00,FF,FF,00,00,00,00,00,00
         db 00,00,FF,00,00,00,00,FF,FF,00,00,00,00,00
         db 00,00,00,00,00,00,00,FF,FF,00,00,00,00,00
         db 00,00,00,00,00,00,00,00,FF,FF,00,00,00,00
         db 00,00,00,00,00,00,00,00,00,FF,FF,00,00,00
         db 00,00,00,00,00,00,00,00,00,FF,FF,00,00,00
         db 00,00,00,00,00,00,00,00,00,00,00,00,00,00
pntmask  db FF,00,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF
         db FF,00,00,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF
         db FF,00,00,00,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF
         db FF,00,00,00,00,FF,FF,FF,FF,FF,FF,FF,FF,FF
         db FF,00,00,00,00,00,FF,FF,FF,FF,FF,FF,FF,FF
         db FF,00,00,00,00,00,00,FF,FF,FF,FF,FF,FF,FF
         db FF,00,00,00,00,00,00,00,FF,FF,FF,FF,FF,FF
         db FF,00,00,00,00,00,00,00,00,FF,FF,FF,FF,FF
         db FF,00,00,00,00,00,00,00,00,00,FF,FF,FF,FF
         db FF,00,00,00,00,00,00,00,00,00,00,FF,FF,FF
         db FF,00,00,00,00,00,00,00,00,00,00,00,FF,FF
```

```
db FF,00,00,00,00,00,00,00,00,00,00,00,00,FF
db FF,00,00,00,00,00,00,00,00,FF,FF,FF,FF,FF
db FF,00,00,00,00,00,00,00,00,FF,FF,FF,FF,FF
db FF,00,00,00,FF,FF,00,00,00,00,FF,FF,FF,FF
db FF,00,00,FF,FF,FF,00,00,00,00,FF,FF,FF,FF
db FF,00,FF,FF,FF,FF,FF,00,00,00,00,FF,FF,FF
db FF,FF,FF,FF,FF,FF,FF,00,00,00,00,FF,FF,FF
db FF,FF,FF,FF,FF,FF,FF,FF,00,00,00,00,FF,FF
db FF,FF,FF,FF,FF,FF,FF,FF,00,00,00,00,FF,FF
db FF,FF,FF,FF,FF,FF,FF,FF,FF,00,00,FF,FF,FF
```

В примере 6.3 метки `pntimage` и `pntmask` предшествуют директиве `db`, поэтому двоеточие после них не ставится. Если вы будете включать текст примера в свою программу, то все коды `FF` надо заменить на `0FFh` или на десятичное число `-1`. Здесь это не сделано только из соображений наглядности, чтобы можно было увидеть образованный из цифр рисунок. Текст примера лучше всего включить в сегмент данных вашей программы. Для того чтобы рисунок курсора был черно-белым, нулевой регистр цвета видеокарты должен быть очищен, а в последнем (255-м) регистре должен находиться код белого цвета (`3F`,`3F`,`3F`) (см. раздел 4.5).

Подведем итог всему сказанному в данном разделе. Курсор, хранящийся в файле формата `Icon`, не удобно использовать без предварительного преобразования рисунка и маски и установки палитры используемых цветов. Если вы хотите, чтобы ваша задача могла работать с файлами формата `Icon`, то в нее придется включить специальную процедуру, выполняющие описанные в данном разделе действия. Если же нужен только один рисунок курсора, то его преобразование проще выполнить вне задачи, а в ее исходный текст включить результат, как это сделано в примере 6.3.

Преобразования выполняются вручную или с помощью специально составленной программы, поскольку стандартные графические редакторы не работают с файлами формата `Icon`. Преобразование вручную занимает сравнительно немного времени. Сначала исходный файл преобразуется в символьную форму, т. е. в файл, содержащий шестнадцатеричные коды (дамп). А затем символьный файл редактируют с помощью любого текстового редактора, например, входящего в Norton Commander и вызываемого нажатием функциональной клавиши <F4>.

6.1.3. Немаскируемый курсор

При построении обычных рисунков их образы копируются в видеопамять, но если таким способом построить рисунок, образ которого приведен в примере 6.3, то изображение белой стрелки будет расположено на фоне черного прямоугольника. Очевидно, что работать с подобным изображением

курсора неудобно и черный фон, окружающий стрелку, надо убрать. Для исключения окружающего фона применяются маскировка, или специальные способы построения изображения курсора.

Один из таких способов мы уже использовали при построении текстового курсора, он описан в разделе 5.2.4, пример 5.24. Здесь нас интересует более универсальный вариант подобной подпрограммы, позволяющий строить изображение курсора произвольного размера и формы. Для записи кодов точек в видеопамять, по-прежнему, будет использоваться логическая операция `XOR`, вычисляющая функцию "исключающее ИЛИ" (`exclusive OR`).

Предварительные замечания. Образ рисунка курсора можно хранить в любом сегменте оперативной памяти. Учитывая его небольшой размер (294 байта), мы будем считать, что он расположен в сегменте данных (см. пример 6.3) и имеет имя `pntimage`. Маска при построении не используется, поэтому массив `pntmask` нас в данном случае не интересует.

Учитывая, что размеры рисунка не фиксированы и зависят от его формы, в разделе данных задачи надо описать две следующие переменные:

```
PntXsize  dw 14  ; количество точек в строке рисунка курсора
pntYsize  dw 21  ; количество строк в рисунке курсора
```

В приведенном описании значения переменных соответствуют размерам рисунка, показанного в примере 6.3.

Курсор является особым рисунком, его координаты в видеопамяти могут использоваться в различных целях. Поэтому они хранятся в специальных переменных, значение которых может изменяться *только* при перемещении манипулятора "мышь". В примере 6.8 будет описано несколько переменных, используемых при работе с курсором. Здесь нас интересуют только две из них. Переменная `Winpnt` содержит текущее окно видеопамяти, а `Offspnt` — адрес (смещение) точки левого верхнего угла рисунка курсора в этом окне.

Подпрограмма Tglpntr. Текст подпрограммы, изменяющей состояние курсора на противоположное, приведен в примере 6.4. При каждом нечетном вызове `Tglpntr` рисунок курсора появляется на экране, а при каждом четном на его месте восстанавливается исходный фон. Явно задаваемые входные параметры отсутствуют. Регистр `es` должен содержать код видеосегмента.

Пример 6.4. Подпрограмма переключения состояния курсора

```
Tglpntr: pusha               ; сохранение содержимого регистров
         push  Cur_win       ; сохранение исходного окна
         mov   ax, Winpnt    ; ax = окно с рисунком курсора
         mov   Cur_win, ax   ; Cur_win = Winpnt
         call  setwin        ; установка исходного окна
         lea   si, pntimage  ; si = адрес массива pntimage
```

```
        mov   di, Offspnt   ; di = адрес в сегменте видеопамяти
        mov   cx, pntYsize  ; cx = кол-во повторов внешнего цикла
        mov   bx, horsize   ; вычисляем константу для
        sub   bx, pntXsize  ; коррекции адресов строк
Displ_1:push  cx            ; сохраняем счетчик строк
        mov   cx, pntXsize  ; cx = количество точек в строке рисунка
Displ_2:lodsb               ; !! al = код очередной точки рисунка
        xor   es:[di], al   ; !! корректируем байт видеопамяти
        inc   di            ; !! увеличение адреса видеопамяти
        jnz   @F            ; -> адрес в пределах текущего сегмента
        call  nxtwin        ; конец сегмента, смена окна
@@:     loop  Displ_2       ; управление повторами цикла
        pop   cx            ; восстанавливаем счетчик строк
        add   di, bx        ; адрес начала следующей строки
        jnc   @F            ; -> адрес в пределах текущего сегмента
        call  nxtwin        ; конец сегмента, смена окна
@@:     loop  Displ_1       ; управление повторами цикла
        pop   Cur_win       ; восстановление Cur_win
        popa                ; восстановление содержимого регистров
        call  setwin        ; восстановление исходного окна
        ret                 ; возврат из подпрограммы
```

В подпрограмме примера 6.4 используется только 5 регистров — `ax`, `bx`, `cx`, `si` и `di`, но для сокращения ее текста первая команда `pusha` сохраняет в стеке содержимое всех регистров. Затем в стек помещается исходное значение переменной `Cur_win`, а ей присваивается новое значение и устанавливается соответствующее окно видеопамяти. В регистры `si`, `di` записываются адреса оперативной и видеопамяти, а в `cx` — количество строк в рисунке курсора. В конце подготовки в регистре `bx` формируется разность `horsize – pntXsize`, используемая в цикле построения для коррекции адресов строк видеопамяти.

Построение рисунка выполняют два вложенных цикла. Внешний имеет метку `Displ_1`. Он начинается с сохранения в стеке и изменения содержимого регистра `cx`, после чего выполняется внутренний цикл.

Цикл построения строки имеет метку `Displ_2`. Его первая команда `lodsb` считывает в регистр `al` байт, адрес которого находится в `ds:si`, и увеличивает содержимое регистра `si` на 1. Затем логическая операция `XOR` записывает содержимое регистра `al` в видеопамять. Регистр-посредник `al` нужен потому, что у команды `xor` (как и у команды `mov`) оба операнда не могут находиться в памяти.

После вывода очередной точки адрес видеопамяти увеличивается на 1, и если его значение осталось в пределах сегмента, то команда `jnz @F` обходит `call nxtwin`. В противном случае команда `call nxtwin` выполняется и уста-

навливается следующее окно. Последняя команда (`loop Disp_2`) повторяет выполнение цикла до тех пор, пока не будет нарисована вся строка.

При возврате во внешний цикл из стека выталкивается содержимое счетчика повторов и вычисляется адрес начала в видеопамяти следующей строки рисунка. Если при этом происходит переполнение, то устанавливается следующее окно видеопамяти. Команда `loop Disp_1` повторяет выполнение внешнего цикла до тех пор, пока не будет построен весь рисунок курсора.

После построения (или удаления) курсора из стека выталкивается содержимое переменной `Cur_win` и всех сохраненных регистров, восстанавливается исходное окно видеопамяти и происходит возврат на вызывающий модуль.

Недостатки немаскируемого курсора. Очевидными преимуществами работы с немаскируемым курсором являются следующие:

- для построения и удаления курсора нужна одна подпрограмма;
- подпрограмма выполняется сравнительно быстро;
- в оперативной памяти хранится только образ рисунка.

Однако такой способ построения имеет один существенный недостаток, сводящий на нет перечисленные преимущества.

Идея использования немаскируемого курсора основана на том, что при определенных значениях операнда-источника команда `xor` инвертирует код операнда-приемника или не изменяет его (см. раздел 5.2.4). В описанной подпрограмме источником являются точки заготовки рисунка, а приемником — точки видеопамяти. Образ рисунка черно-белый, коды его точек имеют значения либо `00`, либо `0FFh`. Поэтому при построении рисунка цвета точек экрана, расположенных под стрелкой, инвертируются, а окружающих стрелку не изменяются. Таким образом, цвет рисунка немаскируемого курсора на экране зависит от исходного цвета точек в том месте экрана, на котором он создается.

Вспомним табл. 4.1 из главы 4. При ее описании говорилось, что два цвета являются дополнительными, если при их наложении получается белый цвет. В частности, дополнением к черному цвету является белый, к синему — желтый, к красному — циан, к зеленому — мажента. Исходя из этого, можно представить, как изменяется цвет курсора в зависимости от исходного цвета точек экрана. Если же на экране находится какая-то картинка, т. е. цвет экрана не однороден, то и изображение курсора будет неоднородным. На пестром фоне оно может "потеряться" — стать трудно различимым для глаза.

При работе в режимах `PPG` описанная подпрограмма инвертирует не код цвета, а номер регистра видеокарты. Полученный при инверсии цвет будет зависеть от установленной (системной) палитры. Эта особенность успешно использовалась, например, в ранних версиях Windows — системная палитра подбиралась так, чтобы можно было использовать немаскируемый курсор.

В заключение заметим, что после описания маскируемого курсора в разделе 6.1.5 мы продолжим обсуждение некоторых вопросов, связанных с построением изображения курсора.

6.1.4. Маскируемый курсор

Маскировка является одним из способов исключения ненужных элементов изображения в процессе построения рисунка. Она применяется не только при выводе на экран курсоров и пиктограмм, но и во многих других случаях, например, при сборке рисунков из отдельных частей. Маска может быть подготовлена заранее с учетом особенностей рисунка или сформирована динамически, на основании анализа цветов строящегося рисунка. В данном разделе рассмотрена работа с готовой маской.

Как производится маскировка. В предыдущем разделе мы использовали тот факт, что при определенных условиях команда `xor` инвертирует значение операнда-приемника. Заметим также, что у этой команды есть еще одно полезное свойство. Вспомним таблицу истинности логической функции "исключающее ИЛИ" (раздел 5.2.4, табл. 5.1). Из нее, в частности, следует, что если один из двух операндов очищен, то результат выполнения команды `xor` будет равен значению другого операнда. Следовательно, при наложении двух цветов с помощью операции `xor` черный цвет становится прозрачным.

При построении маскируемого курсора та часть экрана, которую займет его изображение, предварительно окрашивается в черный цвет (очищается). С помощью команды `xor` (или `or`) на чистом месте можно построить рисунок любого цвета. Чтобы не портить окружающий фон в образе рисунка, точки, дополняющие его основную часть до прямоугольника, должны иметь черный цвет. Это условие обязательно выполняется у стандартных рисунков курсоров и пиктограмм (см. пример 6.3).

Для закрашивания в черный цвет на место расположения выводимого рисунка накладывается маска. Как говорилось в разделе 6.1.2, байты маски могут содержать только два значения кодов — `00` или `0FFh`. Маска является черно-белым рисунком, у которого черные точки соответствуют основной части маскируемого рисунка, а белые — черным точкам маскируемого рисунка, дополняющим его заготовку до прямоугольника. Посмотрите на пример 6.3, и вы увидите, что маска соответствует основному рисунку.

Фактически маска как рисунок не используется. Она записывается в видеопамять с помощью логической операции "И" (конъюнкция), которую вычисляет команда `and`. При выполнении этой команды операнд приемника будет очищен, если очищен операнд источника и не изменится, если у операнда источника установлены все разряды. При маскировке источником являются байты маски, а приемником — байты видеопамяти. Поэтому после наложения маски очищенными будут только те байты видеопамяти, которые очищены в маске, а содержимое остальных не изменится.

Если вы внимательно проанализируете пример 6.3, то обнаружите что находящаяся в массиве `pntmask` маска очищает несколько бóльшую часть экрана, чем нужно для размещения стрелки, хранящейся в массиве `pntimage`. Это сделано для того, чтобы белая стрелка не потерялась на белом фоне. В результате наложения такой маски рисунок курсора на экране окажется окруженным черной окантовкой, и белая стрелка будет видна на любом фоне.

Схема построения рисунка. Для получения изображения маскируемого курсора на экране надо сохранить исходный фон, наложить на этот фон маску, полученный результат объединить с заготовкой рисунка с помощью команды `xor` или `or` и записать его в видеопамять.

Сохранение исходного фона необходимо потому, что восстановить его после построения рисунка курсора невозможно. Для размещения сохраняемого фона в оперативной памяти надо зарезервировать массив `pntbuff`. Его размер соответствует размеру рисунка курсора в байтах, т. е. совпадает с размерами массивов `pntimage` и `pntmask`. Каждый из этих трех массивов занимает в памяти `pntXsize*pntYsize` байтов. Массив `Pntbuff` надо расположить в том же сегменте, в котором находятся `pntimage` и `pntmask`. Мы будем считать, что они размещены в разделе данных задачи и доступ к ним происходит через сегментный регистр `ds`. При этом буфер описывается так:

```
pntbuff  db 294 dup (?); резервирование 294 байтов в разделе данных
```

Если вы расположите массивы в другом сегменте, то в подпрограмму придется внести незначительные изменения, о чем будет сказано после ее описания. Порядок расположения массивов не имеет значения, важно чтобы они находились в одном сегменте.

Подпрограмма Showpnt. Описанные действия выполняются в одном цикле, который повторяется для каждой точки прямоугольной области, в которой располагается рисунок курсора. Текст подпрограммы приведен в примере 6.5. Входные параметры явно не задаются, они находятся в тех переменных, которые были описаны в предыдущем разделе. Регистр `es` должен содержать код видеосегмента (хранящийся в `Vbuff`).

Пример 6.5. Подпрограмма построения рисунка маскируемого курсора

```
Showpnt: pusha                  ; сохранение содержимого регистров
         push  Cur_win          ; сохранение исходного окна
         mov   ax, Winpnt       ; ax = окно с рисунком курсора
         mov   Cur_win, ax      ; Cur_win = Winpnt
         call  setwin           ; установка исходного окна
         xor   si, si           ; очистка регистра si
         mov   di, Offspnt      ; di = адрес в сегменте видеопамяти
         mov   cx, pntYsize     ; cx = кол-во повторов внешнего цикла
         mov   bx, horsize      ; вычисляем константу для
         sub   bx, pntXsize     ; коррекции адресов строк
```

```
Sh_1:   push  cx                ; сохраняем значение счетчика строк
        mov   cx, pntXsize      ; cx = количество точек в строке рисунка
Sh_2:   mov   al, es:[di]       ; !! al = код точки исходного фона
        mov   pntbuff[si], al   ; !! сохраняем его в pntbuff
        and   al, pntmask[si]   ; !! накладываем маску
        xor   al, pntimage[si] ; !! формируем новый код точки
        stosb                   ; !! и записываем его в видеопамять
        inc   si                ; !! коррекция адреса рисунка
        or    di, di            ; адрес в пределах текущего окна ?
        jnz   @F                ; -> да, обходим следующую команду
        call  nxtwin            ; установка следующего окна
@@:     loop  sh_2              ; управление повторами цикла
        pop   cx                ; восстанавливаем счетчик строк
        add   di, bx            ; корректируем адрес видеопамяти
        jnc   @F                ; -> адрес в пределах текущего окна
        call  nxtwin            ; установка следующего окна
@@:     loop  sh_1              ; управление внутренним циклом
        pop   Cur_win           ; восстановление значения Cur_win
        popa                    ; восстановление всех регистров
        call  setwin            ; восстановление исходного окна
        ret                     ; возврат из подпрограммы
```

Начало примера 6.5 отличается от примера 6.4 только одной командой. Вместо записи адреса массива `pntimage` в регистр `si` последний просто очищается. Это сделано потому, что регистр `si` используется для доступа к трем массивам, а не к одному, как это было в примере 6.4.

Принципиальное различие между примерами в основных действиях, выполняемых во внутреннем цикле, который в данном случае имеет метку `Sh_2`. Первая команда внутреннего цикла считывает код очередной точки из видеопамяти в регистр `al`, а вторая сохраняет его в очередном байте массива `pntbuff`. После этого в регистр `al` помещается результат вычисления логической функции "И" от исходного значения регистра и содержимого очередного байта маски. В зависимости от кода байта маски регистр `al` либо будет очищен, либо его исходное значение не изменится — третьего не дано. Следующая команда завершает формирование нового кода точки, она вычисляет логическую функцию "исключающее ИЛИ" от содержимого регистра `al` и очередного байта массива `pntimage`. Остается записать новый код точки в видеопамять, что и делает команда `stosb`, одновременно она увеличивает содержимое регистра `di` на 1.

Основные действия выполнены, шестая команда увеличивает на 1 адрес оперативной памяти (содержимое регистра `si`). Адрес видеопамяти (содержимое `di`) увеличила команда `stosb`, но надо проверить, остался он в пределах текущего видеосегмента или нет. Признаком выхода за пределы сегмента является нуль в регистре `di`, при этом подпрограмма `nxtwin` установит

следующее окно видеопамяти. Повторами внутреннего цикла управляет команда `loop Sh_2`.

После построения строки из стека восстанавливается значение счетчика повторов, вычисляется адрес начала следующей строки в видеопамяти и команда `loop Sh_1` повторяет выполнение внешнего цикла до тех пор, пока на экран не будут выведены все строки изображения курсора.

В заключение из стека выталкиваются исходные значения переменной `Cur_win` и всех регистров, восстанавливается исходное окно видеопамяти и происходит возврат на вызывающий модуль.

Массивы в другом сегменте. Текст примера 6.5 составлен из расчета на то, что массивы `pntimage`, `pntmask` и `pntbuff` расположены в разделе данных и для доступа к ним используется регистр `ds`, имя которого не указывается перед операндами. Если вы предпочитаете расположить указанные массивы в другом сегменте, то в трех командах примера 6.5 перед именами массивов надо явно указать имя выбранного вами сегментного регистра. Например, `mov fs:pntbuff[si], al`, если для доступа к массиву `pntbuff` используется сегментный регистр `fs`.

Важно

Еще раз напоминаем, что все три массива должны располагаться в одном сегменте.

Действия при удалении курсора. Перед перемещением курсора его старое изображение удаляется с экрана. Кроме того, изображение курсора удаляется перед выводом на экран новых рисунков. В разделе 3.3.4 было описано, в каких случаях и почему это надо делать.

Для удаления изображения курсора надо восстановить исходный фон, сохраненный при его построении. Эта процедура ничем не отличается от построения небольшого рисунка, образ которого находится в оперативной памяти, только образом рисунка является исходный фон на месте, построения изображения курсора. В примере 3.21 (см. раздел 3.3.2) описана подпрограмма построения небольшого рисунка. При выполнении графических задач курсор перемещается достаточно часто, поэтому для восстановления исходного фона лучше составить специальную подпрограмму, а не использовать одну из общедоступных. Если же задача работает с указателем мыши в режиме прерываний, то без специальной подпрограммы просто не обойтись. Поэтому мы перепишем пример 3.21 применительно к данному случаю.

Подпрограмма Hidepnt. Текст подпрограммы, выполняющей удаление изображения курсора, приведен в примере 6.6. В нем использованы те же переменные, что и в примерах 6.4 и 6.5, а из трех массивов нужен только `pntbuff`, содержащий ранее сохраненный фон.

Пример 6.6. Восстановление исходного фона на месте рисунка курсора

```
Hidepnt: pusha                ; сохранение содержимого регистров
         push  Cur_win        ; сохранение исходного окна
         mov   ax, Winpnt     ; ax = окно с рисунком курсора
         mov   Cur_win, ax    ; Cur_win = Winpnt
         call  setwin         ; установка исходного окна
         lea   si, pntbuff    ; si = адрес буфера с сохраненным фоном
         mov   di, Offspnt    ; di = адрес в сегменте видеопамяти
         mov   cx, pntYsize   ; cx = кол-во повторов внешнего цикла
         mov   bx, horsize    ; вычисляем константу для
         sub   bx, pntXsize   ; коррекции адресов строк
hid_1:   push  cx             ; сохраняем значение счетчика строк
         mov   cx, pntXsize   ; cx = количество точек в строке рисунка
hid_2:   movsb                ; !! копируем байт из pntbuff в видеопамять
         or    di, di         ; конец сегмента видеопамяти ?
         jnz   @F             ; -> нет, обходим следующую команду
         call  nxtwin         ; устанавливаем следующее окно
@@:      loop  hid_2          ; управление повторами цикла
         pop   cx             ; восстанавливаем счетчик строк
         add   di, bx         ; корректируем адрес видеопамяти
         jnc   @F             ; -> адрес в пределах текущего окна
         call  nxtwin         ; установка следующего окна
@@:      loop  hid_1          ; управление повторами цикла
         pop   Cur_win        ; восстановление значения Cur_win
         popa                 ; восстановление всех регистров
         call  setwin         ; восстановление исходного окна
         ret                  ; возврат из подпрограммы
```

В примере 6.6 выполняются действия, которые уже неоднократно обсуждались, поэтому мы опустим его подробное описание. Автор надеется, что читатель разберется в том, что делают конкретные команды.

Сравнение способов построения. В заключение раздела оценим, что мы получаем и что теряем при работе с маскируемым курсором. Бесспорное преимущество подпрограммы `Showpnt` в том, что возможна работа с изображением курсора, цвет которого зависит *только* от самого рисунка и *не зависит* от находящейся на экране картинки. Собственно говоря, ради этого и применяется маскировка.

Но при этом, объем оперативной памяти, необходимый для хранения рабочих массивов, увеличился в три раза. При описанном способе построения изображения курсора сократить его невозможно. Попробуйте самостоятельно ответить на вопрос — почему нельзя временно сохранять исходный фон в массиве `pntimage` и восстанавливать его содержимое при восстановлении исходного фона?

При построении немаскируемого курсора обработку кода каждой точки выполняли 2 команды внутреннего цикла. В примере 6.5 таких команд стало 5, и сократить их количество невозможно.

Наконец, вместо одной подпрограммы `Tglpntr` при использовании маски нужны две — `Showpnt` и `Hidepnt`. Объединить их в одну подпрограмму невозможно. Включение и выключение курсора это два независимых процесса, которые не могут выполняться одновременно.

Такова реальная плата за улучшение качества изображения и возможность работы с цветным рисунком курсора. Выбор маскировки или отказ от нее зависит от вас.

6.1.5. Замечания к описанным подпрограммам

В двух предыдущих разделах описаны действия, которые надо выполнить для построения или удаления изображения курсора. Здесь мы рассмотрим, как изменяются команды, выполняющие эти действия в зависимости от тех или иных дополнительных условий. Нас будут интересовать способы ускорения работы с курсором и построения изображения в режимах `direct color`, когда цвет указывается непосредственно в коде точки.

Ускорение работы с курсором. Курсор является основным средством управления процессом выполнения графических задач. Чем меньше времени затрачивается на перемещение его рисунка, тем больше времени остается на выполнение основных действий. Поэтому ускорение манипуляций с курсором представляет определенный практический интерес.

Для ускорения работы трех описанных подпрограмм надо сократить количество действий, выполняемых в их внутренних циклах.

В подпрограмме `Hidepnt` (пример 6.6) основное действие выполняет одна команда `movsb` (ее метка `hid_2`), поэтому в этом случае применимы способы ускорения построения строк, описанные в разделе 3.3.1. В частности, вместо внутреннего цикла можно использовать подпрограмму примера 3.16, внеся в нее незначительные изменения.

К подпрограммам `Tglpntr` (см. пример 6.4) и `Showpnt` (см. пример 6.5) описанные в разделе 3.3.1 способы ускорения не применимы, поскольку в них основные действия выполняют несколько (2 или 5) команд.

На первый взгляд достаточно просто изменить основные команды так, чтобы они оперировали не с байтами, а со словами или двойными словами, т. е. обрабатывали коды сразу двух или четырех точек. В первом случае количество повторов внутреннего цикла сократится в 2, а во втором — в 4 раза. Перед входом в цикл содержимое регистра `cx` (`pntXsize`) надо уменьшить, соответственно, в 2 или в 4 раза.

Такая замена дает нужный результат, но необходимы дополнительные меры защиты от возможной аварийной ситуации. Давайте разберемся в причине ее возникновения.

Курсор может находиться в любом месте экрана, поэтому вполне вероятно, что одна из строк его изображения расположится в смежных окнах видеопамяти. Если при этом первая точка строки имеет *нечетный* адрес в видеопамяти, то обрабатывать одной командой сразу две точки такой строки нельзя. При чтении или записи одной из пар точек первый байт операнда окажется в пределах, а второй за пределами текущего сегмента. Это одна из типичных аварийных ситуаций, чаще всего она приводит к тому, что на программистском жаргоне называется "компьютер завис", т. е. он не реагирует ни на какие внешние события, кроме выключения питания.

Для исключения аварийной ситуации можно, например, сделать так, чтобы при работе с курсором адрес его начала в видеопамяти всегда был четным. Изображение курсора следует за манипулятором "мышь". Если при опросе состояния последнего окажется, что он находится в столбце с нечетным номером, то этот номер принудительно делается четным. Такой трюк уменьшает точность позиционирования курсора на экране, поэтому вам придется выбирать меньшее из двух зол. К вопросу о точности позиционирования мы вернемся при описании программирования работы с мышью.

Таким образом, при некотором ограничении точности позиционирования выполнение подпрограмм `Tglpntr` и `Showpnt` можно ускорить в 2 раза, внеся в них описанные выше изменения. Более существенное ускорение связано со значительным увеличением размеров текстов подпрограмм и едва ли целесообразно.

Изменения для режимов direct color. Подробному описанию особенностей программирования для видеорежимов с указанием цвета в коде точки посвящена глава 7. Здесь мы только покажем, какие изменения надо внести в описанные подпрограммы для их использования при работе в режимах `direct color`. Это позволит в дальнейшем не повторять описание способов построения курсора. Вы можете пропустить эту часть раздела и вернуться к ней после прочтения главы 7.

В режимах `PPG` код точки является номером регистра цвета видеокарты, а в режимах `direct color` он является кодом конкретного цвета и занимает 16 разрядов в режиме `Hi-Color` и 32 разряда в режиме `True Color`.

Прежде всего, вам придется изменить заготовку рисунка и маску, которые хранятся в массивах `pntimage` и `pntmask` (см. пример 6.3).

Проще всего изменить описание маски и черно-белого рисунка курсора. В пояснениях к примеру 6.3 мы советовали при его использовании в конкретной программе заменить все коды `0FF` десятичным числом `-1`. Если вы это сделали, то остается только заменить все директивы `db` на `dw` для режима

Hi-Color или на dd для True Color. Макроассемблер зарезервирует требуемое пространство памяти и заменит число −1 кодами 0FFFFh или 0FFFFFFFFh, в зависимости от указанной директивы (dw или dd).

Изменить описание цветного рисунка курсора сложнее, в этом случае недостаточно простой замены директив db на dw или dd. У заготовок цветных рисунков распакованный код точки является порядковым номером строки палитры, хранящейся вместе с рисунком. По коду точки надо выбрать из палитры код цвета, преобразовать его в нужную форму и поместить в описание рисунка. Такое преобразование делается программно, а не вручную. В главе 7 описаны способы преобразования рисунков из формата PPG в форматы direct color, их и можно использовать. Однако на первое время лучше ограничиться черно-белым курсором, а к работе с цветным перейти позже, по мере накопления опыта программирования графики.

В режимах direct color размеры массивов pntimage и pntmask увеличиваются в 2 или в 4 раза, во столько же раз надо увеличить размер массива pntbuff. Это можно сделать одним из двух способов: заменить в его описании директиву db на dw или dd, либо оставить директиву db, а количество резервируемых байтов умножить на 2 или на 4.

Изменения в текстах подпрограмм примеров 6.4, 6.5 и 6.6 связаны только с увеличением размера кода точки в 2 (режим Hi-Color) или 4 (режим True Color) раза. Прежде всего, нужно увеличить значение константы, которая используется для коррекции адресов строк. Во всех примерах ее вычисляют две следующие команды:

```
mov     bx, horsize   ; вычисляем константу для
sub     bx, pntXsize  ; коррекции адресов строк
```

После них надо записать третью команду, сдвигающую содержимое регистра bx на один (shl bx, 1) или на 2 (shl bx, 2) разряда влево. Значение константы увеличится, соответственно, в 2 или в 4 раза.

Остальные изменяемые команды расположены во внутренних циклах. Комментарий к ним начинается с двух восклицательных знаков. Изменения этих команд делятся на следующие три категории:

- у строковых команд lodsb, stosb и movsb последняя буква (b) заменяется буквами w (Hi-Color) или d (True Color);
- если один из операндов команды находится в регистре al, то имя регистра надо изменить на ax (Hi-Color) или на eax (True Color);
- команды inc di и inc si увеличивают значение адреса на 1. Они заменяются командой сложения (add), которая прибавляет к регистру число 2 (Hi-Color) или 4 (True Color).

Перечисленные изменения делают возможным применение описанных подпрограмм при работе в видеорежимах с указанием цвета в коде точки.

Промежуточные итоги. При программировании конкретной задачи важно не только составить нужную подпрограмму, но и корректно ее использовать. Применительно к нашему случаю это означает следующее:

1. При выводе изображения курсора на экран вы должны быть уверены в том, что его там уже нет, в противном случае на экране может оказаться несколько изображений курсоров или будет испорчен фон, сохраненный при выводе предыдущего изображения. Такая уверенность есть при первом выводе курсора в начале выполнения задачи, но после этого задача должна контролировать его текущее состояние.
2. Перед удалением курсора также надо убедиться в том, что он находится на экране, а исходный фон сохранен в массиве `pntbuff`. В противном случае при попытке удалить курсор на экране появится прямоугольник, цвет и узор которого не соответствует ожидаемым.

При организации работы с текстовым курсором в разделах 5.2.4 и 5.2.5 мы использовали специальный признак, указывающий текущее состояние текстового курсора. Работа с графическим курсором имеет специфические особенности, а способы определения его текущего состояния зависят от того, как задача получает данные от манипулятора "мышь".

Если текущее положение манипулятора задача определяет в режиме опроса, то специальный признак состояния курсора не нужен, оно определяется логикой выполняемых действий. Однако если задача взаимодействует с манипулятором в режиме прерываний, то без указанного признака не обойтись. Подробное обсуждение этих вопросов будет производиться в процессе описания программирования работы с манипулятором "мышь", к которому мы и переходим.

6.2. Подготовка к работе с манипулятором "мышь"

Манипулятор "мышь" преобразует свое перемещение в электрические сигналы и посылает их в компьютер. Наибольшее распространение получили электромеханические устройства, у которых датчиком перемещений является металлический шарик. Его вращение разлагается на два направления по осям `X` и `Y`, преобразуется в электрические сигналы и поступает в компьютер по соединительному кабелю. На верхней части манипулятора расположены две или три кнопки, данные об их состоянии также передаются в компьютер.

Отдельные модели манипуляторов различаются не только по устройству и внешнему оформлению, но и по расположенному в них электронному блоку, формирующему электрические сигналы при перемещении мыши и нажатии на ее кнопки. Кроме того, могут различаться кабели и разъемы, с помощью которых мышь подключается к компьютеру.

Указанные различия не влияют на способы программирования работы с манипулятором. Конкретные особенности устройства управления мышью "спрятаны" в драйвере, который поставляется в комплекте с устройством. Поэтому при покупке вы можете выбирать ту модель манипулятора, которая вам больше нравится, например, по оформлению.

6.2.1. Общее описание драйвера мыши

От манипулятора в компьютер поступает первичная информация, которая не пригодна для непосредственного использования в прикладных задачах. Предварительную обработку этой информации выполняет специальная программа — драйвер. При перемещении мыши или при нажатии на одну из ее кнопок возникает аппаратное прерывание, в результате которого приостанавливается выполнение текущего процесса и происходит обращение к драйверу. Он обрабатывает поступившие данные и сохраняет результат в своих внутренних переменных, после чего может быть выполнена специальная прерывающая подпрограмма или завершена работа драйвера и продолжено выполнение приостановленного процесса.

Подчеркнем, драйвер только фиксирует наступление события — нажатие на одну из кнопок или перемещение мыши. Реагировать на само событие должна прикладная задача или одна из компонент операционной системы. Если в данный момент с мышью не работает ни одна задача, то событие останется не востребованным. Поэтому наличие драйвера необходимое, но не достаточное условие для организации взаимодействия с мышью.

Установка драйвера. Драйвер является резидентной, т. е. постоянно находящейся в памяти задачей. В процессе загрузки DOS находит файл, содержащий эту задачу, помещает его содержимое в оперативную память и выполняет первый запуск. При этом драйвер настраивается на дальнейшую работу, после чего продолжается процесс загрузки DOS.

Файл, содержащий драйвер, должен находиться в одном из каталогов жесткого диска. Часто, но не всегда, он имеет имя `mouse`, а его тип может быть `com` или `sys`. Тип влияет только на способ первоначальной установки драйвера и не влияет на дальнейшую работу с ним. Если файл имеет тип `com` (например, `mouse.com`), то его полная спецификация (путь поиска, имя и тип файла) указывается в системном файле `autoexec.bat`. А если он имеет тип `sys` (например, `mouse.sys`), то спецификация указывается в файле `config.sys`.

Обычно при продаже к мыши прилагается дискета, содержащая программу для установки драйвера и текстовый файл (его имя `readme`, или нечто подобное), с рекомендациями по установке. Чаще всего установка сводится к копированию нужных файлов в один из каталогов жесткого диска и включения имени файла драйвера в `autoexec.bat` или `config.sys`.

Если по каким-то причинам у вас есть мышь без установочной дискеты, попытайтесь использовать любой доступный драйвер, скорее всего вам это

удастся. Современные модели манипуляторов, как правило, соответствуют стандарту Microsoft Mouse, поэтому обслуживать их могут все драйверы, при разработке которых были учтены требования этого стандарта.

Основные функции драйвера выполняются независимо от вычислительной среды. Поэтому Windows 3X может использовать установленный в DOS или свой собственный драйвер, загружаемый вместе с системой. Windows 9X являются самостоятельными операционными системами, не зависящими от DOS, поэтому они обязательно загружают драйвер.

Доступ к драйверу. Если драйвер установлен, то при работе в среде DOS адрес его точки входа хранится в векторе `33h`. Поэтому для обращения к нему прикладные задачи должны использовать командное прерывание `int 33h`. Предварительно в регистре `ax` указывается код запрашиваемой функции, который может изменяться в пределах от `0` до `24h`. Если для выполнения функции нужны входные параметры, то их значения передаются в регистрах общего назначения. В тех же регистрах драйвер возвращает выходные параметры (запрашиваемые данные), если таковые имеются.

Например, для приведения драйвера в первоначальное состояние (сброс или инициализация), в задаче надо выполнить две следующие команды:

```
mov   ax, 0 ; ax = код запроса, в данном случае 0
int   33h   ; обращение к драйверу для исполнения запроса
```

В результате внутренние переменные драйвера принимают те значения, которые они имели при первоначальной загрузке. Мы еще раз вернемся к рассмотрению данного запроса в следующем разделе.

Сводка функций драйвера. Основной набор функций, выполняемых всеми драйверами, устоялся. Он описан, например, в разделе Mouse Support электронной справочной системы `Tech Help`. Краткое описание функций, выполняемых конкретным драйвером, как правило, находится на установочной дискете, прилагаемой к манипулятору.

В табл. 6.1 перечислены функции, входящие в основной набор. Указанные в первом столбце таблицы коды являются шестнадцатеричными числами. Обратите внимание на отсутствие кодов `11h`, `12h`, `1Ch`, `22h` и `23h`. Конкретный драйвер может выполнять дополнительные функции, с этими или другими кодами. Однако они мало что добавляют к основному набору, и дополнительные функции лучше не использовать, исходя из соображений совместимости задачи с любыми моделями драйверов.

Таблица 6.1. *Список основных функций драйвера мыши*

Код	Запрашиваемое (исполняемое) действие
00	Инициализация драйвера (настройка на работу с мышью)
01	Включить (нарисовать на экране) изображение курсора

Таблица 6.1 *(окончание)*

Код	Запрашиваемое (исполняемое) действие
02	Выключить (удалить с экрана) изображение курсора
03	Опрос текущих координат курсора и состояния всех кнопок
04	Установить текущие координаты курсора
05	Опрос счетчика нажатий указанной кнопки и координат
06	Опрос счетчика отпусканий указанной кнопки и координат
07	Установить пределы перемещения курсора по горизонтали
08	Установить пределы перемещения курсора по вертикали
09	Установить форму курсора в графическом режиме
0a	Установить форму курсора в текстовом режиме
0b	Определить расстояние последнего перемещения в `mickeys`
0c	Установить подпрограмму для обработки событий
0d	Разрешить эмуляцию светового пера
0e	Запретить эмуляцию светового пера
0f	Установить шаг курсора при медленном перемещении мыши
10	Установить область, в которой курсор не виден
13	Установить шаг курсора при быстром перемещении мыши
14	Изменить подпрограмму, установленную по коду `0c`
15	Получить размер внутреннего буфера состояния драйвера
16	Сохранить в памяти внутренний буфер состояния драйвера
17	Восстановить ранее сохраненный буфер состояния драйвера
18	Установить адрес специальной подпрограммы обработки событий
19	Определить адрес подпрограммы, установленной по коду 18
1a	Установить чувствительность мыши в процентах (0—100)
1b	Определить чувствительность мыши в процентах (0—100)
1d	Установить страницу, на которой должен находиться курсор
1e	Определить страницу, на которой находится курсор
1f	Дезактивация драйвера (программное отключение от мыши)
20	Восстановление работы дезактивированного драйвера
21	Программный сброс драйвера (неполный аналог кода `00`)
24	Определить тип мыши, драйвера и используемый порт

По назначению выполняемых действий функции, перечисленные в табл. 6.1, можно разделить на несколько групп.

Управление курсором. При установке стандартных текстовых или графических режимов IBM драйвер самостоятельно рисует, удаляет и перемещает указатель мыши, что существенно упрощает структуру прикладных задач, работающих с мышью. Тем не менее, задача должна иметь возможность влиять на выполнение драйвером указанных действий. Для этого в базовый набор команд включено 9 функций, коды которых в табл. 6.1 начинаются и заканчиваются символом "*".

Они позволяют задаче в нужные моменты времени включать и выключать курсор и изменять его форму. По умолчанию драйвер выбирает изображение указателя мыши (курсора) в зависимости от установленного видеорежима. В графических режимах оно имеет форму наклоненной влево стрелки, а в текстовых — прямоугольника. В текстовых режимах курсор перемещается не плавно, а скачками из одного знакоместа в другое. Задача может задавать размер этого скачка при обычном и быстром перемещении мыши.

Остается только сожалеть о том, что эти полезные функции нельзя использовать при работе в графических режимах VESA. Как уже говорилось в предисловии к разделу 6.1, драйвер не может определить характеристики этих режимов, необходимые для построения изображения курсора.

Установочные команды. Наиболее важной функцией драйвера является увязка перемещений мыши с позицией курсора на экране. При выполнении этой функции используются внутренние переменные и счетчики позиций, которые должны иметь определенные значения. Часть из них зависит от характеристик мыши и формируется при установке или инициализации драйвера. Другая часть значений зависит от установленного видеорежима, их должна определять задача.

Функции с кодами `00`, `04`, `07`, `08`, `0Fh`, `13h`, `1Ah`, `1Bh`, `21h` позволяют изменять текущие настройки драйвера. С их помощью задача может инициализировать драйвер, установить пределы и скорость перемещения курсора, изменить чувствительность драйвера к перемещениям мыши. Подробное описание этих функций приведено в следующем разделе.

Информационные команды. Для определения текущих координат мыши и состояния ее кнопок предназначены функции с кодами `03`, `05`, `06` и `0Bh`. Примеры использования этих функций мы рассмотрим при описании работы с мышью в режиме опроса ее состояния.

Обслуживание прерываний. Альтернативой режиму опроса состояния является режим прерываний, при котором задача получает информацию от драйвера только при наступлении конкретного события — изменения позиции мыши или состояния ее кнопок. В теле задачи должны быть предусмотрены подпрограммы, выполняющие действия, связанные с данным событием

(или событиями), например перемещение рисунка курсора на новое место. С помощью функций с кодами 0Ch, 14h, 18h и 19h задача может сообщить драйверу адрес прерывающей подпрограммы и событие, на которое она реагирует. Примеры использования этих функций мы рассмотрим при описании работы с мышью в режиме прерываний.

Специальные функции. Пять команд с кодами 15h, 16h, 17h, 1Fh и 20h выполняют специфические действия, которые нужны только в особых случаях. В первую очередь к ним относится смена драйвера при выполнении задачи (как правило, системной, а не прикладной). Простой замены содержимого вектора 33h в этом случае недостаточно, поскольку при первоначальной установке драйвера настраивается контроллер прерываний и изменить эти настройки можно только с помощью специальной функции 1Fh, которая выполняет полную дезактивацию драйвера и возвращает в регистрах es:bx (возможно es:dx) содержимое вектора 33h. Дезактивированный драйвер остается в оперативной памяти. После этого задача может устанавливать свой драйвер или использовать вектор 33h для других целей. Перед выходом из задачи работа драйвера восстанавливается с помощью функции 20h, которая не требует задания входных параметров.

В некоторых случаях может понадобиться сохранить текущие настройки драйвера перед их изменением и спустя некоторое время восстановить первоначальные значения. Все внутренние переменные и счетчики хранятся в специальном буфере состояния драйвера. Порядок действий при сохранении и восстановлении содержимого этого буфера следующий. С помощью функции 15h задача определяет размер буфера состояния, выделяет соответствующее пространство оперативной памяти и помещает адрес его начала в регистры es:dx. После этого она запрашивает выполнение функции 16h, которая сохраняет текущее состояние. Теперь можно изменять текущие установки драйвера мыши.

Для восстановления исходного состояния адрес буфера, в котором оно записано, надо поместить в регистры es:dx и обратиться к драйверу с запросом 17h.

Такова общая характеристика базового набора функций, выполняемых драйвером мыши. Теперь мы переходим к рассмотрению способов программирования работы с ним в режимах VESA.

6.2.2. Предварительные действия

В данном разделе описана настройка драйвера и задачи на совместную работу. Выполняемые при этом действия не зависят от того, как задача взаимодействует с драйвером, — периодически обращаясь к нему, или в режиме прерываний. Они заключаются в инициализации драйвера, установке границ рабочего поля и исходной позиции курсора. Дополнительно может быть

выбрана чувствительность драйвера и курсора к перемещениям мыши. Настройка выполняется после установки видеорежима и получения его характеристик.

Новое макроопределение. В примере 2.12 главы 2 были описаны макроопределения `PushReg` и `PopReg`, которые неоднократно использовались в примерах подпрограмм. Добавим к ним новое макроопределение, формирующее команды запроса функций драйвера мыши. Оно почти не сокращает текст программы, но делает его более наглядным и понятным. Описание макроопределения приведено в примере 6.7, оно должно располагаться перед основным текстом программы.

Пример 6.7. Макроопределение для обращений к драйверу мыши

```
Mouse   macro fun       ; номер функции задает параметр fun
        mov   ax, fun&h ; номер функции помещается в ax
        int   33h       ; обращение к драйверу
endm                    ; конец макроопределения.
```

Макровызов этого определения имеет вид `Mouse fun`, где вместо `fun` указывается шестнадцатеричный номер вызываемой функции без буквы `h` в конце. Обнаружив вызов, Макроассемблер находит одноименное определение, обрабатывает его и включает в задачу две команды. Первая из них пересылает в регистр `ax` указанный в макровызове код `fun`, к которому добавляется буква `h`. Вторая команда `int 33h` выполняет обращение в драйверу мыши. Например, на месте макровызова `Mouse 21` в тексте задачи окажутся две следующие команды:

```
mov     ax, 21h       ; запись в ax кода 21h
int     33h   ; обращение к драйверу мыши
```

Замечание

Обратите внимание на то, что макровызов `Mouse 21h` ассемблер воспримет как ошибку. Поэтому если вы предпочитаете указывать букву `h` после кода, то во второй строке примера 6.7 после слова `fun` надо убрать символы `&h`, которые вызывают добавление буквы `h` к коду функции.

Инициализация драйвера нужна для того, чтобы ликвидировать те изменения значений его внутренних переменных, которые могли оставить после себя другие задачи. Если по каким-то причинам эти изменения надо сохранить, то перед инициализацией производится сохранение буфера состояния драйвера, о чем говорилось в конце предыдущего раздела.

Функция `Mouse 0` выполняет инициализацию драйвера и возвращает дополнительные данные в регистрах `ax` и `bx`.

Если драйвер мыши отсутствует в оперативной памяти, то регистр `ax` очищен. Это может означать либо отсутствие соответствующего файла в `autoexec.bat` или в `config.sys`, либо отсутствие или неисправность мыши. В процессе установки драйвер анализирует наличие и тип мыши, и если работа с ней невозможна, то установка не выполняется.

Если в регистре `ax` находится код `0FFFFh`, то инициализация выполнена успешно. В таком случае в регистре `bx` указано количество имеющихся у мыши кнопок.

После исполнения запроса `Mouse 0` желательно проверить содержимое регистров `ax` и `bx`. Если драйвер отсутствует, то дальнейшее выполнение программы невозможно или для управления задачей должна использоваться клавиатура. Аналогично, если задача рассчитана на работу с тремя кнопками, а у мыши их только две, то придется либо прервать выполнение задачи, либо настроить ее на работу только с двумя кнопками.

Функция `Mouse 21` аналогична функции `Mouse 0`, но при ее исполнении не производится аппаратный сброс мыши и не изменяются значения переменных, зависящих от ее технических характеристик. В большинстве случаев это различие не принципиально.

Пределы перемещения и исходная позиция. При работе в режимах VESA основное назначение драйвера заключается в отслеживании текущей позиции мыши и ее преобразовании в координаты курсора на экране. Для того чтобы это преобразование было корректным, после инициализации драйвера надо установить размер рабочего поля, указав его границы по горизонтали и вертикали.

Функции `Mouse 7` и **`Mouse 8`** передают драйверу предельные значения координат по горизонтали и вертикали. Минимальное значение координат `X` (для `Mouse 7`) или `Y` (для `Mouse 8`) помещаются в регистр `cx`, а максимальное значение — в `dx`. Выходные параметры у обеих функций отсутствуют.

Если рабочее поле занимает весь экран, то минимальные значения обеих координат равны нулю (`Xmin = Ymin = 0`), а максимальные зависят от разрешающей способности видеорежима, поэтому `Xmax = horsize`, а `Ymax = versize`. Например, для режима VESA `101h` `Xmax = 640`, а `Ymax = 480`.

После установки границ рабочего поля задается исходная позиция курсора. Его конкретное расположение может быть произвольным, но обычно курсор помещают в центр экрана.

Функция `Mouse 4` перемещает курсор в заданную позицию. Перед обращением к драйверу в регистры `cx` и `dx` помещаются значения координат `X` и `Y`. Выходные параметры у функции отсутствуют.

Здесь имеется в виду тот курсор, который обычно рисует драйвер мыши. Как уже говорилось, при установке режимов VESA драйвер не может

работать с курсором. Поэтому при выполнении данной функции указанная позиция просто фиксируется в счетчиках драйвера, содержащих текущие координаты.

Для перемещения курсора в центр экрана значения координат составляют `X = horsize/2`, `Y = versize/2`. После установки этих величин можно нарисовать изображение курсора на экране.

Напомним, что при работе в стандартных видеорежимах IBM драйвер автоматически определяет границы рабочего поля и принудительно помещает изображение курсора в центр экрана (если курсор включен).

Новые переменные. При выполнении подготовительных действий надо настроить не только драйвер, но и задачу. В процессе выполнения задачи будет неоднократно анализироваться перемещение мыши и состояние ее кнопок. Для того чтобы анализ был возможен, в разделе данных программы должны быть зарезервированы перечисленные в примере 6.8 переменные.

Пример 6.8. Переменные, используемые при работе с мышью

```
Winpnt   dw 2      ; окно видеопамяти, в котором расположен курсор
Offspnt  dw 22848  ; смещение изображения курсора в этом окне
Xpointer dw 320    ; текущая X координата курсора (номер столбца)
Ypointer dw 240    ; текущая Y координата курсора (номер строки)
Mstatus  db 0      ; текущее состояние манипулятора "мышь"
LBevent  db 0      ; изменение состояния левой кнопки
RBevent  db 0      ; изменение состояния правой кнопки
```

В переменных `Winpnt` и `Offspnt` хранится адрес видеопамяти для левой верхней точки изображения курсора, они уже использовались в примерах 6.4, 6.5 и 6.6 данной главы. Переменные `Xpointer` и `Ypointer` содержат тот же адрес, но представленный в виде номеров строки и столбца. В примере 6.8 указаны их исходные значения для режима VESA `101h` при условии, что курсор находится в центре экрана. Вычисления значений этих четырех переменных производится в примере 6.9.

Три последние переменные примера 6.8 имеют размер байта. В исходном состоянии они должны быть очищены, что и делается при их описании. `Mstatus` содержит данные о текущем состоянии манипулятора, а `LBevent` и `RBevent` — коды конкретного состояния левой (`LB`) и правой (`RB`) кнопок мыши. Как формируются текущие значения этих переменных, показано в примере 6.12.

Выполнение настройки. Способ выполнения всех описанных действий иллюстрирует пример 6.9. Приведенный в нем фрагмент программы должен выполняться в процессе подготовительных действий, но только после установки видеорежима и получения его характеристик.

Пример 6.9. Настройка драйвера, задачи и первый вывод курсора

```
      mouse 0                ; инициализация драйвера
;     !! здесь желательно проверить содержимое регистров AX и BX !!
      xor   cx, cx           ; CX = Xmin = Ymin = 0
      mov   dx, horsize      ; DX = Xmax = horsize
      mouse 7                ; установка границ по горизонтали
      mov   dx, versize      ; DX = Ymax = versize
      mouse 8                ; установка границ по вертикали
      mov   cx, horsize      ; CX = horsize
      shr   cx, 01           ; центр экрана по горизонтали
      shr   dx, 01           ; центр экрана по вертикали
      mouse 4                ; установка значений счетчика драйвера
      mov   Xpointer, cx     ; Xpointer = 0,5 * horsize
      mov   Ypointer, dx     ; Ypointer = 0,5 * versize
      mov   ax, versize      ; AX = versize
      inc   ax               ; AX = versize + 1
      mul   cx               ; DX:AX = (versize + 1) * horsize / 2
;     В режимах direct color результат надо умножить на размер кода точки
      mov   Offspnt, ax      ; сохраняем смещение рисунка курсора
      mov   ax, GrUnit       ; AX = единица измерения размера окна
      mul   dl               ; AX = DL * GrUnit (номер видеоокна)
      add   ax, Base_Win     ; !! учитываем значение базового окна
      mov   Winpnt, ax       ; сохраняем значение окна видеопамяти
      call  Showpnt          ; первое построение рисунка курсора
```

При вычислении адреса видеопамяти по номеру строки и столбца надо учитывать размер кода точки. Пример 6.9 предназначен для выполнения в режимах `PPG`, когда код точки занимает 1 байт. Если ваша задача работает с режимами `direct color`, то результат, вычисленный командой `mul`, надо дополнительно умножить на размер кода точки. Подробнее об этом будет сказано при описании примера 6.13.

Важно

Команда `add ax, Base_win` нужна только в том случае, если задача поддерживает работу со страницами видеопамяти (см. раздел 2.5), в остальных случаях ее надо исключить из текста примера.

В зависимости от способа построения изображения курсора в последней команде примера 6.9 должна вызываться подпрограмма `Showpnt` (см. пример 6.5) или `Tglpntr` (см. пример 6.4).

Чувствительность курсора и мыши. При установке драйвера по умолчанию выбирается режим работы, при котором перемещение мыши на 1 дюйм по горизонтали или вертикали вызывает перемещение курсора на 640 столбцов,

или на 320 строк. Рассмотрим, как драйвер увязывает перемещения мыши и курсора.

Во внутреннем буфере драйвера имеются четыре счетчика, содержащих количество перемещений по вертикали и горизонтали. Два из них связаны с курсором, а два с мышью. Условимся обозначать их как СПК (счетчик перемещений курсора) и СПМ (счетчик перемещений мыши). Прикладные задачи могут изменять значения СПК с помощью команды `Mouse 4`, но значения СПМ они могут только считывать.

Драйвер пересчитывает значения СПМ в значения СПК, используя для этого специальные коэффициенты. При установке драйвера по умолчанию выбираются такие значения коэффициентов, которые вызывают изменение содержимого СПМ и СПК на 1 при каждом перемещении мыши в горизонтальном направлении, и изменение значений СПМ на 1, а СПК — на 2 при каждом перемещении мыши в вертикальном направлении. Прикладная задача или операционная система могут изменить значения коэффициентов.

Функция **`Mouse 1A`** устанавливает, а **`Mouse 1B`** считывает значение коэффициента, задающего чувствительность СПМ к перемещениям мыши. Значения коэффициентов указываются или возвращаются драйвером в регистрах `bx` (горизонтальное направление) и `cx` (вертикальное направление). Содержимое `bx` и `cx` может изменяться от 0 до 100 и интерпретируется как проценты. При задании больших значений они принудительно уменьшаются до 100 (`64h`).

За единицу принято 50% (код `32h`), при котором содержимое СПМ изменяется на 1 при каждом перемещении мыши. Значение 100% вызывает изменение содержимого СПМ на 2 при каждом перемещении мыши. А при коэффициенте 25% оно будет изменяться на 1 при двух перемещениях мыши.

Перемещение мыши принято измерять в `mickey`. Перевод этого термина автору не известен, но, по сути, это величина обратная количеству точек на дюйм (`Dot Per Inch` или `DPI`). У современных манипуляторов `DPI` = 400, соответственно `1 mickey` = 1/400 дюйма или примерно 0,06 миллиметра.

Функция **`Mouse 0F`** устанавливает чувствительность СПК к изменениям СПМ. Перед обращением к драйверу в регистрах `cx` и `dx` указываются значения коэффициентов для горизонтального (`cx`) и вертикального (`dx`) направлений. Эти коэффициенты указывают, на сколько единиц должно измениться значение СПМ для того, чтобы значение СПК изменилось на 8 единиц. При установке драйвера (по умолчанию) коэффициенты равны 8 для горизонтального и 16 для вертикального направлений. В результате при движении по горизонтали СПМ и СПК изменяются синхронно, а при движении по вертикали СПК изменяется в два раза медленнее, чем СПМ.

6.3. Работа в режиме опроса драйвера мыши

С манипулятором "мышь", как с большинством внешних устройств, задача может работать в режиме опроса его текущего состояния, или в режиме прерываний. Принципиальное различие состоит в том, как задача "узнает" об изменении состояния мыши. В первом случае она определяет это самостоятельно, а во втором драйвер "обращает ее внимание" на изменение состояния мыши. Названные режимы обычно дополняют друг друга.

Режим опроса программируется проще, чем режим прерываний, поскольку выполняемые задачей действия не зависят от внешних факторов. Выполнение любого нового действия задача начинает только после завершения предыдущего, что исключает "параллельное" выполнение нескольких разных действий. Именно по этой причине автор выбрал режим опроса для описания способов программирования работы с мышью. Кроме того, приведенные ниже примеры применимы и при работе в режиме прерываний.

В данном разделе описано все, что необходимо для составления завершенной задачи, способной перемещать изображение курсора по экрану и реагировать на нажатие кнопок мыши. Результат можно использовать как основу или как "испытательный полигон" при разработке более сложных и полезных задач и отладке подпрограмм различного назначения.

6.3.1. Управляющий алгоритм для режима опроса

Структура задачи. Прежде всего, давайте уточним некоторые общие вопросы. В структуре задачи, составленной для работы в режиме опроса, можно выделить следующие основные компоненты.

Подготовительные действия. К ним относятся: установка и определение характеристик видеорежима, вычисление значений используемых переменных, резервирование необходимого пространства оперативной памяти, перехват векторов прерываний, настройка драйвера мыши, вывод заставки на экран и пр.

Управляющий алгоритм. В зависимости от конкретных действий оператора инициирует выполнение тех или иных подпрограмм, входящих в состав задачи.

Набор подпрограмм, вызываемых управляющим алгоритмом. Их состав и выполняемые действия зависят от назначения конкретной задачи. Они могут, например, строить рисунки, вводить и выводить текстовые сообщения, перемещать курсор, поддерживать работу с меню и выполнять множество других действий.

Пример управляющего алгоритма. Подготовительные действия и подпрограммы неоднократно обсуждались и еще будут обсуждаться в тексте книги.

Здесь нас интересует управляющий алгоритм, который увязывает разрозненные подпрограммы в единое целое. Он представляет собой бесконечно повторяющийся цикл опроса и анализа текущего состояния клавиатуры и драйвера мыши, и вызова подпрограмм, в зависимости от введенных символов или изменения состояния мыши. Для выхода из цикла предназначена специальная команда, исполнение которой приводит к завершению работы задачи и возврату в DOS. Простейший вариант управляющего алгоритма показан в примере 6.10.

Пример 6.10. Управляющий алгоритм для режима опроса

```
General: mov   ah, 01         ; код функции опроса состояния клавиш
         int   16h            ; опрос состояния клавиш
         jnz   Preskey        ; -> была нажата одна из клавиш
         call  Statms         ; опрос текущего состояния мыши
         xor   bh, bh         ; очистка старшего байта регистра bx
         shl   bx, 01         ; удвоение кода состояния
         call  ChoiceL[bx]    ; обработка состояний левой кнопки
         mov   bl, RBevent    ; bl = код состояния правой кнопки
         xor   bh, bh         ; очистка старшего байта регистра bx
         shl   bx, 01         ; удвоение кода состояния
         call  ChoiceR[bx]    ; обработка состояний правой кнопки
         jmp   short General  ; возврат на начало цикла
Preskey: xor   ah, ah         ; код функции чтения символа
         int   16h            ; чтение введенного символа
         cmp   ah, 31h        ; введена буква N или n ?
         jne   Pk_1           ; -> нет
         lea   si, prmpt01    ; si = адрес подсказки оператору
         call  Outinf         ; вывод подсказки и ввод ответа
         jmp   short General  ; возврат на начало цикла
Pk_1:    cmp   ah, 2Dh        ; введена буква X или x ?
         jne   Pk_2           ; -> нет
         jmp   eoprg          ; переход на завершение задачи
Pk_2:    mov   bx, 01         ; bx = 1 (шаг перемещения курсора)
         cmp   ah, 4Dh        ; символ "стрелка вправо" ?
         jne   Pk_3           ; -> нет
movhor:  call  mothor         ; перемещение курсора по горизонтали
         jmp   short General  ; возврат на начало цикла
Pk_3:    cmp   ah, 50h        ; символ "стрелка вниз" ?
         jne   Pk_4           ; -> нет
movver:  call  motver         ; перемещение курсора по вертикали
         jmp   short General  ; возврат на начало цикла
Pk_4:    neg   bx             ; bx = -1 (шаг перемещения курсора)
         cmp   ah, 4Bh        ; символ "стрелка влево" ?
         je    movhor         ; -> да
```

```
        cmp   ah, 48h       ; символ "стрелка вверх" ?
        je    movver        ; -> да
;       Здесь можно продолжить анализ введенного символа
        jmp   General       ; !! возврат на начало цикла
```

Текст примера 6.10 делится на две основные части. Первая из них начинается с команды, имеющей метку `General`, а вторая — с команды, имеющей метку `Preskey`. Первая часть алгоритма выполняется до тех пор, пока оператор не нажмет на любую клавишу. В этой части производится опрос состояния клавиатуры и драйвера мыши и обработка событий, связанных с изменением состояния мыши. Мы не будем здесь обсуждать, как это делается, поскольку способы опроса и обработки возможных состояний манипулятора "мышь" подробно обсуждаются в следующем разделе. Пока читатель может поверить на́ слово, что если оператор ничего не делает с клавиатурой и мышью, то задача только опрашивает состояние клавиатуры и мыши, не выполняя никаких других действий.

Роль клавиатуры зависит от формы диалога оператора с задачей. В тех случаях, когда применяются командные строки и не поддерживается работа с меню, клавиатура является основным органом управления. Если же задача поддерживает работу с меню, то клавиатура имеет вспомогательное значение и используется только в специальных случаях, например, для ввода текста или спецификаций создаваемых файлов. Однако и в этих случаях клавиатура не является основным средством для ввода данных.

Техника работы с клавиатурой обсуждалась в разделе 5.2.5, там же описана функция `0` прерывания `int 16h`, выполняющая ожидание ввода и чтение кода символа из буфера клавиатуры. В данном случае нам нужна еще одна функция `01` прерывания `int 16`, которая не ждет ввода символа, а только проверяет состояние буфера клавиатуры. Если буфер пуст, то при возврате из `BIOS` установлен Z-разряд регистра флагов (признак нуля), а если в буфере находится код символа, то Z-разряд будет очищен.

В примере 6.10 третья команда (`jnz Preskey`) выполнит переход на метку `Preskey`, если буфер клавиатуры содержит код символа. Его копия находится в регистре `ax`, но прежде чем начинать анализ введенного символа, надо учесть следующее обстоятельство. Функция `01` прерывания `int 16h` оставляет символ в буфере и при следующем опросе клавиатуры он будет прочитан повторно. Поэтому буфер надо принудительно очистить, что и делают первые две команды, расположенные во второй части примера 6.10 (после метки `Preskey`).

Расшифровка и исполнение команд. После чтения введенного символа в регистре `ah` находится `scan code`, а в регистре `al` — код `ASCII`, если таковой существует. Обычно для анализа используется `scan code`, но если надо различать символы верхнего и нижнего (например, `A` и `a`), или латинского и рус-

ского (например, A и Ф) регистров, то используется код ASCII. В примере 6.10 анализируется scan code шести разных символов. Вы можете изменить или дополнить набор анализируемых кодов, только не забывайте, что независимо от результата их анализа необходимо вернуться на начало управляющего алгоритма. Для напоминания последняя команда примера выполняет такой переход, а комментарий к ней начинается с двух восклицательных знаков.

Ввод спецификации файла. В примере 6.10 при опознании кода буквы N или n выводится подсказка оператору и вводится спецификация файла. Текст подсказки должен быть описан в разделе данных задачи, например:

```
prmpt01 db 'Введите спецификацию файла >', 0
```

Ее адрес загружается в регистр si, после чего вызывается подпрограмма OutInf, описанная в примере 5.29 (см. раздел 5.2.5). Введенная спецификация файла находится в буфере Linbuf.

Ввод спецификации это только начало, для работы с файлом его надо открыть, а способ открытия зависит от того, как будет использоваться файл — только для чтения, только для записи или для того и другого. Для открытия файла можно использовать специальную команду (букву) или объединить ввод спецификации и открытие файла в одной подпрограмме. Это удобно потому, что только при открытии проверяется правильность введенной спецификации (существование указанного файла). Для дальнейших манипуляций с файлом так же понадобятся дополнительные команды (буквы), специальные подпрограммы или то и другое. Решение подобных вопросов мы оставляем на усмотрение читателя.

Завершение задачи. Для завершения выполнения задачи и возврата в DOS оператор должен ввести букву X или x. Следует отметить, что для этой цели она используется во многих прикладных задачах для DOS и соответствует команде eXit. Иногда для той же цели используется одновременное нажатие (сочетание) клавиш <Alt> и <X> (<Alt>+<X>). Для опознания такого сочетания в примере 6.10 команду cmp ah, 2Dh надо заменить командой cmp ax, 2D00h.

Расшифровав код 2Dh, управляющий алгоритм выполняет переход на метку eoprg (end of programm), начиная с которой выполняются заключительные действия. Они заканчиваются двумя командами, приведенными в примере 6.11.

Пример 6.11. Завершение работы задачи и выход в DOS

```
eoprg:  ; Сначала выполняются заключительные действия, а потом:
        mov   ah, 4Ch     ; код запроса "завершение задачи"
        int   21h         ; обращение к DOS без возврата в задачу
```

После выполнения последних двух команд примера 6.11 DOS снимает задачу и освобождает занимаемую ей оперативную память общего назначения. Перед выходом надо ликвидировать все внесенные задачей изменения, которые могут нарушить нормальную работу DOS или других задач. В частности, если задача перехватывала векторы прерываний, то надо восстановить их исходные значения. Например, для восстановления исходного состояния вектора `1Ch` после метки `eoprg` надо вставить текст примера 5.26. Если для работы с расширенной памятью использовался драйвер `EMM`, то надо обратиться к нему для освобождения выделенной памяти. Работа с драйвером `EMM` будет описана в приложении Б.

Использование стрелок. В примере 6.10 показано, как можно управлять перемещениями указателя мыши с помощью клавиш, на которых нарисованы стрелки, направленные влево, вправо, вверх и вниз. В зависимости от того, какая клавиша нажата, выбирается одна из двух подпрограмм — `mothor` для перемещения указателя мыши по горизонтали или `motver` для его перемещения по вертикали. Предварительно в регистр `bx` записывается положительная (перемещение влево или вниз) или отрицательная (перемещение вправо или вверх) единица. Текст обеих подпрограмм приведен в примере 6.13.

Таким образом, в описанном варианте управляющего алгоритма клавиатура используется для управления процессом выполнения задачи. Теперь нам предстоит разобраться в том, какую роль играет мышь.

6.3.2. Формирование кодов событий

В примере 6.10 четвертая команда вызывает подпрограмму `Statms`, текст которой приведен в примере 6.12. Она формирует и передает задаче данные о событиях, связанных с манипулятором "мышь". Выполняемые в ней действия оформлены в виде подпрограммы, для того чтобы их можно было использовать не только в управляющем алгоритме, но и в других случаях, когда нужны данные о состоянии мыши.

Прежде чем описывать подпрограмму `Statms`, обсудим те соображения, которыми руководствовался автор при ее разработке.

Функции драйвера. Драйвер поддерживает три функции, которые позволяют получить разные данные о состоянии мыши.

Функция `Mouse 3` возвращает в регистрах `bx`, `cx` и `dx` текущие значения счетчиков координат на экране и состояние кнопок. В `cx` находится номер столбца (координата `X`), а в `dx` — номер строки (координата `Y`). Три младших разряда регистра `bx` отражают состояние кнопок. Левой кнопке соответствует нулевой разряд, правой — первый и средней — второй. Если кнопка нажата, то соответствующий ей разряд установлен, а если не нажата, то очи-

щен. Некоторые драйверы позволяют в процессе установки переопределить правую и левую кнопки, это предусмотрено специально для людей, которым удобнее работать левой рукой. В таком случае нулевой разряд регистра bx отражает состояние правой кнопки, а первый разряд — левой.

Функция Mouse 5 возвращает данные о количестве нажатий на одну из кнопок и значение координат в момент последнего нажатия. Перед ее вызовом в регистре bx указывается номер кнопки: bx=0 для левой, bx=1 для правой и bx=2 для средней. В том же регистре (bx) драйвер возвращает количество нажатий на указанную кнопку, произошедших после последнего опроса ее состояния. Кроме того, в регистре ax возвращается состояние всех кнопок в том же виде, в каком эти данные возвращала функция Mouse 3 в регистре bx. При этом в регистрах cx и dx находятся значения координат в момент последнего нажатия на указанную кнопку.

Функция Mouse 6 отличается от Mouse 5 только тем, что возвращает информацию не о нажатии, а об отпускании указанной кнопки.

Функция Mouse 3 применяется наиболее часто. Функции Mouse 5 и Mouse 6 нужны в специальных случаях и, вообще говоря, при программировании работы с мышью без них можно обойтись.

Взаимосвязь событий. В технической документации любые изменения в состоянии мыши принято называть событиями (event). Функция Mouse 3 возвращает данные об элементарных событиях, в общем случае этого не достаточно для выполнения задачей конкретных действий.

Предположим, что в результате опроса драйвера установлен факт нажатия левой кнопки. Сам по себе этот факт мало что говорит, важно знать, изменилось ее состояние или нет. Если кнопка уже была нажата, то ее состояние не изменилось. Аналогичные рассуждения применимы и в случае, если кнопка не нажата. Следовательно, для оценки изменения состояния кнопки надо учитывать результаты предыдущего и текущего опросов драйвера.

Кроме того, кнопка могла быть нажата при неподвижной мыши или при ее перемещении. На практике движение мыши при нажатой левой кнопке обычно используется для перемещения рисунка на экране. А нажатие на левую кнопку при отсутствии перемещения мыши может применяться, например, для выбора элемента меню. Следовательно, при анализе произошедшего события надо учитывать не только текущее и предшествующее состояние кнопок, но и фактор движения мыши.

Каждый из перечисленных факторов может принимать одно из двух взаимоисключающих значений — кнопка нажата или не нажата, мышь движется или неподвижна. Всего возможны восемь событий, перечисленных в табл. 6.2.

В табл. 6.2 перечисленны элементарные события. Из них могут складываться более сложные события, например, во многих случаях применяется

быстрое двухкратное нажатие (double-click) на кнопку. Его можно описать как повторное нажатие на одну и ту же кнопку в течение короткого отрезка времени при отсутствии перемещения мыши. Для распознания такого события задача должна спустя заданное время повторно опросить драйвер и убедиться в том, что в обоих случаях был получен код 2.

Таблица 6.2. *Перечень событий для одной кнопки мыши*

Код события	Движение мыши	Старое состояние	Новое состояние
0	Неподвижна	Не нажата	Не нажата
1	Неподвижна	Не нажата	Нажата
2	Неподвижна	Нажата	Не нажата
3	Неподвижна	Нажата	Нажата
4	Движется	Не нажата	Не нажата
5	Движется	Не нажата	Нажата
6	Движется	Нажата	Не нажата
7	Движется	Нажата	Нажата

Вопрос о том, состояние каких кнопок надо анализировать в задаче, решает программист. На практике основной является левая кнопка, с ней ассоциируется большинство выполняемых действий. Правая используется реже и имеет вспомогательное значение. Одновременное нажатие обеих кнопок обычно не применяется. Средней кнопки у мыши может просто не быть, поэтому она не используется в большинстве программ.

Подпрограмма Statms опрашивает состояние мыши с помощью функции Mouse 3 и формирует коды событий в соответствии с табл. 6.2.

В качестве параметров подпрограмма, приведенная в примере 6.12, использует переменные, описанные в примере 6.8. Значения входных параметров содержат переменные XPointer, YPointer и Mstatus. Выходные параметры помещаются в те же переменные, кроме того, код события для правой кнопки возвращается в переменной RBevent, а для левой — в LBevent.

Пример 6.12. Формирование кодов событий для двух кнопок

```
Statms: Mouse 3              ; опрос текущего состояния мыши
        xor   al, al         ; признак отсутствия движения
        cmp   XPointer, cx   ; координата X изменилась ?
        jne   SM_1           ; -> да
        cmp   YPointer, dx   ; координата Y изменилась ?
        je    SM_2           ; -> нет, мышь не перемещалась
```

```
SM_1:   or    al, 04          ; признак перемещения мыши
        mov   XPointer, cx    ; сохраняем новое значение X
        mov   Ypointer, dx    ; сохраняем новое значение Y
SM_2:   mov   bh, bl          ; bh = новое состояние кнопок
        xchg  Mstatus, bh     ; переставляем байты bl и Mstatus
        push  bx              ; сохраняем регистр bx
        ; Формирование кода события для правой кнопки
        and   bx, 0202h       ; выделяем разряды состояния кнопки
        shr   bl, 01          ; изменяем код нового состояния
        or    bl, bh          ; двумя командами or формируем
        or    bl, al          ; в bl код события для правой кнопки
        mov   RBevent, bl     ; сохраняем код события в RBevent
        ; Формирование кода события для левой кнопки
        pop   bx              ; восстанавливаем регистр bx
        and   bx, 0101        ; выделяем разряды состояния кнопки
        shl   bh, 01          ; изменяем код старого состояния
        or    bl, bh          ; и двумя командами or формируем
        or    bl, al          ; в bl код события для левой кнопки
        mov   LBevent, bl     ; сохраняем код события в LBevent
eosub:  ret                   ; возврат из подпрограммы
```

После всего сказанного текст примера 6.12 не требует особых пояснений. Напомним только, что код состояния формируется в соответствии с табл. 6.2, его значение может изменяться от 0 до 7. Нулевой разряд кода соответствует текущему состоянию кнопки, первый — предыдущему состоянию, а второй разряд указывает перемещение мыши.

Метка `eosub`, указанная перед командой `ret`, не имеет отношения к тексту примера. Просто в дальнейшем нам понадобится имя подпрограммы, состоящей из единственной команды `ret`.

Выбор исполняющей подпрограммы. Вызывающий модуль анализирует полученный код события и выполняет соответствующие действия. В нашем случае вызывающим модулем является управляющий алгоритм.

В управляющем алгоритме выбор подпрограммы, выполняющей нужные действия, осуществляет переключатель (`switch`). Он применяется во многих языках программирования для выбора одного из нескольких вариантов выполняемых действий. В этом случае код события используется в качестве индекса при выборе одного из адресов, указанных в таблице переходов (`transfer table`), которая является списком адресов подпрограмм. Преимущество переключателя заключается в том, что количество действий, необходимых для выбора нужного адреса, не зависит от размера списка.

В управляющем алгоритме переключатель используется дважды — сначала для обработки событий, связанных с левой, а затем с правой кнопкой. В обоих случаях код события помещается в регистр `bx` и удваивается, по-

скольку списки адресов состоят из слов. После этого команда `call` вызывает одну из подпрограмм перечисленных в списках `ChoiceL` или `ChoiceR`. При выполнении этой команды адрес начала списка (значение меток `ChoiceL` или `ChoiceR`) суммируется с содержимым регистра `bx`, в стеке формируется адрес возврата и управление передается нужной подпрограмме.

Обратите внимание на то, что сразу после возвращения из подпрограммы `Statms` код состояния левой кнопки находится в регистре `bl`. Поэтому при обработке событий левой кнопки нет необходимости копировать его в регистр `bl` из `LBevent`. Однако при обработке событий правой кнопки в регистр `bl` копируется содержимое `RBevent`.

Списки имен подпрограмм, обрабатывающих события, связанные с левой и правой кнопками, должны быть описаны в разделе данных программы, например, следующим способом:

```
ChoiceL dw eosub, name1, name2, eosub, motion, name3, name4, name5
ChoiceR dw eosub, eoprg, name6, eosub, eosub, name7, name8, name9
```

Действия, выполняемые при возникновении каждого из событий, зависят от назначения программы и фантазии программиста. Заведомо очевидны лишь два случая.

Если мышь неподвижна и состояние кнопок не изменилось (коды 0 и 3), то просто ничего не произошло. Этим кодам в обоих списках соответствует имя `eosub`, которое в примере 6.12 указано перед командой `ret`.

Если мышь движется и кнопки не нажаты (код 4), то надо просто перемещать курсор. Поэтому на пятом месте в списке `ChoiceL` указано имя подпрограммы `motion`, выполняющей перемещение курсора, ее подробное описание приведено в следующем разделе.

Во втором слове массива `ChoiceR` указано имя команды `eoprg`, начиная с которой выполняются завершающие действия (см. пример 6.11). При нажатии на правую кнопку мыши выполнение задачи прекратится и произойдет возврат в DOS.

Если при подготовке исходного текста программы вы еще не решили, как обрабатывать остальные события, то просто замените имена `name1` – `name9` именем `eosub`. Управляющий алгоритм будет игнорировать события, для обработки которых не указаны специальные подпрограммы.

Идентификация графических объектов. Кнопки манипулятора "мышь" используются для управления процессом выполнения задач. Выбор выполняемых действий зависит от объекта, на который указывает курсор. Например, если курсор указывает на элемент меню, то при однократном нажатии на кнопку (обычно левую) этот элемент становится активным и выполняются связанные с ним действия. Нажатие на кнопку во время движения мыши может вызывать перемещение или изменение размеров объекта, на который

указывает курсор. При работе в режиме редактирования в этом случае могут изменяться размеры и форма создаваемого объекта.

Перечень действий, выполняемых при нажатии на кнопку мыши, можно продолжать долго. Главное, на что следует обратить внимание, заключается в том, что процедуры, реагирующие на изменение состояния кнопок, как правило, должны идентифицировать расположенный под курсором объект. А как уже говорилось в разделе 3.3.4, для этого задача должна формировать список графических объектов, находящихся на экране в данный момент времени и поддерживать работу с этим списком.

Замечание

Описание процедур, реагирующих на изменение состояния кнопок, выходит за рамки данной книги, поэтому мы ограничимся следующим советом. Начните с составления простой подпрограммы, которая при нажатии левой кнопки проверяет нахождение курсора в прямоугольной области с заданными границами `Xmin`, `Xmax`, `Ymin`, `Ymax`. Если курсор находится в ней, то подпрограмма завершает выполнение задачи. На экране этим координатам может соответствовать прямоугольник с надписью "Выход" или "Exit". Затем усложните подпрограмму сделав так, чтобы значения указанных границ выбирались из формируемого задачей списка. Следующий шаг — при нахождении объекта в списке выполняется подпрограмма, адрес которой хранится в том же элементе, в котором указаны границы объекта. Так постепенно вы создадите универсальную процедуру для обработки событий, связанных с левой кнопкой.

6.3.3. Управление перемещением курсора

Необходимость перемещения курсора возникает при обработке тех событий, которым в табл. 6.2 соответствуют коды от 4 до 7. Перемещение в чистом виде вызывает только событие с кодом 4 — мышь движется, кнопка не нажата и ее состояние не изменялось. Если же мышь движется при нажатой кнопке, то кроме перемещения курсора могут выполняться и другие действия, например "перетаскивание" объекта, на который указывает курсор.

Предварительные замечания. В зависимости от того, какая из кнопок является ведущей (обычно левая), имя подпрограммы, выполняющей перемещение курсора, располагается на пятом месте одного из списков `ChoiceL` или `ChoiceR`. Указывать это имя на пятом месте обоих списков не имеет смысла. При обработке событий с кодами от 5 до 7 такой проблемы не возникает, поскольку предполагается, что одновременное нажатие обеих кнопок в задачах не используется. Подпрограмма `Statms` не фиксирует этот случай, поскольку состояния кнопок анализируются независимо друг от друга.

Для перемещения изображения курсора надо выполнить следующие действия в такой последовательности: восстановить исходный фон на месте старого изображения, вычислить адрес видеопамяти, соответствующий новому значению координат, и вывести изображение курсора на новом месте (одновременно сохранив исходный фон).

Перечисленные действия выполняет подпрограмма `Motion`, текст которой приведен в примере 6.13. Кроме нее в текст примера включены еще две подпрограммы, обращение к которым происходит из управляющего алгоритма при нажатии оператором на клавиши с рисунками стрелок, направленных влево, вправо, вверх и вниз. Подпрограмма `Mothor` перемещает курсор на шаг вправо или влево, а `Motver` — вверх или вниз. Шаг и направление перемещения задается в регистре `bx`, для перемещения в сторону уменьшения значений координат его содержимое должно быть отрицательным числом. Подпрограммы `Mothor` и `Motver` являются вспомогательными, основные действия выполняет `Motion`.

Восстановление исходного фона и построение изображения курсора было описано в разделах 6.1.3 и 6.1.4 данной главы. Напомним только, что если вы предпочитаете работать с немаскируемым курсором, то вместо подпрограмм `Hidepnt` и `Showpnt` надо использовать `Tglpntr`. Способ пересчета значений координат в адрес видеопамяти уже неоднократно обсуждался, остается только применить его в данном конкретном случае.

Мышь не всегда перемещается плавно, поэтому при вычислении адреса видеопамяти не следует исходить из предположения, что курсор перемещается только в одну из смежных точек, показанных в табл. 3.3. Текущие значения координат хранятся в переменных `Xpointer` (номер столбца) и `Ypointer` (номер строки), которые описаны в примере 6.8. Для вычисления адреса номер строки умножается на размер экрана по горизонтали (`horsize`), к произведению прибавляется номер столбца и результат корректируется с учетом `GrUnit`. При этом предполагается, что код точки занимает 1 байт, т. е. установлен один из режимов `PPG`.

Подпрограммы перемещения курсора. Таковы исходные предпосылки и теперь можно обсудить особенности конкретной реализации подпрограммы `Motion`. Ее текст приведен в примере 6.13.

Пример 6.13. Группа подпрограмм для перемещения курсора

```
Mothor:  add   Xpointer, bx   ; изменение значения Xpointer
         jmp   short setpos   ; обход следующей команды
Motver:  add   Ypointer, bx   ; изменение значения Ypointer
setpos:  mov   cx, Xpointer   ; cx = Xpointer
         mov   dx, Ypointer   ; dx = Ypointer
         Mouse 4              ; установка новых значений счетчиков
;   Перемещение изображения указателя мыши на экране
Motion:  call  Hidepnt        ; гасим курсор
         mov   ax, horsize    ; помещаем в ax ширину экрана
         mul   Ypointer       ; умножаем ax на номер строки
         add   ax, Xpointer   ; прибавляем номер столбца
         adc   dx, 00         ; учитываем возможность переполнения
```

```
;   В режимах direct color результат надо умножить на размер кода точки
        mov   Offspnt, ax   ; сохраняем смещение в Offspnt
        mov   ax, GrUnit    ; помещаем в ax значение GrUnit
        mul   dl            ; вычисляем номер окна
        add   ax, Base_win  ; !! учитываем значение базового окна
        mov   Winpnt, ax    ; сохраняем в Winpnt
        call  Showpnt       ; рисуем изображение курсора
        ret                 ; возврат из подпрограммы
```

Подпрограммы примера 6.13 рассчитаны на выполнение в режимах `PPG`. Если вы будете использовать их в режимах `direct color`, то результат полученный после выполнения команды `adc dx, 00` надо умножить на размер кода точки. В тексте примера об этом напоминает комментарий.

В видеорежимах `Hi-Color` код точки занимает 2 байта. Для умножения на 2 после команды `adc dx, 00` надо вставить следующие команды:

```
shld      dx, ax, 1    ; сдвиг dx с добавлением старшего разряда ax
shl       ax, 1        ; сдвиг содержимого ax на разряд влево
```

Первая команда сдвигает содержимое регистров `dx:ax` как одно двойное слово, но изменяет только содержимое `dx`, в который записывается старшая часть результата. Поэтому для сдвига содержимого регистра `ax` нужна дополнительная команда.

В видеорежимах `True Color` код точки занимает 4 байта. Для умножения на 4 в командах сдвига надо заменить 1 на 2.

```
shld      dx, ax, 2      ; сдвиг dx с добавлением старших разрядов ax
shl       ax, 2          ; сдвиг содержимого ax на два разряда влево
```

Если код точки занимает три разряда (см. раздел 7.1.3), то вместо сдвигов придется использовать умножение на 3.

Можно составить такой вариант программы, который учитывает размер кода точки при выполнении сдвига. Этот размер формируется в процессе выполнения подготовительных действий и хранится в специально выделенной переменной `wrdppnt` (см. раздел 7.2). Подпрограмма перемещения может работать либо с этой переменной, либо с дополнительным параметром, содержащим величину сдвига.

Важно

Команда `add ax, Base_win` нужна, только если задача поддерживает работу со страницами видеопамяти (см. раздел 2.5), в остальных случаях ее надо исключить из текста примера.

Дополнительная точка входа. В описанной подпрограмме объединены восстановление исходного фона, вычисление адреса видеопамяти и построение

изображения курсора. Иногда эти действия должны выполняться независимо друг от друга. Например, при перемещении рисунка, на который указывает курсор, сначала восстанавливается исходный фон, затем рисунок перемещается вслед за мышью и только после остановки вычисляется адрес видеопамяти и выводится изображение курсора на новом месте. Специально для подобных случаев можно предусмотреть вторую точку входа в подпрограмму `Motion`. Имя дополнительной точки входа указывается перед командой `mov ax, horsize`. При входе через эту точку исключается удаление изображения курсора с экрана

Цель, сформулированная во введении к данному разделу, достигнута. Мы описали управляющий алгоритм и набор подпрограмм, необходимых для составления простейшей задачи, выполняющей перемещение курсора по экрану в режиме опроса. Теперь можно перейти к рассмотрению более гибкого управления перемещениями курсора.

6.4. Работа в режиме прерываний

Недостаток режима опроса заключается в том, что задача не узнает об изменении состояния мыши до тех пор, пока не обратится к драйверу. В некоторых случаях этот недостаток имеет принципиальное значение, и программист вынужден использовать режим прерываний.

В данном разделе нас будут интересовать те прерывания процесса выполнения задачи, которые вызывает драйвер при изменениях состояния манипулятора "мышь". Задача может разрешить или запретить драйверу прерывать процесс своего выполнения в указанных случаях. Если прерывания разрешены, то момент их возникновения зависит только от действий оператора, работающего с мышью, и никак не связан с действиями, выполняемыми задачей. То есть, как обычно, прерывания происходят по внешним, не зависящим от задачи, причинам.

Указанная особенность режима прерываний требует от программиста определенных навыков и тщательной разработки алгоритма задачи. Кроме внутренних факторов, влияющих на выполнение предусмотренных в задаче действий, приходится учитывать и внешние, а это может существенно изменить конкретную реализацию алгоритма. Мы обсудим этот вопрос на примере работы с курсором в режиме прерываний.

Состояние мыши изменяется не так уж часто, даже если она активно используется оператором. При программировании для режима прерываний, рано или поздно, но вам придется решать, чем занять задачу в паузах между изменениями состояния мыши. Если нет другого занятия, то организуется цикл ожидания действий оператора, а для этого предназначен управляющий алгоритм, описанный в разделе 6.3.1. Таким образом, режимы прерываний и опроса не исключают, а дополняют друг друга.

6.4.1. Функции драйвера

Для того чтобы при изменении состояния мыши драйвер мог прервать выполнение задачи, последняя должна с помощью специальных функций установить прерывающую подпрограмму. В данном разделе описаны варианты установки и удаления прерывающих подпрограмм, требования к их оформлению, условия вызова и данные, передаваемые драйвером.

Обычно термин "прерывающая" указывает на то, что подпрограмма вызывается при возникновении событий, независящих от выполнения основной задачи полностью или частично. В нашем случае момент вызова подпрограммы зависит от действий оператора, работающего с мышью.

Особенность описываемых ниже подпрограмм заключается в том, что в случае необходимости их может вызывать основная задача, но это уже относится к области трюков или искусства программирования.

Установка основной подпрограммы. Драйвер может обслуживать только одну основную прерывающую подпрограмму. Для ее установки задаются адрес точки входа и маска события (или событий), при каждом наступлении которого (которых) будет вызываться установленная подпрограмма.

Функция `0C` (`Set Event Handler`) предназначена для установки подпрограммы, реагирующей на события, связанные с изменением состояния мыши. Перед обращением к драйверу полный адрес точки входа указывается в регистрах `es:dx`, а код маски помещается в младший байт регистра `cx`. Состояние его разрядов (0 или 1) соответствует наличию или отсутствию следующих событий:

разряд 0 — перемещение мыши;

разряд 1 — левая кнопка нажата; разряд 2 — левая кнопка отпущена;

разряд 3 — правая кнопка нажата; разряд 4 — правая кнопка отпущена;

разряд 5 — средняя кнопка нажата; разряд 6 — средняя кнопка отпущена.

Допустимо произвольное сочетание указанных признаков, в частности, при записи в регистр `cx` кода `7Fh` подпрограмма будет вызываться при любых изменениях состояния мыши.

Особым случаем является очищенное состояние регистра `cx` при обращении к драйверу, оно запрещает обслуживание ранее установленной подпрограммы. При изменении состояния мыши драйвер проверяет, указан код произошедшего события в маске или нет. Ни одно из событий не имеет код 0, поэтому при очищенной маске вызов подпрограммы исключается. Для очистки маски достаточно выполнить следующие две команды:

```
xor         cx, cx      ; очистка регистра cx
Mouse       0C          ; запрос функции 0C
```

Если ваша задача устанавливала прерывающую подпрограмму, то не забудьте выполнить эти две команды перед возвратом в DOS. В противном случае при первом же наступлении соответствующего события драйвер обратится к тому участку памяти, в котором подпрограммы уже нет, и возникнет аварийная ситуация.

При исполнении функции `0C` драйвер просто копирует содержимое регистров `cx`, `dx` и `es` в три слова внутреннего буфера, не выполняя никаких проверок. Таким простым способом исключается возможность установки нескольких подпрограмм. В любой момент времени драйвер обслуживает только ту подпрограмму, которая была установлена последней и "обмануть" его невозможно.

Если задача должна реагировать на несколько разных событий, то в маске надо установить соответствующие разряды, а в прерывающей подпрограмме уточнять причину (или причины) прерывания. Другими словами, иногда приходится сочетать режимы прерываний и опроса состояния драйвера мыши.

Функция `14` (`Exchange Event Handler`) выполняет те же действия, что и функция `0C`, кроме того, драйвер возвращает в регистрах `es:dx` адрес ранее установленной подпрограммы, а в регистре `cx` маску событий, на которые она реагировала. Эта функция может быть полезна в тех случаях, когда по каким-то причинам надо заменить прерывающую подпрограмму, а через некоторое время восстановить ее работоспособность.

Установка альтернативных подпрограмм. Учитывая, что основная подпрограмма может быть только одна, разработчики предусмотрели возможность установки трех альтернативных подпрограмм. Их принципиальное отличие в том, что в момент изменения состояния мыши должна быть нажата хотя бы одна из трех специальных клавиш.

Функция `18` (`Set Alternate Event Handler`) устанавливает альтернативную подпрограмму, реагирующую на изменения состояний мыши *при условии*, что нажата одна из трех клавиш — <Alt>, <Ctrl>, <Shift>, или любая их комбинация. Перед обращением к драйверу полный адрес подпрограммы указывается в регистрах `es:dx`, а код маски помещается в младший байт регистра `cx`. Состояние его разрядов (0 или 1) соответствует наличию или отсутствию следующих событий:

разряд 0 — перемещение мыши; разряд 1 — левая кнопка нажата;

разряд 2 — левая кнопка отпущена; разряд 3 — правая кнопка нажата;

разряд 4 — правая кнопка отпущена; разряд 5 — нажата клавиша <Shift>;

разряд 6 — нажата клавиша <Ctrl>; разряд 7 — нажата клавиша <Alt>.

Три старших разряда маски отведены для указания одной из клавиш или их сочетания. Драйвер анализирует их не независимо друг от друга, а как трехразрядный код. Поэтому установка любых двух старших разрядов соответст-

вует одновременному нажатию двух клавиш, а установка трех разрядов — одновременному нажатию трех клавиш <Alt>+<Ctrl>+<Shift>.

Указание в коде маски, по крайней мере, одной клавиши обязательно. Если очищены все три старших разряда кода маски, то драйвер отвергает попытку установить подпрограмму. При этом он возвращает в регистре `ax` код `0FFFFh`.

Коды событий, связанных с изменением состояния мыши, занимают в маске пять младших разрядов, места для средней кнопки не хватает. Если установлены все младшие разряды, то подпрограмма будет вызываться при любом изменении состояния мыши, если одновременно нажата клавиша, указанная в старших разрядах кода маски. Например, если код маски `9Fh`, то подпрограмма будет вызываться, если нажата клавиша <Alt> и мышь перемещается, либо изменяется состояние ее левой или правой кнопок.

Особым случаем является очищенное состояние пяти младших разрядов кода маски, оно запрещает обслуживание установленной альтернативной подпрограммы. Обращаем ваше внимание на то, что должны быть очищены *только пять* младших разрядов. Например, если подпрограмма работала с клавишей <Alt>, то для ее запрета выполняются две следующие команды:

```
mov   cx, 80h  ; код маски при работе с клавишей <Alt>
Mouse 18       ; запрос функции 18
```

Перед завершением задачи надо обязательно запретить обслуживание как основной, так и альтернативных подпрограмм, если они были установлены.

Функция `19` (`Query Alternate Event Handler`) проверяет факт установки альтернативной подпрограммы, маска для поиска указывается в регистре `cx`. Если подпрограмма была установлена, то драйвер возвращает ее адрес в регистрах `es:dx`, в противном случае он помещает в регистр `ax` код `0FFFFh`.

Ошибка в драйвере Mitsumi. Автор исследовал три драйвера мыши, разработанные фирмами Microsoft, Genius и Mitsumi. В последнем из них допущена трудно диагностируемая ошибка, она заключается в следующем. Если основная и альтернативная подпрограммы реагируют на одно и то же изменение состояния мыши, то драйвер отдает предпочтение основной и не вызывает альтернативную подпрограмму. В частности, если основная подпрограмма реагирует на любые изменения состояния мыши (код маски `7Fh`), то альтернативные подпрограммы не будут вызваны драйвером ни при каких условиях. На этикетке установочной дискеты, прилагаемой к мыши, написано MITSUMI Mouse Driver Version 6.0.

Вызов подпрограммы драйвером. Если произошло событие, код которого указан в маске, то драйвер вызывает установленную подпрограмму. При входе в нее в регистрах `bx`, `cx` и `dx` находятся данные о состоянии кнопок и значениях координат, представленные в том виде, в котором они получаются после выполнения функции 3.

Три младших разряда регистра bx указывают состояние кнопок. Если разряд установлен, то соответствующая ему кнопка нажата, а если очищен, то не нажата. Разряды 0, 1, 2 соответствуют левой, правой и средней кнопкам.

Дополнительно в регистре ax находится код события, явившегося причиной вызова подпрограммы. В этом регистре может быть установлен только один из указанных в коде маски разрядов (см. описание функции 0C). Например, если подпрограмма вызывается при любом изменении состояния мыши (код маски 7Fh), то будет установлен один из 7 младших разрядов регистра ax.

Предположим, что оператор перемещает мышь при нажатой левой кнопке. В момент начала перемещения произойдут два вызова подпрограммы. В одном случае в регистре ax будет находиться код 1, а во втором 2. Какой из двух вызовов произойдет первым, зависит от того, что раньше сделал оператор — нажал левую кнопку или начал двигать мышь. В дальнейшем состояние левой кнопки не меняется до того момента, пока она не будет отпущена, поэтому при вызовах подпрограммы в регистре ax будет находиться код 1. При отпускании левой кнопки в регистре ax окажется код 4.

Если подпрограмма реагирует только на перемещение курсора (код маски 1), то состояние кнопок указывает код, находящийся в регистре bx.

В конце раздела 6.2.2 говорилось о том, что драйвер поддерживает счетчики перемещений курсора (СПК) и мыши (СПМ), причем значения СПК он вычисляет по значениям СПМ. Обычно в прикладных задачах используется СПК, его значения возвращает функция Mouse 3 в регистрах cx и dx. В отличие от этой функции, при входе в прерывающую подпрограмму доступны оба типа счетчиков.

Значения СПК находятся в регистрах cx и dx, первый из них содержит текущее значение номера столбца, а второй — номера строки.

Значения СПМ находятся в регистрах si и di, первый из них содержит количество перемещений мыши по горизонтали, а второй — по вертикали. Оба счетчика очищаются при инициализации драйвера (функции 0 и 21) и при выполнении специальной функции 0B. Текущие значения счетчиков могут быть отрицательными или положительными числами. Положительные значения соответствуют перемещениям мыши вправо и вниз, а отрицательные — влево и вверх. Обычно в прикладных задачах значения СПМ не используются.

Общие требования к подпрограммам. При входе в подпрограмму регистр ds содержит сегмент оперативной памяти, в котором расположен драйвер. Поэтому выполнение подпрограммы должно начинаться с записи в регистр ds значения сегмента данных задачи. Кроме того, драйвер использует регистр es для доступа к области данных BIOS, поэтому при входе в подпрограмму он может оказаться очищенным. Если подпрограмма использует регистр es при обращениях к видеопамяти, то в него надо записать код сегмента ви-

деобуфера. Перед возвратом из подпрограммы в драйвер восстанавливать исходные значения регистров `ds` и `es` не требуется.

При разработке подпрограммы вы можете использовать регистры общего назначения по своему усмотрению, не заботясь о сохранении их исходных значений. После возврата в драйвер он восстановит содержимое тех регистров, которые нужны для продолжения работы.

Последней выполняемой командой подпрограммы должна быть `retf`. Если подпрограмма использует стек для своих нужд, то к моменту выполнения `retf` должно быть восстановлено исходное (на момент входа в подпрограмму) состояние стека. Последнее слово стека содержит сегмент оперативной памяти, в котором расположен драйвер, а предпоследнее — адрес возврата, относящийся к этому сегменту.

Замечание

Повторный вызов подпрограммы будет возможен только после того, как она выполнит команду `retf` и драйвер завершит обработку предыдущего события. Это упрощает структуру подпрограммы, но ограничивает время, в течение которого она может выполняться.

6.4.2. Примеры прерывающих подпрограмм

Некоторые особенности программирования для режима прерываний были описаны в разделе 5.2.4 на примере обработки аппаратных прерываний от таймера. Программирование работы с мышью имеет свои специфические особенности. Например, при обслуживании двухкнопочной мыши драйвер различает пять событий, но для их обработки он может вызвать только одну подпрограмму. Поэтому мы сначала рассмотрим подпрограмму, реагирующую только на одно событие — перемещение курсора, а затем обсудим, что делать с кнопками.

Способы перемещения курсора. Основные действия, выполняющие перемещение курсора, были описаны в разделе 6.3.3 на примере подпрограммы `Motion`. При работе в режиме прерываний задача должна иметь возможность запрещать перемещение курсора. Например, при построении нового рисунка изображение курсора временно удаляется с экрана. При этом подпрограмма не должна перемещать курсор. Задача может либо временно запрещать драйверу вызов прерывающей подпрограммы, либо устанавливать специальный признак, анализируемый при перемещении курсора.

Первый способ имеет следующий недостаток. В тот момент, когда задача удаляет изображение курсора с помощью подпрограммы `Hidepnt`, значения переменных `Winpnt`, `Offspnt`, `Xpointer`, `Ypointer` (см. пример 6.8) соответствуют значениям СПК внутреннего буфера драйвера. Если задача запретила драйверу вызывать подпрограмму перемещения курсора, а оператор в это

время двигает мышь, то новые значения СПК не будут соответствовать значениям указанных переменных. Когда задача восстановит изображение курсора с помощью подпрограммы `Showpnt`, то оно появится на старом месте. А после восстановления работы прерывающей подпрограммы, при первом движении мыши, курсор скачком переместится в новую позицию.

Второй способ избавлен от этого недостатка. Если установлен специальный признак, то прерывающая подпрограмма вычисляет текущие значения переменных `Winpnt`, `Offspnt`, `Xpointer`, `Ypointer`, но не перемещает курсор. В результате значения этих переменных и СПК всегда будут соответствовать друг другу. Теперь, когда задача вызовет подпрограмму `Showpnt`, изображение курсора появится на новом месте.

Подпрограмма Mousm. Текст подпрограммы, перемещающей курсор по прерываниям, приведен в примере 6.14. В нем использованы имена переменных `curmp` и `pntstat`, их следует добавить к описанным в примере 6.8, т. е. разместить в разделе данных задачи. Вполне достаточно, если они будут иметь размер байта. Советуем вам сравнить текст данной подпрограммы с описанным в примере 6.13.

Пример 6.14. Прерывающая подпрограмма для перемещения курсора

```
Mousm:   mov   ax, data          ; ax = значение сегмента данных
         mov   ds, ax            ; установка сегмента данных
         mov   es, Vbuff         ; установка сегмента видеобуфера
         inc   curmp             ; !! счетчик перемещений курсора
         mov   Xpointer, cx      ; сохранение нового значения
         mov   Ypointer, dx      ; сохранение нового значения
         mov   ax, dx            ; ax = номер строки
         mul   horsize           ; dx:ax = Ypointer*horsize
         add   ax, cx            ; прибавляем к ax номер столбца
         adc   dx, 00            ; учитываем возможность переполнения
         xchg  ax, dx            ; переставляем содержимое ax и dx
         mov   byte ptr GrUnit   ; ax = al*GrUnit
         add   ax, Base_win      ; !! учитываем значение базового окна
         test  pntstat, 01       ; проверка состояния курсора
         je    msmx              ; -> курсор удален с экрана
         call  Hidepnt           ; гасим курсор
         mov   Offspnt, dx       ; сохраняем смещение в Offspnt
         mov   Winpnt, ax        ; сохраняем в Winpnt
         call  Showpnt           ; рисуем курсор и выходим
         retf                    ; возвращение в драйвер
msmx:    mov   Offspnt, dx       ; сохраняем смещение в Offspnt
         mov   Winpnt, ax        ; сохраняем в Winpnt
         retf                    ; возвращение в драйвер
```

Выполнение подпрограммы начинается с восстановления содержимого сегментных регистров ds и es, поскольку оно было изменено драйвером. Далее расположена команда, увеличивающая значение переменной curmp на 1. Она нужна для того, чтобы задача могла определить, перемещался курсор или нет. Если при выполнении вашей задачи такая информация не нужна, то просто исключите команду из текста подпрограммы.

Затем новые значения координат присваиваются переменным Xpointer и Ypointer и пересчитываются в адрес видеопамяти. В отличие от примера 6.13 вычисленные величины сразу не присваиваются переменным Offspnt и Winpnt, а сохраняются в регистрах dx и ax. Это делается потому, что значения указанных переменных можно изменять только после выполнения подпрограммы Hidepnt.

Перед вызовом Hidepnt проверяется состояние младшего разряда переменной pntstat. Если он установлен, то изображение курсора находится на экране, его можно удалять и перемещать. В противном случае содержимое регистров dx и ax сохраняется в переменных Offspnt и Winpnt и команда retf выполняет возврат в драйвер.

Таким образом, подпрограмма Mousm в любом случае изменяет значения переменных, указывающих координаты и адрес видеопамяти для изображения курсора, а само изображение перемещается только в том случае, когда установлен младший разряд переменной pntstat — признак нахождения курсора на экране.

Изменение флага состояния. Для того чтобы каждый раз не вспоминать о необходимости изменить текущее значение переменной pntstat, мы рекомендуем просто включить в текст подпрограмм Showpnt (см. пример 6.5), Hidepnt (см. пример 6.6) и Tglpntr (см. пример 6.4) следующие команды:

```
or    pntstat, 01        ; в подпрограмму Showpnt
and   pntstat, 0Feh      ; в подпрограмму Hidepnt
xor   pntstat, 01        ; в подпрограмму Tglpntr
```

Все три команды изменяют состояние только младшего разряда pntstat, это позволяет, в случае необходимости, использовать остальные разряды pntstat для хранения других нужных для задачи признаков.

Установка Mousm. Для установки подпрограммы Mousm после действий, описанных в примере 6.9, надо выполнить группу команд, приведенную в примере 6.15.

Пример 6.15. Установка прерывающей подпрограммы Mousm

```
;     Дополнение к примеру 6.9
      push  es          ; сохраняем содержимое es
      push  cs          ; помещаем в стек содержимое cs
      pop   es          ; и выталкиваем его в es
```

```
lea   dx, Mousm  ; пишем в dx адрес "Mousm"
mov   cx, 01     ; код события "перемещение курсора"
mouse 0C         ; обращаемся к драйверу мыши
pop   es         ; восстанавливаем содержимое es
```

Подведем итог всему сказанному в данном разделе. Перемещение курсора слабо зависит от действий основной задачи и его можно выполнять в режиме прерываний, поступающих от драйвера. Правда, в таком случае возникает вопрос о том, как задача узнает, что курсор перемещался, если для ее выполнения это необходимо знать. На этот случай мы использовали в примере 6.14 переменную `curmp`. Предполагается, что в нужные моменты времени задача проверяет значение этой переменной и узнает о факте перемещения курсора. Вы можете отказаться от этой переменной и ввести другой признак перемещения курсора, но в любом случае без анализа специального признака задача не узнает о факте перемещения курсора, поскольку оно выполняется без ее прямого участия.

Анализ состояния кнопок. Текущее состояние мыши, как правило, надо анализировать в тех случаях, когда это необходимо для выполнения задачи. Поэтому мы рекомендуем, по крайней мере, на первое время использовать опрос состояния. При этом вместо обращений к драйверу мыши надо проверять значения переменных `LBevent` и `RBevent`, которые формирует вызываемая драйвером подпрограмма, при наступлении соответствующих событий. Вместо 8 разных значений, перечисленных в табл. 6.2, в режиме прерываний `LBevent` и `RBevent` могут иметь только три значения 0, 1 и 2.

Переменная `Mstatus` имела вспомогательное значение и в данном случае она не используется. Вместо нее мы введем три новые переменные:

```
Noevent   db 0  ; признак изменения (1) состояния кнопок
Xasevent  dw 0  ; координата X в момент изменения состояния кнопок
Yasevent  dw 0  ; координата Y в момент изменения состояния кнопок
```

При изменении состояния кнопок подпрограмма устанавливает младший разряд переменной `Noevent`, очищать эту переменную должна задача после обработки события. Переменные `Xasevent` и `Yasevent` введены на тот случай, когда надо знать, в каком месте находился курсор в момент нажатия или отпускания кнопок мыши.

Универсальная подпрограмма. Драйвер всегда обрабатывает только одно событие, это обстоятельство и использовано в подпрограмме `Eventm`. Она либо перемещает курсор, либо формирует новые значения переменных `Noevent`, `LBevent` и `RBevent`. Текст подпрограммы приведен в примере 6.16.

Пример 6.16 . Обслуживание прерываний от драйвера мыши

```
Eventm:  push  ax            ; сохранение содержимого ax
         mov   ax, data      ; ax = значение сегмента данных
```

```
        mov   ds, ax          ; установка сегмента данных
        pop   ax              ; восстановление содержимого ax
        shr   ax, 01          ; сдвиг содержимого ax на разряд вправо
        jnc   btnstat         ; -> курсор не перемещался
;     Сюда надо вставить текст примера 6.14, начиная с третьей команды
Btnstat: mov  Noevent, 01     ; признак изменения состояния кнопок
        mov   Xasevent, cx    ; координата X в момент события
        mov   Yasevent, dx    ; координата Y в момент события
        mov   ah, al          ; копируем al в ah
        and   ax, 0C03h       ; выделяем нужные разряды
        mov   LBevent, al     ; сохраняем код события в LBevent
        shr   ah, 02          ; сдвиг кода в ah на 2 разряда вправо
        mov   RBevent, ah     ; сохраняем код события в RBevent
        retf                  ; возвращение в драйвер
```

Текст примера 6.16 надо дополнить текстом подпрограммы `Mousm`, как это указано в комментарии. При этом получится универсальная прерывающая подпрограмма, перемещающая курсор вслед за мышью и формирующая коды событий, связанных с кнопками при изменении их состояния.

Первая команда примера 6.16 сохраняет в стеке содержимое регистра `ax`, потому, что его портит следующая команда. Перед началом основных действий надо поместить в регистр `ds` значение используемого в задаче сегмента данных, для того чтобы стали доступны расположенные в нем переменные, что и делают две следующие команды. После их выполнения восстанавливается исходное содержимое регистра `ax`.

Выполняемые подпрограммой действия зависят от причины ее вызова, т. е. от кода, находящегося в регистре `ax`. Его содержимое сдвигается на 1 разряд вправо, при этом в зависимости от состояния младшего разряда будет очищен или установлен признак переноса (флаг `Carry`). Если он установлен, то надо перемещать курсор. В противном случае команда `jnz btnstat` выполняет переход на ветку с меткой `btnstat`.

В этой ветке формируются значения переменных `Noevent`, `Xasevent`, `Yasevent`, `LBevent` и `RBevent`. Выполняемые при этом действия не требуют особых пояснений. Если вам непонятен способ вычисления значений `LBevent` и `RBevent`, то вернитесь к разделу 6.4.1, в котором описано, что передает драйвер вызываемой подпрограмме.

Для установки подпрограммы `Eventm` в примере 6.15 надо изменить команды, формирующие адрес подпрограммы и маску событий, а именно:

```
lea   dx, Eventm        ; пишем в dx адрес "Eventm"
mov   cx, 1Fh           ; устанавливаем код маски событий
```

После установки подпрограмма будет вызываться при перемещениях мыши и при изменении состояний ее левой и правой кнопок.

Таким образом, описанная подпрограмма передает задаче все данные об изменении текущего состояния мыши, и дополнительные обращения задачи к драйверу не требуются. Надо просто анализировать значения переменных, которые формирует подпрограмма `Eventm`. Это упрощает структуру задачи, а взаимодействие с драйвером производится только в тех случаях, когда изменяется состояние мыши.

Замечание

Если оператор перемещает мышь или нажимает на ее кнопки в тот момент, когда задачу "не интересует" состояние мыши, то его действия будут зафиксированы в значениях переменных и не окажут никакого влияния на выполнение задачи. Но, вернувшись к опросу, задача отреагирует на последнее изменение расположения мыши и состояния ее кнопок. Для исключения ложных срабатываний задача должна очищать переменные `curmp` и `Noevent` после их использования. Не забывайте также, что в процессе выполнения задачи можно запрещать вызов ранее установленной подпрограммы или устанавливать вместо нее другую подпрограмму.

Мы закончили последнюю из четырех глав, посвященных программированию для видеорежимов `PPG`. Об особенностях именно этих режимов речь шла только при описании работы с рисунками и палитрой цветов. При рассмотрении работы с текстом и курсором автор стремился акцентировать внимание читателя на основных вопросах, безотносительно к видеорежимам. Поэтому приемы, описанные в главах 5 и 6, могут пригодиться вам и при программировании для видеорежимов `direct color`.

ГЛАВА 7

Цвет в коде точки

Видеорежимы с указанием цвета непосредственно в коде точки (`direct color`) были введены в версии стандарта VBE 1.2, опубликованной в октябре 1991 года. К этому времени цветные сканеры уже преодолели барьер в 256 цветов, и возникла необходимость стандартизации способов работы с цветом. Кроме того, за время, прошедшее после публикации первых версий стандарта VBE, элементная база существенно улучшилась и позволяла выпускать видеокарты с нужными техническими характеристиками.

При работе в полноцветных видеорежимах регистры цвета видеокарты не используются, код точки поступает из видеопамяти непосредственно на входы преобразователей код-аналог, выходы которых подключены к монитору. Это исключает необходимость манипуляций с системной палитрой, в которой при работе в режимах `PPG` хранилась копия содержимого регистров цвета видеокарты. И при построении новых рисунков можно не беспокоиться о том, что использованные в них цвета испортят ранее созданное изображение.

Данная глава посвящена особенностям программирования для режимов `direct color`. В ней описаны способы кодирования цвета, пересчет координат точек в адреса видеопамяти, манипуляции с точками и построение рисунков. В последнем случае особое внимание уделено преобразованиям кодов точек образа рисунка в формат, соответствующий видеорежиму. Дополнительно приведен краткий обзор способов сжатия полноцветных рисунков и примеры манипуляций с цветом.

7.1. Кодирование цвета

Манипуляции с графическими объектами во многих случаях зависят не только от размера кода точки, но и от того, как расположены базовые цвета в этом коде. Поэтому мы начнем с описания способов кодирования цвета.

Размер кода точки и расположение в нем базовых цветов зависят от видеорежима. Стандарт VESA предусматривает возможность определения указанных величин при выполнении задачи. В главе 2 мы договорились, что характеристики установленного видеорежима находятся в массиве `Info`, а их перечень приведен в табл. 1.2. В массиве `Info` количество разрядов в коде точки хранится в байте `19h`, а расположение базовых цветов для режимов `direct color` — в байтах `1Fh-26h`. Корректно составленная задача должна использовать эти величины для настройки на конкретный видеорежим.

7.1.1. Среднее количество цветов

Режимы среднего цветового разрешения в англоязычной литературе принято называть `Hi-Color`. При их установке возможны два способа кодирования цвета, различающиеся размерами кода точки. Например, в режиме `110h` код точки занимает 15 разрядов, в которых можно указать 32 768 (или 32K) различных комбинаций (цветов), а в режиме `111h` код точки занимает 16 разрядов, в которых можно указать 65 536 (или 64K) разных комбинаций. Разрешение в обоих режимах одинаковое (640×480 точек), различается только расположение базовых цветов.

Кодирование цвета. В режимах `Hi-Color` код точки занимает одно слово, расположение базовых цветов в его разрядах показано в табл. 7.1.

Таблица 7.1. *Размещение базовых цветов в слове*

	Режим 32K цветов															
	Красный цвет					Зеленый цвет					Синий цвет					
F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	
Режим 64K цветов																
Красный цвет					Зеленый цвет						Синий цвет					
F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	

В режимах 32K коды базовых цветов занимают по 5 разрядов, старший разряд слова не используется. В режимах 64K код зеленого цвета занимает 6 разрядов, поэтому используются все разряды слова. Например, коды базовых цветов максимальной интенсивности имеют следующие значения:

Режим 32K: красный — 7C00h, зеленый — 3E0h, синий — 1Fh.
Режим 64K: красный — 0F800h, зеленый — 7E0h, синий — 1Fh.

Замечание

Трудно сказать, зачем разработчикам стандарта VESA понадобилось вводить экзотический режим 64K, вероятно для этого были какие-то особые причины. Палитру цветов он существенно не расширяет, но зато доставляет дополнительные хлопоты программистам.

Для того чтобы задача могла поддерживать обе разновидности режимов `Hi-Color`, при работе с кодом цвета надо учитывать содержимое байтов `19h` (размер кода точки) и `1F-26h` (расположение базовых цветов) массива `Info`.

Сравнение с режимом PPG. В видеорежимах `PPG` системная палитра позволяла использовать одновременно только 256 разных цветов, а в режимах `Hi-Color` ее размер увеличился в 128 или в 256 раз. Сравнение явно не в пользу режимов `PPG`, но давайте вспомним и еще один факт. В режимах `PPG` код базового цвета занимал 6 разрядов, поэтому точка могла иметь один из 256K различных оттенков, а в режимах `Hi-Color` — в 4 или в 8 раз меньше! Таким образом, в режимах `Hi-Color`, по сравнению с режимом `PPG`, уменьшается цветовое разрешение, но увеличивается разнообразие цветов, которые можно одновременно увидеть на экране. Последнее обстоятельство является решающим доводом в пользу режимов `Hi-Color`, но говорить при этом об улучшении качества передачи цвета никак нельзя.

7.1.2. Максимальное цветовое разрешение

Максимальное цветовое разрешение обеспечивают видеорежимы VESA, которые в англоязычной литературе принято называть `True Color` или `24-bit Color`, последнее название указывает размер *кода цвета, а не кода точки*. В этих режимах базовые цвета имеет 256 градаций, а общая палитра содержит 256×256×256 = 16 777 216 (или 16M) цветов.

Кодирование цвета. Код точки обычно занимает в видеопамяти 32 разряда (4 байта или двойное слово), кроме базовых цветов в него входит дополнительный (резервный) пустой байт. Расположение базовых цветов и пустого байта в разрядах двойного слова показано в табл. 7.2.

Таблица 7.2. *Расположение базовых цветов в 32-разрядном слове*

Байт 3 Разряды 24—31			Байт 2 Разряды 16—23			Байт 1 Разряды 8—15			Байт 0 Разряды 0—7		
1F	...	18	17	...	10	F	...	8	7	...	0
Резервный байт			Красный цвет			Зеленый цвет			Синий цвет		

Байты оперативной и видеопамяти располагаются в порядке увеличения их номеров (адресов). Именно в такой последовательности (слева направо) они выводятся на экран при просмотре содержимого памяти с помощью различ-

ных редакторов. Адрес двойного слова совпадает с адресом его нулевого байта, поэтому первым в памяти хранится код синего цвета, вторым — зеленого, третьим — красного, а четвертый байт пустой (резервный).

Код в памяти и в регистре. Байты регистров принято нумеровать в направлении справа налево. На экран же они выводятся, начиная со старших (слева направо), т. е. так, как расположены десятичные разряды в общепринятой форме записи чисел.

При сравнении содержимого памяти и регистров у неискушенного человека может сложиться впечатление, что базовые цвета переставлены. На самом деле это не так, просто одни и те же данные представлены двумя разными способами. Сказанное иллюстрирует табл. 7.3.

Таблица 7.3. *Расположение кодов цвета в памяти и в регистре*

Название цвета	Байты памяти				Байты памяти			
	0	1	2	3	3	2	1	0
Черный	00	00	00	00	00	00	00	00
Синий	FF	00	00	00	00	00	00	FF
Зеленый	00	FF	00	00	00	00	FF	00
Красный	00	00	FF	00	00	FF	00	00
Белый	FF	FF	FF	00	00	FF	FF	FF

Как уже говорилось, после установки видеорежима его основные характеристики можно прочитать в массиве `Info`. В нем, начиная со смещения `1F`, расположены четыре пары байтов, содержащие размер кода и адрес его младшего бита для красного, зеленого, синего цветов и резервного пространства. Обычно в них находятся следующие коды: `08`, `10h`, `08`, `08`, `08`, `00`, `08`, `18h`, что соответствует табл. 7.2.

После установки режимов `True Color` задача *обязательно* должна проверять байт массива `Info` со смещением `19h`, содержащий размер точки в битах. Если этот байт содержит код `20h`, то видеокарта поддерживает 32-разрядный код точки. Не все видеокарты устанавливают код такого размера, одно из исключений описано в следующем разделе.

Возможность работы в режимах `True Color` зависит от объема памяти, расположенной на видеокарте. При объеме видеопамяти 1 Мбайт эти режимы не поддерживаются. При объеме видеопамяти 2 Мбайт памяти доступны видеорежимы `112h` и `115h`, имеющие разрешения 640×480 и 800×600 точек. При объеме видеопамяти 4 Мбайт к ним добавляется режим `118h` с разрешением 1024×768 точек. Напоминаем, что перечень режимов приведен в табл. 1.1.

Сравнение с режимом Hi-Color. В режимах `True Color` код базового цвета увеличился на три разряда, соответственно количество градаций базовых цветов увеличилось в 8 раз, а общее количество цветовых оттенков — в 256 раз. Поэтому передача цвета в режимах `True Color` существенно улучшается по сравнению с режимами `Hi-Color`. Однако последние в два раза сокращают необходимый объем видеопамяти и во столько же раз или больше ускоряют манипуляции с графическими объектами. Поэтому во многих случаях, например при работе графических акселераторов, основным является режим `Hi-Color`. Тем более, что способность человеческого глаза различать 256 градаций базовых цветов весьма сомнительна.

7.1.3. 24-разрядный код точки

Стандартом VESA не оговорено обязательное наличие резервного байта в коде точки. Поэтому видеокарты, у которых он отсутствует, а код точки занимает всего 24 разряда, формально соответствуют требованиям стандарта VBE 1.2.

Такой размер кода точки был обнаружен автором при исследовании единственной видеокарты MACH64 фирмы ATI Technologies, выпущенной 20 октября 1997 г. Как уже говорилось в главе 1, по данным на сентябрь 1998 года фирма ATI вошла в первую пятерку производителей графических чипов, на ее долю приходится 27% этой продукции. Поэтому весьма вероятно, что видеокарты, поддерживающие 24-разрядный код в режимах `True Color`, будет выпускать не только фирма ATI.

Расположение базовых цветов в коде точки и их размеры соответствуют табл. 7.2, за исключением отсутствующего пустого байта. Поэтому не будем повторять все сказанное о кодировании цвета, а перейдем к существу проблемы.

Недостатки трехбайтового кода. У рассмотренных ранее видеорежимов размер кода точки совпадал с одной из единиц измерения памяти — байт, слово, двойное слово. Нас не интересовало хорошо это или плохо, поскольку не было оснований для постановки такого вопроса, но, прочитав данный раздел, вы поймете, что это было хорошо. Трехбайтовый код точки порождает две основные проблемы.

1. У всех без исключения команд микропроцессоров Intel размер операнда кратен степени двойки, поэтому обработать три байта одной командой нельзя. В таком случае при обмене данными с видеопамятью приходится обрабатывать сначала слово, а затем байт или наоборот, что замедляет процесс обмена. Однако это не самое неприятное.
2. Размер сегмента оперативной или видеопамяти так же кратен степени двойки. Поэтому в нем не помещается целое количество трехбайтовых точек. У одной из них (первой или последней) в текущем сегменте ока-

жется только часть кода, соответствующая одному или двум базовым цветам. Вот это настоящий подарок! Он вынуждает пересмотреть логику манипуляций с точками, которая использовалась до сих пор, и в некоторых случаях применять специальные подпрограммы для записи кодов точек в видеопамять и их чтения из нее.

Подпрограммы для записи и чтения трехбайтового кода точки приведены ниже. При их составлении учтено следующее:

- код точки находится в трех младших байтах регистра `eax`, причем базовые цвета расположены так, как показано в табл. 7.2, а старший резервный байт не используется;
- адрес видеопамяти находится в регистре `di`, а текущее окно задает переменная `Cur_win`, при выполнении подпрограмм исходные значения адреса и окна не изменяются;
- код видеосегмента находится в регистре `es`;
- доступ к видеопамяти происходит через два окна, окно `A` используется при записи, а окно `B` — при чтении;
- если код точки помещается в текущем окне, то подпрограммы должны выполнять минимум вспомогательных действий.

Подпрограмма записи кода точки. Текст подпрограммы, выполняющей запись кода точки из регистра `eax` в видеопамять, приведен в примере 7.1.

Пример 7.1. Подпрограмма записи 24-разрядного кода точки

```
wrtpnt:  push  eax            ; сохранение содержимого eax
         mov   es:[di], al    ; запись первого байта кода точки
         shr   eax, 08        ; сдвиг содержимого eax
         cmp   di, -2         ; сколько байтов до конца окна ?
         jae   wrtpn1         ; -> 1 или 2 байта
         mov   es:[di+1], ax  ; запись остатка кода точки
         pop   eax            ; восстановление содержимого eax
         ret                  ; выход из подпрограммы
wrtpn1:  push  Cur_win        ; сохранение значения Cur_win
         je    wrtpn2         ; -> до конца окна 2 байта
         call  NxtWinA        ; установка следующего окна
         mov   es:[di+1], ax  ; запись остатка кода точки
         jmp   SHORT wrtpn3   ; переход на метку wrtpn3
wrtpn2:  mov   es:[di+1], al  ; запись второго байта кода точки
         call  NxtWinA        ; установка следующего окна
         mov   es:[di+2], ah  ; запись третьего байта кода точки
wrtpn3:  pop   Cur_win        ; восстановление значения Cur_win
         call  SetWinA        ; восстановление исходного окна
         pop   eax            ; восстановление eax
         ret                  ; выход из подпрограммы
```

Выполнение примера 7.1 начинается с сохранения в стеке кода точки, записи в видеопамять его младшего байта и сдвига содержимого регистра `eax` на 8 разрядов влево. После сдвига оставшиеся два байта кода точки находятся в регистре `ax`. Теперь надо проверить, сколько байтов осталось до конца окна, и выбрать способ записи остатка кода точки. Если оставалось больше двух байтов, то команда `jae` не выполняет переход на метку `wrtpn1`. Последние два байта кода точки записываются в видеопамять, восстанавливается исходный код в регистре `eax` и происходит возврат из подпрограммы.

Если до конца окна оставалось 1 или 2 байта, то команда `jae` выполняет переход на метку `wrtpn1`. На этой ветке подпрограммы сначала в стеке сохраняется значение переменной `Cur_win` и вновь проверяется оставшееся количество байтов (команда `push` не изменяет состояние флагов).

Если до конца окна оставался 1 байт, то команда `je` не производит переход на метку `wrtpn2` и выполняется следующая команда, устанавливающая новое окно видеопамяти. Затем в это окно записываются два оставшихся байта кода точки, и происходит безусловный переход на метку `wrtpn3` для завершающих действий.

Если до конца окна оставалось ровно 2 байта, то команда `je` выполнит переход на метку `wrtpn2`. В этом случае в видеопамять надо записать еще один байт кода точки, находящийся в регистре `al`, вызвать подпрограмму для установки следующего окна и записать в это окно старший байт кода точки из регистра `ah`.

Фрагмент подпрограммы, начинающийся с метки `wrtpn3`, является общим для случаев, когда до конца буфера оставался один или два байта. В нем восстанавливается исходное значение переменной `Cur_win`, исходное окно видеопамяти и содержимое регистра `eax`. После этих действий происходит возврат на вызывающий модуль.

Подпрограмма чтения кода точки. Текст подпрограммы приведен в примере 7.2. При ее использовании в разделе данных задачи надо зарезервировать один байт, присвоив ему имя `DotBuff`.

Пример 7.2. Подпрограмма чтения 24-разрядного кода точки

```
rdpnt:   xor   eax, eax       ; очистка регистра eax
         mov   al, es:[di]    ; чтение младшего байта кода точки
         mov   DotBuff, al    ; и его сохранение в DotBuff
         cmp   di, -2         ; сколько байтов до конца окна ?
         jae   rdpnt2         ; -> 1 или 2 байта
         mov   ax, es:[di+1]  ; чтение старших байтов кода точки
rdpnt1:  shl   eax, 08        ; сдвиг содержимого eax влево
         mov   al, DotBuff    ; добавление кода младшего байта
         ret                  ; выход из подпрограммы
```

```
rdpnt2:  push  Cur_win        ; сохранение значения Cur_win
         je    rdpnt3         ; -> до конца окна 2 байта
         call  NxtWinB        ; установка следующего окна
         mov   ax, es:[di+1]  ; чтение старших байтов кода точки
         jmp   SHORT rdpnt4   ; переход на метку rdpnt4
rdpnt3:  mov   al, es:[di+1]  ; чтение 2-го байта кода точки
         call  NxtWinB        ; установка следующего окна
         mov   ah, es:[di+2]  ; чтение старшего байта кода точки
rdpnt4:  pop   Cur_win        ; восстановление значения Cur_win
         call  SetWinB        ; восстановление исходного окна
         jmp   SHORT rdpnt1   ; переход на метку rdpnt2
```

Выполнение примера 7.2 начинается с очистки регистра `eax`. Это нужно для очистки старшего байта формируемого кода. Затем младший байт кода точки считывается в регистр `al` и помещается в `DotBuff`. Теперь надо проверить, сколько байтов осталось до конца окна, и выбрать способ чтения старших байтов кода точки.

Если до конца окна осталось больше двух байтов, то переход на метку `rdpnt2` не происходит и выполняется команда, следующая за `jae rdpnt2`. Она помещает в регистр `ax` два старших байта кода точки. Содержимое регистра `eax` сдвигается на 8 разрядов влево, в освободившийся младший байт копируется содержимое `DotBuff` и происходит возврат из подпрограммы на вызывающий модуль.

Если до конца окна осталось меньше чем 3 байта, то команда `jae` выполняет переход на метку `rdpnt2`. При этом в стеке сохраняется значение переменной `Cur_win`, и если в буфере остался 1 байт, то команда `je` не выполняет переход на метку `rdpnt3`. В этом случае устанавливается следующее окно видеопамяти, в регистр `ax` считываются два старших байта кода точки, и происходит безусловный переход на метку `rdpnt4` для завершающих действий.

Если до конца окна осталось 2 байта, то команда `je rdpnt3` выполняет переход на метку `rdpn3`. В таком случае в регистр `al` сначала записывается средний код точки, после этого устанавливается следующее окно видеопамяти и в регистр `ah` считывается старший байт кода точки.

Далее выполняется фрагмент подпрограммы, имеющий метку `rdpnt4`. В нем восстанавливаются значение переменной `Cur_win` и исходное окно видеопамяти, после чего происходит безусловный переход на метку `rdpnt1` для окончательного формирования кода и выхода из подпрограммы.

Работа с двумя окнами. В приведенных примерах использованы имена подпрограмм `NxtWin` и `SetWin` с добавленными к ним буквами `A` и `B`. Они встречаются первый раз, потому уточним, о чем идет речь.

Способы работы с двумя окнами видеопамяти описаны в разделе 2.4. Там говорилось о двух вариантах переключения окон, одно из которых доступно

только для чтения, а другое только для записи. Первый вариант основан на одновременном переключении окон. Текст соответствующей подпрограммы `SetWin` приведен в примере 2.9. Второй вариант основан на независимом переключении окон для записи и чтения. В разделе 2.4 описано, как составить две группы подпрограмм для независимой работы с окнами. При этом рекомендовалось добавить к основным именам подпрограмм буквы `A` и `B`. Выбор одного из этих вариантов зависит от конкретных особенностей алгоритма преобразования графического объекта. До сих пор нас вполне устраивало одновременное изменение номеров обоих окон.

Таким образом, если при использовании подпрограмм примеров 7.1 и 7.2 необходимо работать с двумя разными окнами, то надо использовать два комплекта подпрограмм с именами, указанными в примерах 7.1 и 7.2. Если же допустимо одновременное переключение окон, то в именах подпрограмм надо просто убрать буквы `A` и `B`.

Когда используются подпрограммы. Расположение кода точки в двух смежных окнах событие достаточно редкое. Если вести отсчет от начала видеопамяти, то в трех подряд расположенных окнах оно происходит дважды. Например, при установке режима `112h` на экране помещается 307 200 точек, их коды занимают 15 неполных окон видеопамяти. Из них только коды 10 точек расположены в двух смежных окнах (10 случаев из 307 200)!

Если код точки расположен в одном окне, то в описанных подпрограммах выполняется 9 команд при записи и 10 при чтении (с учетом команды вызова подпрограммы). Поэтому при разработке конкретной задачи имеет смысл взвесить все доводы за и против применения описанных подпрограмм. Одним из возможных компромиссов между размером задачи и временем ее выполнения является обращение к подпрограммам только в тех случаях, когда код точки находится в двух смежных окнах.

Замечание

При работе видеокарты ATI MACH64 в режимах `True Color` используемый объем видеопамяти сокращается на 25%. В обмен на это мы получаем не только усложнение программ и замедление процесса выполнения задач. *Впервые* видеокарта, формально соответствующая стандарту VESA, оказывается несовместимой с другими видеокартами. Эта несовместимость проявляется только при работе в среде DOS в режимах `True Color`, но факт остается фактом!

7.2. Координаты и адреса точек

Для вывода точки заданного цвета в нужное место экрана надо связать координаты этого места с адресом видеопамяти, по которому должен быть записан код точки. Поэтому мы вновь возвращаемся к вопросам, рассмотренным в разделе 3.1.3, но с учетом особенностей режимов `direct color`.

Новые переменные. После чтения массива `Info` задаче доступны две величины, имеющие отношение к разрешающей способности режима.

1. Одна из них расположена в слове со смещением `12h`, она указывает ширину экрана, выраженную в точках. В главе 2 предлагалось хранить копию этого слова в переменной `Horsize`, которая затем неоднократно использовалась в примерах.
2. Другая величина расположена в слове со смещением `10h`, она указывает, сколько байтов видеопамяти отображается при выводе строки на экран. Иначе говоря, это ширина строки, умноженная на размер кода точки, выраженный в байтах. В документации VESA она называется `bytes per scan line`.

При работе в режимах `PPG` обе величины совпадают, поэтому мы использовали только первую из них. Теперь нам может пригодиться и вторая величина, поэтому после чтения массива `Info` ее значение надо присвоить переменной `bperline`, она была описана в примере 2.11, но не применялась.

В некоторых случаях нам будет нужен размер кода точки в байтах. Такой величины в массиве `Info` нет, но байт со смещением `19h` содержит количество разрядов в коде точки. Если его сдвинуть на три разряда вправо и результат преобразовать в слово, то получится нужная нам переменная.

Если копию этой переменной сдвинуть еще на 1 разряд вправо, то получится еще одна переменная, содержащая количество слов в коде точки. Имена и описания новых переменных следующие:

```
bperline   dw  2560   ; размер строки отображаемой на экране в байтах
bytppnt    dw  0004   ; размер кода точки, выраженный в байтах
wrdppnt    dw  0002   ; размер кода точки, выраженный в словах
```

В этом описании переменных указаны значения, которые получаются при установке режима `112h` — `True Color`, 640×480 точек.

Преобразование координат в адрес выполняется перед началом работы с большинством графических объектов. Примеры таких преобразований при работе в режимах `PPG` приводились неоднократно. Здесь нас будет интересовать универсальный вариант преобразования, который можно использовать при работе во всех видеорежимах VESA.

При описании подпрограммы примера 6.13 (перемещение курсора) было рекомендовано для учета размера кода точки, после вычисления адреса, сдвинуть результат на 1 или 2 разряда влево, что равносильно умножению на 2 или на 4. Это самый простой, но не универсальный способ, поскольку при составлении программы надо знать величину сдвига, которая зависит от видеорежима. Кроме того, с помощью сдвигов невозможно выполнить умножение на 3, нужны дополнительные команды сложения. Поэтому, в общем случае, целесообразно выполнять умножение, а не сдвиг.

Формулу для вычисления адреса по заданным значениям координат `X` и `Y` можно записать в следующем виде:

```
Address = (Y*horsize + X)*bytppnt
```

При замене умножения на `bytppnt` сдвигами действия выполнялись в той последовательности, в какой они указаны в формуле — сначала умножение, затем сложение и, наконец, сдвиг. Если же сдвиг заменить умножением, то последовательность действий придется изменить.

Результат заключенных в скобки действий расположен в двух регистрах, `dx` содержит его старшую часть, а `ax` — младшую. Для умножения двойного слова (или содержимого двух регистров) на значение переменной `bytppnt` потребуется много вспомогательных действий. Чтобы упростить вычисления в приведенной выше формуле, надо раскрыть скобки и учесть, что `bperline = horsize*bytppnt`, в результате получится следующее выражение:

```
Address = Y*bperline + X*bytppnt
```

Подпрограмма Caladdr. В примере 7.3 приведен текст подпрограммы, выполняющей вычисления по этой формуле. Перед ее вызовом в регистре `cx` указывается номер столбца (координата `X`), а в регистре `dx` — номер строки (координата `Y`). Вычисленный адрес помещается в регистры `dx:ax`, т. е. в `ax` находится значение окна, а в `dx` — адрес (смещение) в этом окне.

Пример 7.3. Универсальная подпрограмма вычисления видеоадреса

```
Caladdr: mov   ax, bperline     ; ax = размер строки в байтах
         mul   dx               ; dx:ax = Y*bperline
         push  dx               ; сохраняем старшую часть результата
         xchg  ax, cx           ; обмен содержимого регистров
         mul   bytppnt          ; ax = X*bytppnt, dx = 0
         add   ax, cx           ; вычисляем младшую часть адреса
         mov   dx, ax           ; и сохраняем ее в регистре dx
         pop   ax               ; ax = старшая часть Y*bperline
         adc   ax, 00           ; учитываем возможность переноса
         mul   byte ptr GrUnit  ; ax = al * GrUnit
         add   ax, Base_win     ; !! если используется базовое окно
         ret                    ; выход из подпрограммы
```

Текст примера 7.3 не нуждается в подробных пояснениях, обращаем ваше внимание только на следующие особенности. Содержимое регистра `dx` (старшую часть произведения `Y*bperline`) надо сохранить в стеке потому, что оно будет испорчено при втором умножении. После второго умножения и вычисления младшей части адреса старшая часть выталкивается из стека в регистр `ax`. К ней прибавляется единица переноса, которая могла возникнуть, если при выполнении команды `add ax, cx` произошло переполнение и

был установлен С-разряд регистра флагов (признак Carry). Команды пересылки и выталкивания из стека не изменяют состояние С-разряда. Поэтому если он был установлен, то adc ax, 00 прибавит единицу к содержимому регистра ax.

Можно изменить текст примера 7.3 так, чтобы вычисленный адрес возвращался в регистре di, значение окна присваивалось переменной Cur_win и выполнялась установка окна (call SetWin). В результате получится вариант подпрограммы CallWin, описанной в примере 3.4, применимый в любых видеорежимах VESA.

Другой вариант Caladdr. В примере 7.4 показан вариант подпрограммы Caladdr, в котором вместо умножения X*bytppnt содержимое регистра cx сдвигается на K разрядов влево. В зависимости от видеорежима, в команде сдвига букву K надо заменить цифрами 1 или 2, т. е. эта подпрограмма не универсальна.

Пример 7.4. Пересчет координат в адрес с использованием сдвига

```
Caladdr: mov   ax, bperline     ; ax = размер строки в байтах
         mul   dx               ; dx:ax = Y*bperline
         shl   cx, k            ; k=1 для Hi-Color; k=2 для True Color
         add   ax, cx           ; вычисляем младшую часть адреса
         adc   dx, 00           ; учитываем возможность переноса
         xchg  ax, dx           ; обмен содержимого регистров
         mul   byte ptr GrUnit  ; ax = al * GrUnit
         add   ax, Base_win     ; !! если используется базовое окно
         ret                    ; выход из подпрограммы
```

В примере 7.4 сдвигается не результат умножения, а только значение координаты X (содержимое регистра cx). Это возможно потому, что значение координаты Y (содержимое регистра dx) умножается не на Horsize, а на bperline = horsize*bytppnt.

Особенность операций сдвигов заключается в том, что величина сдвига может либо находиться в регистре cl (младший байт регистра cx), либо указываться непосредственно в команде, других вариантов нет. Поэтому для автоматического выбора величины сдвига в примере 7.4 вместо двух подряд расположенных команд shl cx, k и add ax, cx надо записать следующие:

```
mov         bx, wrdppnt    ; bx = величина сдвига
xchg        bx, cx         ; обмен содержимого регистров
shl         bx, cl         ; сдвиг значения координаты X
add         ax, bx         ; вычисляем младшую часть адреса
```

Напомним, что если видеокарта в режиме True Color поддерживает трехбайтовый код точки, то заменять умножение сдвигами не целесообразно.

Подведем итог. Первая из двух описанных подпрограмм универсальная, а вторая специализированная. Вопрос о том, какая из них лучше, вообще говоря, не корректен. Корректен другой вопрос — в каких случаях нужны универсальные подпрограммы, а в каких специализированные. Первые целесообразно составлять при разработке библиотечных модулей, особенно для языков высокого уровня. А если вы разрабатываете задачу, в которой большинство подпрограмм специализировано, то целесообразность включения в ее текст одной или нескольких универсальных подпрограмм весьма проблематична.

Координаты и адреса смежных точек. Значения координат нужны для вычисления базового адреса видеопамяти, соответствующего некой опорной точке, как правило, левого верхнего угла, графического объекта. В процессе работы с объектом адреса остальных точек вычисляются упрощенным способом исходя из текущего значения адреса, т. е. учитывается зависимость приращения адресов от взаимного расположения точек на экране.

Если базовая точка не лежит на одной из четырех границ экрана, то ее окружает восемь смежных точек. В табл. 7.4 показаны приращения значений координат и адресов смежных точек. Базовая точка имеет координаты `X`, `Y`, а приращение ее адреса равно нулю. В правой части таблицы буква `k` соответствует переменной `bytppnt`, а буква `w` — переменной `bperline`.

Таблица 7.4. *Приращения координат и адресов смежных точек*

Приращение координат смежных точек			Приращение их адресов		
X-1, Y-1	X, Y-1	X+1, Y-1	-k-w	-w	k-w
X-1, Y	X, Y	X+1, Y	-k	**0**	k
X-1, Y+1	X, Y+1	X+1, Y+1	w-k	w	w+k

Из табл. 7.4, в частности, следует, что при перемещении по горизонтали адреса точек уменьшаются или увеличиваются на значение переменной `bytppnt`. Если для работы с кодами точек используются обычные операции, то после их выполнения текущий адрес надо изменить на `bytppnt`. Если же применяются строковые операции, то они автоматически изменяют текущий адрес на размер операнда. При обработке точек в естественном порядке, т. е. в сторону увеличения значений их координат, строковые операции увеличивают адрес, а при обработке точек в обратном порядке уменьшают его. Таким образом, при последовательной обработке точек строки для получения адреса очередной точки достаточно простой переадресации операндов.

Адрес следующей строки. Если строки графического объекта обрабатываются последовательно друг за другом, то после построения текущей строки надо определить адрес начала следующей (или предыдущей) строки. В этом случае простой переадресации операндов недостаточно.

Первые точки строк прямоугольной области расположены на экране в одном столбце. Значения координаты `X` у них совпадают, а координаты `Y` различаются на величину, кратную значению переменной `bperline`. Это и надо учесть при вычислениях.

Адрес начала текущей строки можно хранить в специально выделенном месте и для перехода к следующей строке увеличивать или уменьшать его значение на `bperline`. Однако, как уже говорилось в разделе 3.2.2, неизвестно, какому окну принадлежит сохраненный адрес, поскольку при обработке строки могла произойти смена окна. Поэтому при таком способе вычислений будет нужен специальный признак переключения окна и анализ его состояния.

Для упрощения вычислений надо использовать адрес, полученный в конце обработки текущей строки. Он заведомо принадлежит установленному окну, а отличается от адреса начала строки на ширину прямоугольной области, выраженную в байтах. Поэтому если к нему прибавить значение переменной `bperline`, уменьшенное на ширину прямоугольной области, то получится адрес начала следующей строки. Ширина прямоугольной области задается в виде количества точек, которое надо умножить на размер кода точки. Приведенные рассуждения можно записать в виде следующей формулы вычисления константы коррекции адреса (`offsline` обозначает смещение строки).

```
offsline = bperline – widthrect*bytppnt = (horsize – widthrect)*bytppnt.
```

В этих формулах `widthrect` обозначает ширину прямоугольной области, выраженную в точках, имена остальных переменных вам известны. Оба варианта формулы равноценны по количеству выполняемых действий.

При работе в режимах `PPG bytppnt=1` и формула превращается в разность (`horsize – widthrect`), которая и вычислялась в приведенных ранее примерах. В разделах 6.1.5 и 6.3.3 при описании подпрограмм построения и перемещения изображения курсора мы советовали в режимах `direct color` сдвигать указанную разность на 1 или 2 разряда влево. Такой прием прост, но не универсален. Универсальные способы вычисления значения `offsline` для режимов `direct color` описаны ниже в разделе 7.3.3.

Мы обсудили два простых способа вычисления адресов при работе с графическими объектами, но ими не исчерпывается все разнообразие возможных вариантов. В общем случае при рисовании линий и геометрических фигур приращения адресов смежных точек не равны единице, они вычисляются по специальным алгоритмам. Поэтому основные действия при работе со сложными объектами сводятся к вычислению адресов точек, а собственно графика отступает на второй план.

7.3. Линии, строки и прямоугольные области

Описание манипуляций с кодами точек мы начнем со случаев, когда они просто записываются в видеопамять или считываются из нее. Для записи кодов нескольких точек при этом используются специальные подпрограммы, рисующие линии геометрических фигур или выполняющие построение строк рисунков. В главе 3 описано несколько подпрограмм различного назначения для режимов PPG. В данном разделе будут рассмотрены аналогичные подпрограммы, предназначенные для выполнения в режимах direct color.

Способ пересылки зависит от размера кода точки и не зависит от расположения в нем базовых цветов. В режимах direct color код точки может занимать 2, 3 или 4 байта, а команды пересылки и строковые операции работают только со словами (2 байта) или с двойными словами (4 байта). Тем не менее при определенных условиях можно составить подпрограммы, выполнение которых не зависит от размера кода точек. Мы опишем эти условия и приведем примеры универсальных подпрограмм для рисования линий и построения строк рисунков.

7.3.1. Подпрограммы для рисования линий

При оформлении рабочей области экрана часто используются одноцветные горизонтальные и вертикальные линии. Способ их рисования зависит от угла наклона и направления линии. Изображение на экране дискретно по своей природе, поэтому гладкими являются только линии, наклоненные под углом кратным 45 градусам, при их рисовании значения одной или обеих координат изменяются от точки к точке на 1. В остальных случаях приращения координат вычисляются по специальным алгоритмам. От направления линии зависят способы переадресации операндов и изменения окна видеопамяти в тех случаях, когда значения адресов выходят за пределы сегмента.

Перечисленные особенности рисования линий подробно обсуждались в разделе 3.2.1, там же было приведено несколько вариантов подпрограмм, выполняющих соответствующие действия при работе в видеорежимах PPG. В данном разделе описано рисование одноцветных горизонтальных линий в прямом направлении (слева направо). При этом, основное внимание уделяется влиянию размера кода точки на выполняемые действия.

Базовые варианты подпрограмм. В примере 7.5 приведены два варианта подпрограмм рисования линии, различающиеся способом пересылки. Перед их вызовом код цвета точки помещается в регистр ax или eax, а размер линии (количество точек) в cx. Исходный адрес видеопамяти указывается в регист-

ре di, и устанавливается окно видеопамяти, которому принадлежит этот адрес. Как обычно, регистр es должен содержать код видеосегмента (значение переменной Vbuff).

Пример 7.5. Цикл рисования горизонтальной линии в режимах Hi-Color

```
        ; Вариант 1, используется команда пересылки
horline: mov   es:[di], ax  ; !! для True Color — mov es[di], eax
        add   di, bytppnt  ; переадресация операнда
        jne   @F           ; переход, если не нуль
        call  NxtWin       ; установка следующего окна
@@:     loop  horline      ; управление повторами цикла
        ret                ; возврат из подпрограммы
        ; Вариант 2, используется строковая операция.
horline: stosw             ; !! для True Color — stosd
        or    di, di       ; начало нового сегмента ?
        jne   @F           ; -> нет
        call  NxtWin       ; установка следующего окна
@@:     loop  horline      ; управление повторами цикла
        ret                ; возврат из подпрограммы
```

Текст примера 7.5 отличается от текста примера 3.6 незначительными изменениями. В первом варианте при переадресации используется не 1, а значение переменной bytppnt, которое равно 2 или 4.

Для использования подпрограмм примера 7.5 в режимах True Color надо первые команды в обоих вариантах заменить командами, указанными в комментариях. В этих режимах перед вызовом подпрограмм код цвета точек помещается в регистр eax, поскольку он занимает 32 разряда.

Давайте уточним, почему приведены два варианта циклов, если они содержат одинаковое количество команд. Команда пересылки удобна в тех случаях, когда переадресация не может выполняться сразу после записи или чтения кода точки (см. пример 6.5), или когда шаг переадресации не совпадает с размером кода точки, например при рисовании вертикальных линий. Если указанные условия не существенны, то второй вариант цикла 7.5 предпочтительнее. После компиляции он окажется короче первого на 3 байта, и будет выполняться несколько быстрее. Но главное, при определенных условиях возможно применение микропрограммного цикла, существенно ускоряющего процесс рисования. Об этом мы поговорим особо.

Ускорение цикла рисования. В примере 7.5 после каждой переадресации выполняется проверка принадлежности нового значения адреса текущему сегменту и, в случае необходимости, устанавливается следующее окно видеопамяти. Вероятность того, что новое значение адреса выйдет за границу сегмента, достаточно мала. Например, при установке видеорежимов 110h

или 111h (Hi-Color, 640×480) рабочее пространство видеопамяти занимает 9 неполных окон. Если создаваемое изображение заполняет все это пространство, то только в восьми случаях из 307 200 возникнет необходимость изменения окна, а в остальных случаях проверка адресов не требуется. Сказанное не означает, что она вообще не нужна. Для сокращения количества дополнительных действий надо изменить способ проверки.

Один из вариантов сокращения бесполезных действий заключается в том, чтобы перед началом рисования линии проверять, помещается она в текущем сегменте или выходит за его пределы. Это исключает необходимость контроля адресов в цикле рисования. Если линия целиком расположена в одном сегменте, то цикл рисования выполняется один раз. В противном случае после рисования части линии, расположенной в текущем окне, устанавливается следующее окно и повторяется цикл рисования остатка линии.

Замечание

Рисование линии по частям в режимах PPG уже описано в разделе 3.2.1, а соответствующая подпрограмма приведена в примере 3.8.

Подпрограмма Twopart. Нас интересует не простое рисование, а возможность ускорения выполнения различных действий с точками, расположенными на прямой линии. Поэтому текст примера 7.6 является своеобразным управляющим алгоритмом, который для выполнения конкретных действий вызывает вспомогательную подпрограмму baselp. В данном примере она рисует прямую линию.

Входные параметры подпрограммы Twopart полностью соответствуют параметрам подпрограммы horline (пример 7.5) и расположены в тех же регистрах.

Пример 7.6. Рисование линии по частям в режимах Hi-Color

```
Twopart:  push  dx          ; сохраняем содержимое регистра dx
          mov   dx, di      ; копируем адрес в регистр dx
          shl   cx, 01      ; !! для True Color — shl cx, 02
          add   dx, cx      ; сумма исходного адреса и размера линии
          jc    @F          ; -> прямая расположена в двух окнах
          xor   dx, dx      ; очистка регистра dx
@@:       sub   cx, dx      ; количество точек в текущем окне
          shr   cx, 01      ; !! для True Color — shr cx, 02
          call  baselp      ; рисуем всю линию или ее первую часть
          or    di, di      ; адрес в пределах текущего окна ?
          jne   hrl_exit    ; -> да, линия нарисована полностью
          call  NxtWin      ; установка следующего окна
          add   cx, dx      ; количество не нарисованных точек
          je    hrl_exit    ; линия нарисована полностью
```

```
          shr   cx, 01       ; !! для True Color — shr cx, 02
          call  baselp       ; рисуем остаток линии
hrl_exit: pop   dx           ; восстановление содержимого dx
          ret                ; возврат из подпрограммы
;         Подпрограмма, выполняющая основные действия
baselp:   mov   es:[di],ax   ; !! для True Color — mov es:[di], eax
          add   di, bytppnt  ; переадресация операнда
          loop  baselp       ; управление повторами цикла
          ret                ; возврат из подпрограммы
```

Выполнение примера 7.6 начинается с сохранения в стеке содержимого регистра `dx`, поскольку оно изменяется в подпрограмме. При входе регистр `cx` содержит количество рисуемых точек, его надо преобразовать в количество байтов, с помощью сдвига и сложить с исходным адресом видеопамяти (в регистре `dx`). Если при этом происходит переполнение, то линия не помещается в текущем окне и ее надо рисовать по частям. В противном случае регистр `dx` очищается. После этого вычисляется количество точек в первой части и вызывается подпрограмма `baselp`, рисующая начало линии.

При первом вызове `baselp` может быть нарисована вся линия или только ее первая часть. Это важно знать для выполнения дальнейших действий. Они начинаются с проверки значения адреса, находящегося в регистре `di`.

Возможен случай, когда при первом рисовании достигнута граница окна. В таком случае в регистре `di` находится 0, но нулевой адрес принадлежит не текущему, а следующему окну видеопамяти. Поэтому если регистр `di` очищен, то обязательно надо сменить окно видеопамяти. Если же содержимое `di` отлично от нуля, то нарисована вся линия.

После установки окна суммируется содержимое регистров `cx` и `dx`. Если сумма равна нулю, то линия нарисована целиком, а код ее последней точки был записан в последнее слово сегмента. В противном случае количество байтов, полученное в регистре `cx`, преобразуется в количество точек и повторно вызывается подпрограмма `baselp`. Перед возвратом на вызывающий модуль восстанавливается исходное содержимое регистра `dx`, соответствующая команда имеет метку `hrl_exit`.

Замечание

Практическая ценность примера 7.6 заключается в том, что он иллюстрирует способ обработки строки графического объекта по частям. Основные действия локализованы в подпрограмме `baselp`. Ее можно изменить так, чтобы вместо записи в видеопамять содержимого регистра `ax` выполнялись другие действия, например инверсия кодов точек, пересылка кодов из видеопамяти в оперативную или наоборот и т. д.

Ускоренное рисование линии. В подпрограмме `baselp`, текст которой приведен в примере 7.6, работу с видеопамятью выполняет одна команда, поэтому

сразу после нее возможна переадресация операнда. Если при этом шаг переадресации совпадает с размером операнда, то вместо команды пересылки можно использовать строковую операцию. В таком случае тело цикла пересылки сокращается до одной команды `rep stosw`, которую надо вставить вместо `call baselp`. Это и сделано в примере 7.7.

Пример 7.7. Ускоренное рисование линии в режимах Hi-Color

```
horline:  push  dx          ; сохраняем содержимое регистра dx
          mov   dx, di      ; копируем адрес в регистр dx
          shl   cx, 01      ; !! для True Color — shl cx, 02
          add   dx, cx      ; сумма исходного адреса и размера линии
          jc    hrl_1       ; -> прямая расположена в двух окнах
          xor   dx, dx      ; очистка регистра dx
hrl_1:    sub   cx, dx      ; количество точек в текущем окне
          shr   cx, 01      ; !! для True Color — shr cx, 02
          rep   stosw       ; !! для True Color — stosd
          or    di, di      ; адрес в пределах текущего окна ?
          jne   hrl_exit    ; -> да, линия нарисована полностью
          call  NxtWin      ; установка следующего окна
          mov   cx, dx      ; количество не нарисованных точек
          shr   cx, 01      ; !! для True Color — shr cx, 02
          rep   stosw       ; !! для True Color — stosd
hrl_exit: pop   dx          ; восстановление содержимого dx
          ret               ; возврат из подпрограммы
```

Обратите внимание на то, что во второй части примера 7.7 проверяется только содержимое регистра `di` и не проверяется оставшееся количество точек. Это допустимо потому, что если регистр `cx` очищен, то цикл `rep stosw` не будет выполняться и предварительная проверка содержимого `cx` не обязательна.

В комментариях к примерам 7.6 и 7.7 указано, как надо изменить переменные команды для использования подпрограмм в режимах `True Color`, в таком случае код цвета линии помещается в регистр `eax`.

Условное ассемблирование. Тексты трех приведенных примеров рисования линий являются стандартными заготовками, у которых от видеорежима зависит лишь несколько команд. Исключить эту зависимость и сделать подпрограммы универсальными невозможно. Однако их исходный текст можно подготовить так, чтобы Макроассемблер выбирал нужный вариант команды в зависимости от заданного вами признака. Это упростит вашу работу и позволит включать в текст задачи заранее подготовленные и отлаженные исходные тексты подпрограмм.

Признак (условие для выбора) описывается как обычная константа и располагается в начале текста программы перед описанием первого сегмента.

Предположим, что ему присвоено имя `variant`, а значения равны 1 для режимов `Hi-Color` и 2 для режимов `True Color`. Описание выглядит так:

```
variant = 1    ; Описание признака "variant" для режимов Hi-Color
```

В текст подпрограммы, приведенной в примере 7.8, вместо переменной команды вставлен условный блок, в котором описаны варианты выбора.

Пример 7.8. Выбор варианта команды по заданному признаку

```
IF      variant EQ 1        ; проверка условия выбора
        mov es:[di], ax     ; основной вариант команды
ELSE                        ; признак альтернативного варианта
        mov es:[di], eax    ; альтернативный вариант команды
ENDIF                       ; конец условного блока
```

При выполнении примера 7.8 Макроассемблер выберет вариант команды пересылки в зависимости от значения признака `variant`. Если `variant=2`, то будет выбрана команда, записанная после `ELSE`.

Дополнительно отметим, что в условных блоках альтернативный вариант может отсутствовать, если он не нужен, и в обоих вариантах может быть задана не одна, а несколько команд.

Значения признака `variant` выбраны не случайно. При таких значениях в примерах 7.6 и 7.7 в операциях сдвига можно заменить 1 на имя `variant`. Тогда в примере 7.6 останется только одна переменная команда, а в примере 7.7 — две. Соответственно, в текст примера 7.6 понадобится включить один условный блок (описанный в примере 7.8), а в текст примера 7.7 — два для выбора вариантов команд `rep stosw` или `rep stosd`.

Таким образом, при создании подпрограмм для видеорежимов `direct color` можно использовать условное ассемблирование и специальные признаки. Это позволяет включать в исходные тексты задач заранее подготовленные заготовки подпрограмм, выполняющих нужные действия.

Трехбайтовый код точки. Такой код не укладывается в общую схему по двум причинам. Во-первых, размер операндов команд не может быть равен трем байтам. Во-вторых, существуют особые точки, код которых расположен в двух смежных сегментах. Поэтому нужны специальные подпрограммы, при составлении которых учитываются особенности трехбайтовых кодов. Две такие подпрограммы, выполняющие запись и чтение кода точки, приведены в примерах 7.1 и 7.2 (см. раздел 7.1.3), первая из них (`wrtpnt`) нам пригодится.

При описании подпрограмм `rdpnt` и `wrtpnt` говорилось, что к ним желательно обращаться только для чтения или записи особых точек, а остальные точки обрабатывать более простым способом. Поэтому мы составим специализированную подпрограмму, которая самостоятельно обрабатывает боль-

шинство точек и вызывает wrtpnt только для записи кодов последних точек видеосегментов.

Текст такой подпрограммы приведен в примере 7.9, перед вызовом адрес первой точки, как обычно, помещается в регистры es:di, а код цвета точки в регистр eax, это сделано для совместимости с четырехбайтовыми режимами.

Пример 7.9. Рисование линии, режим True Color, трехбайтовый код

```
horline: push  bx             ; сохраняем содержимое регистра bx
         mov   ebx, eax       ; копируем eax в ebx
         shr   ebx, 16        ; bx = старшие разряды кода точки
hrln_1:  cmp   di, -3         ; это последняя точка в окне?
         jae   hrln_3         ; -> да, особый случай
         stosw                ; запись двух младших байтов кода
         mov   es:[di], bl    ; запись старшего байта кода
hrln_2:  inc   di             ; переадресация операнда
         loop  hrln_1         ; управление повторами цикла
         pop   bx             ; восстанавливаем содержимое bx
         ret                  ; возврат из подпрограммы
hrln_3:  call  wrtpnt         ; запись кода особой точки
         call  NxtWinA        ; установка следующего окна
         jmp   short hrln_2   ; короткий переход на метку hrln_2
```

Для упрощения действий, выполняемых в основном цикле примера 7.9, код точки надо расположить в регистрах ax и bx. Исходное содержимое регистра bx сохраняется в стеке, а на его место помещается старшая половина регистра eax, поэтому код красного цвета оказывается в регистре bl.

Основной цикл имеет метку hrln_1 и начинается с проверки адреса. Если окажется, что до конца сегмента осталось больше чем 3 байта, то продолжается выполнение основной части цикла. Сначала в видеопамять записываются коды двух младших байтов точки, а затем код старшего байта. Затем производится переадресация операнда, и команда loop управляет повторами цикла. По окончании цикла восстанавливается исходное содержимое регистра bx и происходит возврат на вызывающий модуль.

Особые точки обрабатывает фрагмент примера, имеющий метку hrln_3. Переход на нее происходит в тех случаях, когда до конца видеосегмента остается меньше четырех байтов. При этом вызывается подпрограмма wrtpnt (см. пример 7.1) для записи кода точки, устанавливается следующее окно видеопамяти и происходит переход на метку hrln_2 для продолжения цикла.

Если вам придется работать с видеокартой, поддерживающей в режимах True Color трехбайтовый код точки, то составьте вариант примера 7.6, работающий с таким кодом. Мы не описываем этот вариант потому, что пока подобные видеокарты не получили широкого распространения.

Замечание

Рисование линий сводится к многократно повторяемой записи одного и того же кода в подряд расположенные адреса видеопамяти. В видеорежимах `direct color` этот код может занимать 2, 3 или 4 байта и от его размера зависит способ записи в видеопамять. Это обстоятельство не позволяет составить одну универсальную подпрограмму для рисования линий в любых видеорежимах VESA. Однако если исключить из рассмотрения трехбайтовый код точки, то можно составить универсальную заготовку подпрограммы, в которой варианты некоторых команд будут выбираться Макроассемблером в зависимости от значения специального признака.

7.3.2. Подпрограммы для построения строк

Строки отличаются от линий тем, что в оперативной памяти хранится их точечный образ. Он может быть получен в результате чтения файла, содержащего рисунок, сохранения изображения находившегося на экране или любым другим способом, для нас это не имеет значения. Однако сам факт существования образа (или заготовки) строки в отличие от ее рисования непосредственно в процессе вывода на экран существенно изменяет структуру подпрограмм.

Если не требуется дополнительных преобразований кодов точек, то построение строки рисунка сводится к пересылке заданного количества байтов из оперативной в видеопамять или в обратном направлении (для сохранения содержимого видеопамяти). Следовательно, возможно составление универсальных подпрограмм, выполняющихся в любом видеорежиме.

Подпрограммы, выполняющие различные манипуляции со строками при работе в видеорежимах `PPG`, описаны в разделе 3.3.1. Здесь мы продолжим эту тему применительно к режимам `direct color`.

Для всех вариантов подпрограммы построения строки мы сохраним то расположение входных параметров в регистрах, которое было принято в разделе 3.3.1. Адрес оперативной памяти (источника) указывается в паре регистров `fs:si`, а адрес видеопамяти (приемника) — в регистре `di`. Предварительно устанавливается окно видеопамяти, которому принадлежит адрес первой точки (указанный в `di`). Регистр `es` должен содержать код сегмента видеобуфера, хранящийся в переменной `Vbuff`.

Исходный вариант подпрограммы. Для дальнейших рассуждений нам нужен простой вариант подпрограммы, иллюстрирующий последовательность действий при построении строки. Он приведен в примере 7.10.

Пример 7.10. Цикл построения строки в режиме Hi-Color

```
drawline: mov   ax, fs:[si]  ; !! для True Color — mov eax, fs[si]
          mov   es:[di], ax  ; !! для True Color — mov es:[di], eax
```

```
          add   si, bytppnt  ; переадресация операнда источника
          add   di, bytppnt  ; переадресация операнда приемника
          jne   @F           ; переход, если не нуль
          call  NxtWin       ; установка следующего окна
@@:       loop  drawline     ; управление повторами цикла
          ret                ; возврат из подпрограммы
```

В комментариях к тексту примера 7.10 показано, как надо изменить две первые команды для того, чтобы подпрограмма `drawline` могла использоваться при работе в режимах `True Color`.

Вариант со строковой операцией. Команды пересылки целесообразно использовать только в тех случаях, когда переадресацию операнда надо отложить на некоторое время или когда приращение адреса не совпадает с размером операнда.

При простом копировании строк размер операнда совпадает с кодом точки (если он не трехбайтовый) и возможна коррекция адреса сразу после записи точки. Поэтому в примере 7.10 команды пересылки имеет смысл заменить строковой операцией. Одна строковая операция заменяет четыре первые команды — две пересылки и две переадресации операндов. Измененный цикл построения строки приведен в примере 7.11. Его можно использовать в тех случаях, когда код точки занимает 2 или 4 байта.

Пример 7.11. Улучшенный цикл построения строки в режиме Hi-Color

```
drawline: movs  word ptr [di], fs:[si]  ; !! movs dword ptr [di], fs:[si]
          or    di, di       ; начало нового сегмента ?
          jne   @F           ; -> нет
          call  NxtWin       ; установка следующего окна
@@:       loop  drawline     ; управление повторами цикла
          ret                ; возврат из подпрограммы
```

Первая команда примера 7.11 переменная, способ ее записи для пересылки 32-разрядных кодов (режим `True Color`) показан в комментарии.

Для использования всех преимуществ строковой операции из цикла надо исключить проверку значений адресов, т. е. пересылать строку по частям так, как это делалось в примере 7.7. В зависимости от видеорежима, основные действия в нем выполняли команды `rep stosw` (режим `Hi-Color`) или `rep stosd` (режим `True Color`). При подстановке в текст примера 7.11 их надо изменить так, как показано ниже.

`rep stosw` заменяется командой `rep movs word ptr [di], fs:[si]`

`rep stosd` заменяется командой `rep movs dword ptr [di], fs:[si]`

Мы не будем приводить измененный вариант примера 7.7, а перейдем к описанию универсального способа пересылки.

Универсальная подпрограмма пересылки. Давайте вернемся к постановке вопроса. Задана строка, содержащая N точек, код каждой из них занимает M байтов. Эту строку надо скопировать из одного места памяти в другое. Мы специально употребили выражение "место памяти", поскольку пересылка из оперативной в видеопамять является частным случаем.

При такой формулировке задачи напрашивается очевидный способ ее решения. Надо вычислить количество байтов в строке (L = N * M) и переслать L байтов из одного места памяти в другое. Обратите внимание, в результате умножения мы избавились от размера кода точки и при составлении подпрограммы учитывается только количество пересылаемых байтов.

Частные случаи решения такой задачи обсуждались уже несколько раз. Наиболее подробно был рассмотрен один из них при описании способов ускорения рисования линии в режимах PPG (примеры 3.8—3.10). Остается собрать указанные примеры в одну подпрограмму, включив в нее вычисление количества байтов в строке. Это и сделано в примере 7.12.

Пример 7.12. Универсальный (цикл пересылки) способ построения строк

```
drawline: push  dx           ; сохранение содержимого регистра dx
          xchg  ax, cx       ; обмен содержимого регистров (ax = N)
          mul   bytppnt      ; dx:ax = L = N * M
          xchg  cx, ax       ; обмен содержимого регистров (cx = L)
          pop   dx           ; восстановление содержимого регистра dx
drawalt:  push  dx           ; сохранение содержимого регистра dx
          mov   dx, di       ; копирование адреса в регистр dx
          add   dx, cx       ; dx = исходный адрес + L
          jc    @F           ; -> прямая расположена в двух окнах
          xor   dx, dx       ; остаток в dx равен нулю
@@:       sub   cx, dx       ; количество байтов в текущем окне
          call  moveto       ; строим первую часть строки
          mov   cx, dx       ; cx = оставшееся количество байтов
          pop   dx           ; восстановление содержимого регистра dx
          or    di, di       ; адрес в пределах текущего окна ?
          jne   d_exit       ; -> да, строка построена полностью
          call  NxtWin       ; установка следующего окна
moveto:   shr   cx, 01       ; преобразуем байты в слова
          jnc   @F           ; -> четное число байтов
          movs  byte ptr [di], fs:[si]     ; пересылка одного байта
@@:       shr   cx, 01       ; преобразуем слова в двойные слова
          jnc   @F           ; -> четное число слов
          movs  word ptr [di], fs:[si]     ; пересылка одного слова
```

```
@@:       je    d_exit       ; -> пересылать больше нечего
          rep   movs dword ptr [di], fs:[si]  ; основной цикл пересылки
d_exit:   ret                                 ; возврат из подпрограммы
```

Выполнение примера 7.12 начинается с вычисления количества байтов в строке. При умножении используются регистры `dx` и `ax`, поэтому содержимое `dx` сохраняется в стеке, а содержимое `ax` — за счет двухкратного использования команды `xchg`. Произведение находится в регистрах `dx:ax`. Мы будем считать, что оно меньше чем 65 536, т. е. `dx` содержит 0. Команда обмена `xchg` помещает результат в `cx`, одновременно восстанавливая исходное состояние `ax`, а из стека выталкивается исходное содержимое регистра `dx`.

Если количество байтов в строке известно заранее, то заново вычислять его не имеет смысла. Оно указывается в регистре `cx`, а для вызова подпрограммы используется вторая точка входа, имеющая имя `drawalt`.

Команда с меткой `drawalt` сохраняет в стеке содержимое регистра `dx`, затем в него копируется адрес видеопамяти, который увеличивается на размер строки в байтах. Если при сложении произойдет переполнение (установка C-разряда), то команда `jc @F` исключает очистку `dx`. В противном случае строка помещается в текущем окне и регистр `dx` очищается. Затем вычисляется количество выводимых точек, и подпрограмма `moveto` строит первую часть строки.

После построения первой части строки в регистр `cx` копируется содержимое `dx` (остаток строки). Регистр `dx` освободился и надо восстановить его исходное состояние. Для выбора дальнейших действий проверяется текущий адрес в регистре `di`, если он отличен от нуля, то построение строки завершено и выполняется команда `ret`. В противном случае устанавливается следующее окно видеопамяти, и подпрограмма `moveto` строит остаток линии. После ее выполнения завершится работа основной подпрограммы, т. к. в верхушке стека находится адрес возврата на вызывающий модуль.

В подпрограмме `moveto` команда `rep movs dword ptr [di], fs:[si]` является основной, она пересылает группу 32-разрядных слов. Однако количество байтов в строке не обязательно кратно четырем. Поэтому нужна предварительная проверка и пересылка от одного до трех "лишних" байтов, так чтобы остаток был кратен четырем.

Команда, имеющая метку `moveto`, сдвигает содержимое регистра `cx` на разряд вправо. Если оно было нечетным, то пересылается первый байт строки, в противном случае `jnc @F` исключает эту пересылку.

Затем содержимое `cx` еще раз сдвигается на разряд вправо. Если оно было нечетным, то пересылается слово (два байта строки), в противном случае `jnc @F` исключает эту пересылку.

В результате выполнения двух сдвигов и пересылки "лишних" байтов в строке остается целое число 32-разрядных слов, количество которых находится

в регистре cx. Если оно равно нулю, то произойдет переход на команду ret, в противном случае выполняется микропрограммный цикл копирования 32-разрядных слов. После этого выполняется команда ret.

Вторая точка входа drawalt введена для тех случаев, когда известен размер строки в байтах. Приведем несколько примеров таких случаев. При работе в режимах PPG количество байтов совпадает с количеством точек. Если размер строки равен ширине экрана, то ее размер в байтах указывает переменная bperline и вычислять его нет смысла. При построении рисунка, содержащего большое количество строк, целесообразно один раз вычислить размер строки в байтах, а не повторять одни и те же вычисления при построении каждой строки. Наконец, возможны также случаи, когда количество байтов вычисляется нестандартным способом, например, зависит от определенных условий.

Обсуждение результатов. Описанная подпрограмма не только не зависит от размера кода точек, но и затрачивает минимально возможное время на построение строки, что особенно важно при работе в режимах direct color. Поэтому мы советуем использовать именно ее в тех случаях, когда возможна простая пересылка строк графических объектов.

Подпрограмма может записывать в видеопамять не более чем 65 536 байтов. Это ограничение связано с тем, что источник или приемник расположен в оперативной памяти. Мы исходили из предположения, что он полностью помещается в одном сегменте, и поэтому исключили контроль адресов. Если же это условие нарушено, то в подпрограмму придется добавить контроль адресов и переключение сегментов оперативной памяти. Способ переключения сегментов зависит от того, в какой части оперативной памяти они расположены (см. приложение Б).

Если при работе со строкой выполняется не простое копирование, а более сложные действия, то для сокращения количества проверок адресов видеопамяти можно использовать работу с двумя частями строки. Алгоритм работы с двумя частями приведен в примере 7.6 (подпрограмма Twopart). Для выполнения конкретных действий надо составить подпрограмму baselp, которая в примере 7.6 вызывается для обработки каждой из двух частей строки. Например, такая подпрограмма может считывать код очередной точки, как-то обрабатывать его и возвращать результат на старое место. Вопрос о целесообразности работы с двумя частями строки решается с учетом конкретных особенностей алгоритма работы с графическим объектом.

7.3.3. Работа с прямоугольными областями

В данном разделе нас будет интересовать многофункциональная подпрограмма, способная выполнять различные манипуляции с графическими объектами прямоугольной формы. Ее составление возможно при условии, что

требуемые действия выполняют специализированные вспомогательные подпрограммы.

Переадресация строк. При работе с графическими объектами после обработки каждой строки надо вычислять адрес начала следующей. В разделе 7.2 было рекомендовано использовать для этого константу коррекции адреса видеопамяти, которая вычисляется по следующей формуле:

```
offsline = bperline – widthrect*bytppnt = (horsize – widthrect)*bytppnt.
```

Как видно из этой формулы, значение `offsline` зависит от видеорежима (переменные `Horsize` и `bytppnt`) и от ширины объекта (переменная `widthrect`), поэтому его приходится вычислять в каждом конкретном случае.

При работе в видеорежимах `PPG` команды для вычисления значения `offsline` (пересылка в и вычитание) мы включали в тексты примеров.

При работе в видеорежимах `direct color` увеличивается не только количество команд, вычисляющих значение `offsline`, но и количество регистров, в которых расположены операнды этих команд. Поэтому имеет смысл составить короткую подпрограмму, выполняющую необходимые вычисления и учитывающую характеристики установленного видеорежима.

Варианты подпрограммы calloffs. В примере 7.13 показаны два варианта подпрограмм, вычисляющие значение `offsline` с использованием команд умножения или сдвигов. Входным параметром является значение переменной `widthrect`, которое указывается в регистре `dx`. Этот регистр выбран потому, что во всех примерах он использовался для указания ширины прямоугольной области. Для совместимости с ранее приведенными примерами результат вычислений находится в регистре `bx`.

Пример 7.13. Варианты подпрограмм для вычисления offsline

```
;         Вариант 1 – вычисление offsline с использованием сдвигов
calloffs: push  cx           ; сохранение содержимого cx
          mov   cx, wrdppnt  ; cx = величина сдвига
          mov   bx, horsize  ; bx = ширина экрана в точках
          sub   bx, dx       ; bx = horsize – widthrect
          shl   bx, cl       ; bx = (horsize – widthrect)*bytppnt
          pop   cx           ; восстановление содержимого cx
          ret                ; возврат из подпрограммы
;         Вариант 2 – вычисление offsline с использованием умножения
calloffs: push  dx           ; сохранение содержимого dx
          xchg  ax, bx       ; обмен содержимого регистров ax, bx
          mov   ax, horsize  ; ax = ширина экрана в точках
          sub   ax, dx       ; ax = horsize – widthrect
          mul   bytppnt      ; ax = (horsize – widthrect)*bytppnt
```

```
        xchg  ax, bx        ; обмен содержимого регистров ax, bx
        pop   dx            ; восстановление содержимого dx
        ret                 ; возврат из подпрограммы
```

Первый вариант подпрограммы примера 7.13 короче на одну команду и выполняется немного быстрее второго, но его можно использовать только в тех случаях, когда код точки занимает 1, 2 или 4 байта.

Второй вариант длиннее первого на одну команду и выполняется немного дольше, но его можно использовать при любом размере кода точки. Выбор конкретного варианта подпрограммы остается за вами.

Пересылка в видеопамять. При работе с графикой достаточно часто приходится сохранять и восстанавливать содержимое видеопамяти. Это делается, например, при каждом перемещении курсора.

В примере 7.14 приведен текст подпрограммы, выполняющей копирование содержимого оперативной памяти в видеопамять. Входными параметрами подпрограммы являются размер прямоугольной области и адреса операндов источника и приемника. Ширина прямоугольной области помещается в регистр `dx`, а высота в регистр `cx`. Адрес оперативной памяти (источника) указывается в регистрах `fs:si`. Адрес видеопамяти (приемника) помещается в регистр `di` и устанавливается окно видеопамяти, которому принадлежит этот адрес. В регистре `es` должен находиться код видеосегмента (`A000h`).

Пример 7.14. Подпрограмма пересылки из оперативной в видеопамять

```
Rstreg: PushReg <bx, cx, di, si, Cur_win> ; сохранение в стеке
        call  calloffs      ; вычисление константы offsline
mvsr:   push  cx            ; сохранение значения счетчика строк
        mov   cx, dx        ; задание количества точек в строке
        call  drawline      ; копируем очередную строку
        add   di, bx        ; адрес начала следующей строки
        jnc   @F            ; -> адрес в пределах окна
        call  Nxtwin        ; установка следующего окна
@@:     pop   cx            ; восстановление счетчика строк
        loop  mvsr          ; управление повторами цикла
        PopReg <Cur_win, si, di, cx, bx>  ; восстановление из стека
        call  setwin        ; восстановление исходного окна
        ret                 ; возврат из подпрограммы
```

Текст примера не требует особых пояснений — подобные циклы мы описывали неоднократно, например, в разделе 3.3.2 (подпрограмма `draw`). Поговорим о том, что явно не следует из текста.

Зависимость от установленного видеорежима в данном примере скрыта в подпрограммах `calloffs` и `drawline`. Если вы выберете второй вариант под-

программы calloffs примера 7.13 и подпрограмму drawline, текст которой описан в примере 7.12, то подпрограмма Rstreg будет выполняться в любом видеорежиме, независимо от размера кода точки.

Размер прямоугольной области, выраженный в байтах, не должен превышать размера стандартного сегмента памяти, т. е. 65 536 байтов. Это ограничение связано с тем, что пересылаемые данные находятся в оперативной памяти, которая сегментирована так же, как и видеопамять, а в примере 7.14 отсутствует контроль значения адресов оперативной памяти.

Способы контроля значений адресов оперативной и видеопамяти ничем не отличаются друг от друга, но существенно различаются способы переключения сегментов, которые зависят еще и от типа оперативной памяти. Они подробно описаны в приложении Б данной книги. Там же приведен пример подпрограммы, выполняющей сохранение или восстановление содержимого всего пространства видеопамяти отображаемого на экране (см. примеры Б.7 и Б.8).

Пересылка из видеопамяти. Для сохранения исходного содержимого видеопамяти производится его копирование (пересылка) в оперативную память. Нас интересуют универсальные процедуры пересылки, основанные на применении строковых операций. Однако у строковых операций фиксированы регистры, содержащие адреса операндов источника и приемника. Поэтому при пересылке в обратном направлении в es:di должен находиться адрес оперативной памяти, а в fs:si — адрес видеопамяти. Для удобства лучше сохранить единообразный способ указания адресов в регистрах и изменять его в подпрограмме на время пересылки.

Заметим, что и после перестановки адресов использовать подпрограмму drawline из примера 7.12 нельзя. При ее составлении предполагалось, что адрес видеопамяти находится в регистре di, а он оказался в регистре si. Поэтому надо сделать копию примера 7.12, присвоить ей новое имя, например saveline, и заменить в двух командах копии имя регистра di именем регистра si. В результате получится универсальная подпрограмма, выполняющая сохранение строки видеопамяти в оперативной памяти.

Подпрограмма Savereg. В примере 7.15 показано, как можно переставить адреса операндов на время выполнения цикла пересылки, а затем восстановить их исходное расположение в регистрах. Входные параметры в данном случае задаются так же, как для примера 7.14.

Пример 7.15. Подпрограмма пересылки из видеопамяти в оперативную

```
Savereg: PushReg <bx, cx, di, si, es, Cur_win>    ; сохранение в стеке
         mov   bx, fs       ; копируем код сегмента из fs в bx
         mov   es, bx       ; копируем код сегмента из bx в es
         mov   fs, Vbuff    ; fs = сегмент видеобуфера
```

```
          xchg  di, si       ; перестановка адресов di и si
          call  calloffs     ; вычисление константы offsline
svrg:     push  cx           ; сохранение значения счетчика строк
          mov   cx, dx       ; задание количества точек в строке
          call  saveline     ; копируем очередную строку
          add   si, bx       ; адрес начала следующей строки
          jnc   @F           ; -> адрес в пределах окна
          call  Nxtwin       ; установка следующего окна
@@:       pop   cx           ; восстановление счетчика строк
          loop  svrg         ; управление повторами цикла
          push  es           ; помещаем код сегмента из es в стек
          pop   fs           ; и выталкиваем его из стека в fs
          PopReg <Cur_win, es, si, di, cx, bx> ; восстановление из стека
          call  setwin       ; восстановление исходного окна
          ret                ; возврат из подпрограммы
```

Напомним, что команда `xchg` не работает с сегментными регистрами, а у команды `mov` только один операнд может быть именем сегментного регистра. Поэтому для пересылки содержимого одного сегментного регистра в другой приходится использовать либо регистр-посредник, либо стек. Оба этих способа показаны в примере 7.15.

Основной цикл пересылки примера 7.15 имеет имя `svrg`, он отличается от аналогичного цикла примера 7.14 (`mvsr`) только одной командой. При вычислении адреса следующей строки константа коррекции прибавляется к содержимому регистра `si`, а не `di`, как это делалось в примере 7.14.

Заливка прямоугольной области. Изменим текст примера 7.14 так, чтобы его можно было использовать для окрашивания прямоугольной области заданным цветом. В этом случае при выполнении цикла должна вызываться подпрограмма `horline`, а не `drawline`. Регистр `si` не используется, поэтому его имя исключается из списков `PushReg` и `PopReg`.

Измененный текст подпрограммы показан в примере 7.16. При ее вызове код цвета указывается в регистрах `ax` или `eax` (в зависимости от видеорежима). Ширина прямоугольной области задается в регистре `dx`, а высота — в `cx`. Адрес видеопамяти для левого верхнего угла должен находиться в регистре `di`, регистры `fs:si` не используются. Предполагается, что `es` содержит код видеосегмента и установлено исходное окно видеопамяти.

Пример 7.16. Окрашивание прямоугольной области заданным цветом

```
Fillreg: PushReg <bx, cx, di, Cur_win>     ; сохранение в стеке
         call  calloffs      ; вычисление константы offsline
fillrg:  push  cx            ; сохранение значения счетчика строк
         mov   cx, dx        ; задание количества точек в строке
```

```
        call  horline       ; рисуем очередную строку
        add   di, bx        ; адрес начала следующей строки
        jnc   @F            ; -> адрес в пределах окна
        call  Nxtwin        ; установка следующего окна
@@:     pop   cx            ; восстановление счетчика строк
        loop  fillrg        ; управление повторами цикла
        PopReg <Cur_win, di, cx, bx>      ; восстановление из стека
        jmp   setwin        ; установка окна и выход
```

В отличие от примеров 7.14 и 7.15, в данном случае размер закрашиваемой области экрана не ограничен, лишь бы хватило памяти, установленной на видеокарте. Например, для окрашивания всей рабочей поверхности экрана в нужный цвет, перед вызовом подпрограммы код цвета помещается в регистре `ax` (или `eax`), в `cx` копируется переменная `Versize`, а в `dx` — `Horsize`, регистр `di` очищается и устанавливается нулевое окно видеопамяти.

Текст примера 7.16 не зависит от видеорежима, но в нем вызывается подпрограмма `horline`, в тексте которой есть переменные команды, зависящие от видеорежима (см. раздел 7.3.1). Поэтому, в отличие от подпрограмм пересылки, подпрограмма `Fillreg` *не является универсальной.*

Многофункциональная подпрограмма. Основное различие текстов примеров 7.14 и 7.16 заключается в имени вспомогательной подпрограммы, используемой для выполнения конкретных действий.

Для того чтобы приведенная в примере 7.14 подпрограмма стала многофункциональной, команду `call drawline` надо заменить командой `call bp`, а перед вызовом указывать в регистре `bp` адрес вспомогательной подпрограммы. Аналогичный прием описан в разделе 3.3.2, на примере подпрограммы `draw`. Там же показано, как формируется адрес в регистре `bp`.

После указанного изменения подпрограмму `Rstreg` можно использовать, например, для построения рисунков, изменения их цвета, окрашивания прямоугольной области и любых других действий, которые способны выполнять вызываемые вспомогательные подпрограммы.

7.4. Рисунки, использующие палитру

Полноцветные и подготовленные с применением палитры рисунки имеют разное назначение. В режимах `direct color` графические задачи должны "уметь" работать с любыми рисунками, независимо от способа их подготовки.

Если рисунок подготовлен с использованием палитры, то коды его точек являются порядковыми номерами строк таблицы цветов (палитры), хранящейся в файле вместе с образом рисунка. При построении таких рисунков в режимах `direct color` требуется преобразование кода каждой точки в код ее цвета, который затем записывается в видеопамять.

Количество перечисленных в палитре цветов, как правило, меньше количества точек в рисунке и это различие тем больше, чем больше размеры рисунка. Поэтому имеет смысл преобразовать исходную палитру в форму, упрощающую перекодировку точек при построении рисунка. Это не только упростит действия, выполняемые в соответствующих подпрограммах, но и сделает их независящими от формата исходной палитры.

Употребляя выражение "формат", мы имеем в виду количество байтов в строке палитры, и порядок расположения кодов базовых цветов в этих байтах. Формат зависит от стандарта хранения графических данных, которому соответствует файл, содержащий образ рисунка.

В большинстве стандартов используется формат `rgb`, впервые он был применен в стандарте `PCX`. В формате `rgb` строка состоит из трех байтов, в которых хранятся коды красного (`r`), зеленого (`g`) и синего (`b`) базовых цветов, расположенные в указанной последовательности.

Единственный в своем роде, но широко распространенный стандарт `BMP` для Windows предусматривает хранение палитры в формате `bgr0`. В этом случае строка палитры состоит из четырех байтов. В трех первых хранятся коды синего (`b`), зеленого (`g`) и красного (`r`) базовых цветов, расположенные в указанной последовательности, последний байт резервный, обычно он очищен.

В модифицированном стандарте `BMP` для OS/2 из строки палитры исключен резервный байт, а в трех оставшихся байтах базовые цвета расположены в формате `bgr`.

При преобразовании строки палитры в код цвета точки надо учитывать не только формат строки, но и расположение цветов в коде точки, которое зависит от используемого режима `direct color`. Поэтому данный раздел делится на три подраздела, в двух первых описаны подпрограммы, предназначенные для применения в режимах `Hi-Color` и `True Color`, а в третьем обсуждаются способы построения рисунков в обоих режимах.

7.4.1. Преобразование палитры в форматы Hi-Color

Для того чтобы с палитрой можно было работать, ее надо прочитать из файла, содержащего образ рисунка, в оперативную память. Расположение палитры в файле и ее размеры зависят от стандарта хранения графических данных, которому соответствует выбранный вами файл. Никакого единообразия тут нет, но всегда остается возможность преобразовать файл в тот стандарт, с которым вы предпочитаете работать. Большинство графических редакторов позволяют сделать такое преобразование.

Способы чтения палитры файлов стандартов `BMP` и `PCX` описаны, соответственно, в приложении А и в разделе 4.4. В обоих случаях мы советовали раз-

мещать прочитанную палитру в буфере обмена, сегмент которого хранится в переменной SwpSeg, расположенной в разделе данных задачи.

Независимо от стандарта, для определения размера и местонахождения палитры в файле надо, прежде всего, прочитать и проанализировать его заголовок. При чтении заголовок помещается в буфер обмена, поэтому к моменту начала обработки палитры в регистре fs находится код сегмента (копия переменной SwpSeg), а в si — адрес начала палитры в памяти.

При анализе заголовка файла определяются две важные величины — количество строк (цветов) в палитре и ее размер. Напомним, что он в три или в четыре раза больше количества строк. Размер нужен при чтении палитры, а количество строк — при ее преобразовании в таблицу цветов. Перед вызовом описываемых ниже подпрограмм количество строк указывается в регистре cx.

После чтения палитры вызываются описываемые ниже подпрограммы. Они формируют таблицу (одномерный массив), содержащую коды цветов, формат которых соответствует установленному видеорежиму. Для хранения таблицы надо выделить пространство оперативной памяти. Учитывая, что количество цветов в исходной палитре не больше 256-ти, в режимах Hi-Color размер указанного пространства памяти составляет 256 слов (512 байтов).

Расположение таблицы цветов. Таблица цветов нужна только при первом построении рисунка, поэтому отводить для ее хранения постоянное место в памяти не целесообразно. В подобных случаях, обычно, рекомендуется выделять память только на время ее использования. В разделе 3.3.3 мы специально создавали буфер общего назначения, пространство которого доступно различным подпрограммам. Например, начало этого буфера использовалось для сохранения исходного фона на месте расположения информационной строки. Этот буфер и можно применять для временного размещения таблицы цветов.

Напомним, смещение и сегмент буфера общего назначения хранятся в следующих переменных, которые должны быть описаны в разделе данных:

```
GenOffs  dw 0  ; адрес (смещение) в буфере общего назначения
GenSeg   dw 0  ; сегмент, содержащий буфер общего назначения
```

Способы выделения пространства для размещения буфера описаны в приложении Б данной книги. После его выделения становится известным значение сегмента, которое задача должна сохранить в GenSeg. Исходное значение GenOffs равно нулю, а текущее значение зависит от того, какая часть буфера используется в данный момент времени.

Все приведенные ниже *подпрограммы* используют следующие входные параметры:

- адрес начала преобразуемой палитры указан в регистрах fs:si;
- размер палитры (в виде количества строк) в регистре cx;

- адрес начала формируемой таблицы цветов указывают переменные `GenSeg` и `GenOffs`;
- результат преобразований помещается в таблицу цветов.

Теперь о преобразованиях. Формируемый код цвета зависит от установленного видеорежима `Hi-Color`. Существует две разновидности этих режимов. В одном случае код цвета занимает 15, а в другом 16 разрядов, расположение базовых цветов для обоих случаев показано в табл. 7.1. Мы рассмотрим два варианта подпрограмм, формирующих 15-разрядный код цвета, и обсудим, как сформировать 16-разрядный код цвета.

Палитра формата rgb. В этом случае базовые цвета в строке палитры и в формируемом коде расположены в одинаковой последовательности. Поэтому коды базовых цветов, считанные из строки палитры, надо сокращать до пяти разрядов и помещать в формируемый код цвета. Для того чтобы они оказались в нужном месте, перед добавлением кода очередного цвета формируемый код сдвигается на 5 разрядов влево. В примере 7.17 показан способ выполнения этих действий.

Пример 7.17. Преобразование палитры rgb в 15-разрядный код

```
cnvpal:  PushReg <ax,bx,cx,di,si,es>; сохранение содержимого регистров
         les   di, dword ptr GenOffs; es:di = адрес таблицы цветов
modcol:  mov   al, fs:[si]      ; читаем код красного цвета в al
         shr   al, 03           ; сокращаем его до 5-ти разрядов
         mov   bh, fs:[si+1]    ; читаем код зеленого цвета в bh
         shld  ax, bx, 05       ; !! или shld ax, bx, 06
         mov   bh, fs:[si+2]    ; читаем код синего цвета в bh
         shld  ax, bx, 05       ; сдвигаем и дополняем код в ax
         add   si, 03           ; адрес следующей строки палитры
         stosw                  ; записываем новый код цвета
         loop  modcol           ; управление повторами цикла
         PopReg <es,si,di,cx,bx,ax> ; восстановление регистров
         ret                    ; возврат из подпрограммы
```

В примере 7.17 основную роль играет команда `shld`, напомним ее особенности. При выполнении команды содержимое первых двух операндов, в данном случае это регистры `ax` и `bx`, сдвигается как одно 32-разрядное слово, но записывается только старшая часть результата в первый операнд, а содержимое второго операнда не изменяется. Таким образом, в примере 7.17 команда `shld` сдвигает содержимое регистра `ax` на пять разрядов влево и записывает в освободившееся место пять старших разрядов регистра `bx`.

В зависимости от установленного видеорежима код зеленого цвета может занимать 5 или 6 разрядов. В комментарии показано, как изменится одна из команд `shld` в случае формирования 16-разрядного кода цвета.

Устаревший формат палитры PCX. В разделе 4.4 говорилось о том, что существует устаревший формат 256-цветной палитры, поддерживаемый версией стандарта PCX, разработанной фирмой Genius. Если вам надо работать с файлом, соответствующим этому стандарту, то проще всего преобразовать его в основной стандарт фирмы ZSoft. Это можно сделать, например, с помощью графического редактора PhotoFinish фирмы ZSoft.

Палитры формата bgr и bgr0. В этом случае базовые цвета в строке палитры и в формируемом коде расположены в противоположном порядке, коды красного и синего цветов переставлены местами. Есть два способа учесть эту особенность. Байты палитры можно считывать по порядку, а перед добавлением очередного цвета сдвигать формируемый код вправо, а не влево, как это делалось раньше. Другой способ заключается в том, чтобы считывать байты строки палитры в обратном порядке, а формируемый код по-прежнему сдвигать влево. В примере 7.18 реализован второй способ формирования кода цвета.

Пример 7.18. Преобразование палитры bgr в 15-разрядный код

```
cnvpal:  PushReg <ax,bx,cx,di,si,es>; сохранение содержимого регистров
         les   di, dword ptr GenOffs; es:di = адрес таблицы цветов
modcol:  mov   al, fs:[si+2] ; читаем код красного цвета в al
         shr   al, 03        ; сокращаем его до 5-ти разрядов
         mov   bh, fs:[si+1] ; читаем код зеленого цвета в bh
         shld  ax, bx, 05    ; !! или shld ax, bx, 06
         mov   bh, fs:[si]   ; читаем код синего цвета в bh
         shld  ax, bx, 05    ; сдвигаем и дополняем код в ax
         add   si, 03        ; !! для формата "bgr0" — add si, 04
         stosw               ; записываем новый код цвета
         loop  modcol        ; управление повторами цикла
         PopReg <es,si,di,cx,bx,ax> ; восстановление регистров
         ret                 ; возврат из подпрограммы
```

Если вы сравните тексты примеров 7.17 и 7.18, то обнаружите, что в них различаются только индексные выражения у первой и третьей команд пересылки. За счет такого трюка мы считывает базовые цвета в том порядке, в котором они должны располагаться в формируемом коде.

В примере 7.18 появилась вторая переменная команда, выполняющая переадресацию строк палитры. При работе с форматом bgr значение адреса увеличивается на 3, а с форматом bgr0 — на 4. На практике нет никакой необходимости работать с переменной командой, ее второй операнд просто не должен зависеть от размера строки палитры.

Палитры форматов bgr и bgr0 используются в двух разных версиях стандарта BMP. Заголовки файлов обрабатываются по единой схеме, не зависящей от

версии (см. приложение А данной книги). При обработке заголовка обязательно определяется размер строки палитры т. к. он нужен при всех манипуляциях с палитрой. Поэтому вторым операндом команды `add si, 03` должно быть не значение константы, а имя регистра или переменной, в котором (в которой) хранится размер строки палитры.

Универсальный вариант подпрограммы. В обоих приведенных примерах от установленного режима `Hi-Color` зависит третий операнд одной из команд `shld`. От переменной команды можно избавиться следующим способом.

При выполнении задачей подготовительных действий обязательно считывается массив `Info`, содержащий наиболее важные характеристики установленного видеорежима. В частности, в байте этого массива со смещением `21h` указан размер кода зеленого цвета. В режимах `Hi-Color` там находится число 5 или 6. Это число надо сохранить в специально выделенном байте, присвоив ему удобное для вас имя, например `gcolsize`, и использовать в подпрограммах примеров 7.17 и 7.18 при обработке зеленого цвета.

Причем величина сдвига может быть либо явно указана в записи команды, либо помещена в регистр `cl`, который применяется в качестве первого операнда команды. В примерах 7.17 и 7.18 регистр `cx` используется в качестве счетчика, поэтому понадобятся две дополнительные команды для сохранения и восстановления его содержимого. Таким образом, одна переменная команда `shld` может быть заменена следующей группой команд:

```
push    cx              ; сохраняем счетчик повторов цикла
mov     cl, gcolsize    ; cl = величина сдвига (5 или 6)
shld    ax, bx, cl      ; сдвиг на величину, указанную в cl
pop     cx              ; восстанавливаем счетчик повтором цикла
```

Учитывая, что цикл преобразований повторяется не более 256-ти раз, дополнительные потери времени будут не столь велики, зато подпрограмма окажется применимой во всех видеорежимах `Hi-Color`.

7.4.2. Преобразование палитры в форматы True Color

В данном разделе описаны две подпрограммы, предназначенные для преобразования кодов цветов палитры в формат, соответствующий режимам `True Color`. При установке этих видеорежимов код базового цвета занимает 1 байт, поэтому сокращение размеров кодов цветов палитры не требуется, что упрощает выполняемые в подпрограммах действия.

Мы предполагаем, что вы внимательно прочитали, по крайней мере, начало предыдущего раздела и знаете, как задаются входные параметры перед вызовом подпрограмм, где расположены исходная палитра и формируемая таблица цветов. Отметим только, что при работе в видеорежимах `True Color`

размер таблицы цветов увеличивается до 1024 байтов (1 Кбайт), поскольку формируемый код цвета занимает 32 разряда (двойное слово).

Палитра формата bgr0 не требует никаких преобразований, поскольку расположение базовых цветов и резервного байта полностью соответствует режимам `True Color` (см. табл. 7.2 и 7.3). В этом случае палитру из файла надо просто прочитать в буфер общего назначения, а не в буфер обмена, как мы это обычно делали.

Замечание

Напомним, что чтение из файла выполняет функция `3Fh` прерывания `int 21h`. Перед ее вызовом адрес для размещения прочитанных данных указывается в регистрах `ds:dx`. Мы использовали для чтения подпрограмму `readf`, описанную в примере 3.23, которая загружала в эти регистры сегмент и смещение буфера обмена. Для чтения в любое указанное место памяти она не предназначена, поэтому вам придется составить аналогичный вариант подпрограммы, выполняющей чтение из файла в буфер общего назначения.

Палитра формата bgr отличается от формата `bgr0` отсутствием в ее строках пустого (резервного) байта. Поэтому формирование кода цвета сводится к копированию базовых цветов из палитры и добавлению пустого байта. Текст подпрограммы приведен в примере 7.19.

Пример 7.19. Преобразование палитры bgr в 32-разрядный код

```
cnvpal:  PushReg <ax,cx,di,si,es>    ; сохранение содержимого регистров
         les   di, dword ptr GenOffs ; es:di = адрес таблицы цветов
modcol:  movs  word ptr [di], fs:[si]; копируем 2 младших байта
         lods  byte ptr fs:[si]   ; читаем в регистр al третий байт
         xor   ah, ah             ; очищаем старший байт регистра ax
         stosw                    ; записываем 2 старших байта
         loop  modcol             ; управление повторами цикла
         PopReg <es,si,di,cx,ax> ; восстановление регистров
         ret                      ; возврат из подпрограммы
```

В цикле формирования кода примера 7.19 из палитры в таблицу цветов сначала копируется первое слово, содержащее коды синего и зеленого цветов. Затем в байт `al` считывается из палитры код красного цвета, а байт `ah` очищается. В результате в регистре `ax` оказывается старшая часть 32-разрядного кода цвета. Следующая команда `stosw` записывает содержимое `ax` в таблицу цветов. Код очередного цвета сформирован и записан, а поскольку для чтения и записи в цикле использованы строковые операции, то дополнительные команды переадресации не требуются. Последняя команда `loop` управляет повторами цикла.

Палитра формата rgb. В этом случае в строках палитры не только отсутствует пустой (резервный) байт, но и базовые цвета расположены в обратном

порядке, по сравнению с форматом bgr. Поэтому, в отличие от примера 7.19, их простое копирование не возможно — требуется изменение расположения цветов. В примере 7.17 для расположения базовых цветов в нужной последовательности мы использовали сдвиг формируемого кода перед добавлением очередного цвета. Этот прием применим и в данном случае, как выполняется подобное преобразование, показано в примере 7.20.

Пример 7.20. Преобразование палитры rgb в 32-разрядный код

```
cnvpal:  PushReg <ax,cx,di,si,es>   ; сохранение содержимого регистров
         les   di, dword ptr GenOffs; es:di = адрес таблицы цветов
modcol:  xor   ah, ah           ; очистка регистра ah
         lods  byte ptr fs:[si] ; чтение кода красного цвета
         shl   eax, 16          ; сдвиг содержимого eax влево на слово
         lods  word ptr fs:[si] ; чтение зеленого и синего цвета
         xchg  ah, al           ; перестановка синего и зеленого цвета
         stosd                  ; запись строки в таблицу цветов
         loop  modcol           ; управление повторами цикла
         PopReg <es,di,cx,ax>   ; восстановление регистров
         ret                    ; возврат из подпрограммы
```

Выполнение цикла modcol в примере 7.20 начинается с очистки регистра ah. В результате, после сдвига на 16 разрядов, окажется очищенным резервный байт в формируемом коде цвета. Вторая команда тела цикла считывает в регистр al код красного цвета, и в регистре ax оказывается старшая часть формируемого кода.

Она сдвигается на 16 разрядов и оказывается в старшей половине регистра eax, а в младшую считываются коды зеленого и синего цветов. Для того чтобы содержимое регистра eax соответствовало табл. 7.2, эти цвета надо поменять местами, что и делает команда xchg. Сформированный код записывает в таблицу цветов команда stosd. Для чтения и записи в цикле использованы строковые операции, поэтому команды переадресации не нужны. Сразу после stosd выполняется команда loop, управляющая повторами цикла.

7.4.3. Построение рисунков с использованием палитры

В разделе 3.3.2 были подробно описаны способы построения не сжатых рисунков, в образах которых точки расположены в естественном порядке. При этом мы различали рисунки ограниченного и произвольного размера. Соответствующие варианты подпрограмм приведены в примерах 3.21 и 3.22, они предназначены для выполнения в режимах PPG. В данном разделе описаны

такие варианты этих примеров, которые можно использовать в любых видеорежимах.

Построение строки рисунка. Для построения строки рисунка в примерах 3.21 и 3.22 вызывалась подпрограмма `drawline`. Большинство описанных ранее ее вариантов выполняет копирование кодов точек образа рисунка в видеопамять. Исключением являются примеры 2.17 и 2.18, в которых при построении строки выполняется распаковка 16- и 2-цветных рисунков. В данном случае нам нужен вариант подпрограммы, выполняющий перед записью в видеопамять перекодирование точек образа рисунка по таблице цветов.

Текст подпрограммы приведен в примере 7.21. Перед ее вызовом в регистрах указываются следующие данные: `cx` — размер строки, `di` — адрес ее первой точки в видеопамяти, `fs:si` — адрес начала рисунка в оперативной памяти, `gs` — сегмент оперативной памяти, содержащий таблицу цветов. Адрес начала таблицы в этом сегменте подпрограмма выбирает сама из переменной `GenOffs`. Как обычно, предварительно надо установить окно видеопамяти, к которому относится указанный в `di` адрес, а в регистре `es` должен находиться код видеосегмента.

Пример 7.21. Построение строки с перекодированием по таблице цветов

```
drawline: push  eax          ; сохранение содержимого eax
drwlin:   lods  byte ptr fs:[di]    ; al = код точки образа рисунка
          and   eax, 0FFh    ; очистка старших разрядов eax
          shl   ax, wrdppnt  ; учет размера строки таблицы
          add   ax, GenOffs  ; ax = смещение начала таблицы
          mov   ax, gs:[eax] ; !! или mov eax, gs:[eax] для True Color
          stosw              ; !! или stosd для True Color
          or    di, di       ; достигнута граница окна ?
          jne   @F           ; -> нет, обход следующей команды
          call  NxtWin       ; установка следующего окна
@@:       loop  drwlin       ; управление повторами цикла
          pop   eax          ; восстановление содержимого eax
          ret                ; возврат из подпрограммы
```

В цикле примера 7.21 регистр `eax` используется для указания адреса при чтении кода цвета точки. Адрес формируется в младшем слове регистра, а его старшее слово должно быть очищено. Поэтому после чтения в `al` кода образа точки старшие разряды регистра `eax` очищаются с помощью операции "конъюнкция". В зависимости от установленного видеорежима, третья команда сдвигает содержимое `ax` на 1 или 2 разряда влево. Остается прибавить к `ax` смещение таблицы в сегменте `gs`, и адрес требуемого кода будет сформирован. Код считывается в регистр `ax` (или в `eax`) и оттуда записывается в видеопамять с помощью строковой операции `stosw` или `stosd`. Далее,

как обычно, проверяется адрес видеопамяти и в случае необходимости устанавливается следующее окно. Повторами цикла управляет команда `loop`.

В тексте примера две команды зависят от видеорежима, комментарий к ним начинается с двух восклицательных знаков. Избавиться от переменных команд невозможно, но для выбора их нужной пары можно использовать директивы условного ассемблирования (см. пример 7.8).

Упрощение подпрограммы. В примере 7.22 приведен текст упрощенного варианта подпрограммы, выполняющей перекодировку и запись одной точки, код которой указан в регистре `al`.

Пример 7.22. Перекодировка по таблице и запись точки в видеопамять

```
wrtpnt: push  eax           ; сохранение содержимого eax
        and   eax, 0FFh     ; очистка старших разрядов eax
        shl   ax, wrdppnt   ; учет размера строки таблицы
        add   ax, GenOffs   ; ax = смещение начала таблицы
        mov   ax, gs:[eax]  ; !! или mov eax, gs:[eax] для True Color
        stosw               ; !! или stosd для True Color
        pop   eax           ; восстановление содержимого eax
        ret                 ; возврат из подпрограммы
```

Построение небольшого рисунка. Подпрограмма построения рисунка, образ которого помещается в одном сегменте, приведена в примере 7.23.

Перед ее вызовом устанавливается окно видеопамяти, содержащее левый верхний угол рисунка, а адрес этого угла помещается в регистр `di`. Адрес начала образа рисунка в оперативной памяти записывается в регистры `fs:si`. Его ширина и высота указываются, соответственно, в `dx` и `cx`. Адрес начала таблицы подпрограмма выбирает из переменных `GenSeg` и `GenOffs`. Как обычно, должно быть установлено окно видеопамяти, к которому относится указанный в `di` адрес, а регистр `es` должен содержать код видеосегмента.

Пример 7.23. Построение рисунка из файла небольшого размера

```
drawing: PushReg <di,si,bx,cx,gs,Cur_win>; сохранение в стеке
         call  calloffs     ; вычисление константы offsline
         mov   gs, GenSeg   ; gs = сегмент таблицы цветов
drwout:  push  cx           ; сохраняем счетчик повторов
         mov   cx, dx       ; задаем размер строки рисунка
         call  drawline     ; построение очередной строки
         pop   cx           ; восстанавливаем счетчик повторов
         add   di, bx       ; коррекция адреса видеопамяти
         jnc   @F           ; -> адрес в пределах сегмента
         call  NxtWin       ; установка следующего окна
```

```
@@:     loop  drwout          ; управление повторами цикла
        PopReg <Cur_win,gs,cx,bx,si,di>; восстановление из стека
        call  setwin          ; восстановление исходного окна
        ret                   ; возврат из подпрограммы
```

Если вы сравните тексты примеров 3.21 и 7.23, то обнаружите, что они различаются только второй и третьей командами, а в списках параметров макровызовов `PushReg` и `PopReg` в примере 7.23 добавился регистр `gs`. Ну и, конечно же, имя `drawline` относится к двум разным подпрограммам.

Построение большого рисунка. Если образ рисунка не помещается в одном сегменте, то его приходится считывать из файла и выводить на экран по частям. Поэтому перед началом цикла построения вычисляется размер порции считываемых данных в байтах и количество содержащихся в ней строк. Способ построения рисунка по частям показан в примере 3.22. В него надо внести некоторые изменения, для учета особенностей режимов `direct color`.

Измененный текст приведен в примере 7.24. Напомним, что в этом случае при вызове явно указывается только адрес начала рисунка в видеопамяти (в регистре `di`). Размеры рисунка подпрограмма выбирает из переменных `iheight` и `iwidth`. Как обычно, должно быть установлено исходное окно видеопамяти, а в регистре `es` указан код видеосегмента (обычно `A000h`).

Пример 7.24. Построение рисунка из файла произвольного размера

```
BigDraw: pusha              ; сохраняем стандартные регистры
         PushReg <fs, gs, Cur_win>  ; сохраняем fs, gs и Cur_win
         mov   fs, SwpSeg   ; fs = сегмент буфера обмена
         mov   gs, GenSeg   ; gs = сегмент таблицы цветов
         mov   SwpOffs, 0   ; нулевой адрес в буфере обмена
         xor   dx, dx       ; старшая часть делимого dx=0
         mov   ax, -1       ; младшая часть делимого ax=65535
         div   iwidth       ; ax = 65535 / iwidth
         mov   part, ax     ; число строк в порции для чтения
         mul   iwidth       ; количество байтов в порции для чтения
         mov   numbyte, cx  ; сохраняем его в numbyte
         mov   ax, iheight  ; копируем количество строк в рисунке
         mov   remline, ax  ; в счетчик не выведенных строк
         mov   dx, iwidth   ; dx = iwidth, ширина рисунка
         call  calloffs     ; вычисляем константу переадресации
NewPart: mov   cx, numbyte  ; размер порции для чтения
         call  readf        ; чтение очередной порции данных
         jnc   sucread      ; -> чтение без ошибок
;        Здесь должны выполняться действия в случае ошибки чтения
```

```
sucread: mov  cx, part     ; cx = стандартное количество строк
         cmp  remline, cx  ; осталось меньше строк ?
         jae  lbl_1        ; -> нет, обходим команду пересылки
         mov  cx, remline  ; cx = оставшееся число строк
lbl_1:   sub  remline, cx  ; уменьшаем оставшееся число строк
         xor  si, si       ; адрес начала в буфере обмена
drwout:  push cx           ; сохраняем счетчик повторов
         mov  cx, dx       ; задаем размер строки рисунка
         call drawline     ; построение очередной строки
         pop  cx           ; восстанавливаем счетчик повторов
         add  di, bx       ; адрес начала следующей строки
         jnc  @F           ; -> адрес в пределах сегмента
         call NxtWin       ; установка следующего окна
@@:      loop drwout       ; управление повторами цикла
         cmp  remline, 0   ; все строки выведены ?
         jne  NewPart      ; -> нет, продолжаем построение
         PopReg <Cur_win, gs, fs>   ; восстановление Cur_win, gs и fs
         popa              ; восстановление всех регистров
         call setwin       ; восстановление исходного окна
         ret               ; возврат из подпрограммы
```

По сравнению с оригиналом (см. пример 3.22), текст примера 7.24 изменился в той части, где выполняются подготовительные действия. Добавилось сохранение в стеке содержимого регистра `gs` и запись в него кода сегмента, содержащего таблицу цветов, а для вычисления константы переадресации строк видеопамяти вызывается подпрограмма `calloffs`. В заключительной части примера добавилось восстановление из стека исходного содержимого `gs`.

Цикл построения рисунка не изменился, но имя `drawline` соответствует другой подпрограмме построения строки, поскольку нужна перекодировка точек по таблице цветов.

Учет лишних байтов. В примерах 7.23 и 7.24, так же, как и в примерах 3.21 и 3.22, предполагается, что строки образа рисунка расположены в файле подряд друг за другом. На практике это не всегда так. По разным причинам к каждой строке может быть добавлено некоторое количество байтов, содержимое которых не определено и их нежелательно записывать в видеопамять.

В указанных примерах вывод "лишних" байтов исключен, поскольку обрабатывается ровно столько точек, сколько их содержится в строке. Однако в них отсутствует пропуск "лишних" байтов в строке образа рисунка. Если таковые имеются, то построенное изображение будет искажено.

Способ определения наличия дополнительных байтов зависит от особенностей формата (или стандарта), которому соответствует файл, содержащий

образ рисунка. Общих правил не существует. Как это делается при работе с файлами формата BMP, подробно описано в приложении А.

7.5. Рисунки, не использующие палитру

Полноцветные (full-color) или художественные точечные компьютерные рисунки не используют палитру. Цвет каждой точки указывается в ее коде, который занимает три байта, содержащих восьмиразрядные коды базовых цветов. Обычно такие рисунки получаются путем оцифровки, например, сканирования, цветных фотографий, слайдов, картинок, видеокадров и т. п. Иногда при подготовке значков или пиктограмм различного назначения они создаются вручную с помощью графических редакторов.

Чаще всего образы рисунков хранятся в сжатом виде, поэтому перед записью кодов точек в видеопамять производится их распаковка, но и в тех случаях, когда образ рисунка не упакован, приходится преобразовывать коды точек в формат, соответствующий установленному видеорежиму.

В данном разделе описано построение рисунков, хранящихся в файлах, соответствующих стандартам BMP и PCX. Кроме того, автор счел целесообразным включить в него краткий обзор способов сжатия полноцветных рисунков, поскольку от этого зависят не только выполняемые в задачах действия, но и качество получаемого изображения.

7.5.1. Рисунки, подготовленные в стандарте BMP

Принятый в стандарте BMP способ сжатия не эффективен, поэтому образы полноцветных рисунков обычно не упакованы. Это упрощает цикл построения рисунка, но не исключает необходимости преобразования кодов точек в формат, соответствующий установленному задачей видеорежиму.

При построении полноцветных рисунков существенно изменяются подготовительные действия, поэтому мы начнем с их подробного обсуждения. Полное описание стандарта BMP приведено в приложении А данной книги. Вам лучше предварительно прочитать его, для того чтобы узнать, как производится чтение заголовка файла и анализ его основных полей. Без этого вы не сможете использовать описанную ниже подпрограмму.

Строки образа рисунка хранятся в файле в обратном порядке, т. е. первой записана последняя строка, а последней — первая. Код каждой точки занимает три байта, содержащих базовые цвета в формате bgr. Адрес начала строки в файле должен быть кратен 4, а т. к. размер строк (в байтах) кратен 3, то после обработки каждой из них может понадобиться пропустить от 0 до 3 байтов в файле. Специального признака, указывающего наличие до-

полнительных байтов в строке, не существует, поэтому надо вычислить размер строки в файле и количество "лишних" байтов. Размер строки в файле равен утроенному количеству точек в строке, округленному до значения кратного четырем. А разность между округленным и не округленным значениями равна количеству дополнительных байтов.

Способ построения рисунка. При построении рисунка небольшого размера (см. пример 7.24) мы выбирали из оперативной памяти и записывали в видеопамять строки образа рисунка в порядке их естественного расположения (в строящемся рисунке). Однако при построении таким способом большого рисунка, как уже говорилось, потребуется дополнительное позиционирование файла, что нежелательно. Проще позиционировать строки в видеопамяти. Поэтому мы будем выбирать строки образа рисунка в том порядке, в котором они хранятся в файле, а выводить на экран снизу вверх. Первой в файле записана последняя строка, ее и надо строить первой, затем изменится адрес видеопамяти и строится предпоследняя строка и так вплоть до первой. Но при такой последовательности действий надо изменить способ вычисления адресов строк рисунка в видеопамяти.

Коррекция адресов строк. Для определения адреса начала следующей строки мы прибавляли к текущему адресу видеопамяти константу коррекции, которая вычисляет подпрограмма `calloffs` (см. пример 7.13) по формуле `(horsize - widthrect)*bytppnt`. В данном случае нас интересует адрес предыдущей строки, поэтому формулу для вычисления константы коррекции надо записать в виде `(horsize + widthrect)*bytppnt` и *вычитать* вычисленное значение из текущего адреса видеопамяти.

Объясним, почему так, а не иначе. В процессе построения строки исходный адрес видеопамяти увеличивается на `widthrect*bytppnt` байтов. Если текущий адрес видеопамяти уменьшить на эту величину, то получится адрес начала построенной строки. А если адрес начала текущей строки уменьшить на величину `bperline` (`horsize*bytppnt`), то получится адрес начала предыдущей строки (см. табл. 7.4).

Есть два способа вычисления нужной нам величины. Можно составить вариант подпрограммы `calloffs`, в котором вычитание заменено сложением. А можно ничего не изменять в `calloffs`, но перед ее вызовом указывать в регистре `dx` отрицательное значение переменной `widthrect` (ширина рисунка). Мы используем второй способ.

Адрес начала последней строки. Для построения рисунка в нужном месте экрана обычно задается адрес видеопамяти, соответствующий левой верхней точке (началу) рисунка. В данном случае построение рисунка начинается с его левой нижней точки (начало последней строки). Поэтому перед построением рисунка в подпрограмме надо вычислять адрес этой точки.

В видеопамяти начало последней строки рисунка отстоит от начала первой на `(N - 1)*bperline` байтов, где `N` — количество строк в рисунке. Следова-

тельно, надо вычислить указанную величину, выразить ее старшую часть в единицах приращения окна видеопамяти и сложить полученный результат с адресом первой точки рисунка.

Размер порции данных. Рисунки большого размера считываются из файла и строятся по частям. Размер считываемой части файла (порции данных) надо выбрать таким, чтобы в нем укладывалось целое число строк. Это исключает лишние проверки в процессе построения рисунка.

При чтении из файла размер порции данных задается в байтах, а для управления повторами цикла построения рисунка используется количество считанных строк. В примере 3.22 количество строк вычислялось путем деления числа 65 535 на размер строки, а размер порции — путем умножения количества строк на 65 535. В данном случае размер строки в файле может не совпадать с размером строки в рисунке и перед делением его надо вычислить.

Новые переменные. Для хранения размеров рисунка и величин, вычисляемых в процессе подготовительных действий, в примере 3.22 использовались 5 переменных, описанных в разделе данных задачи. В этом случае к ним добавляется еще три, поэтому в разделе данных задачи должны быть описаны следующие переменные:

```
iwidth    dw  0   ; количество точек в строке (ширина) рисунка
fwidth    dw  0   ; количество байтов в строке файла
iheight   dw  0   ; количество строк в рисунке (высота рисунка)
rmndr     dw  0   ; добавка к адресу для пропуска "лишних" байтов
part      dw  0   ; количество строк в считываемой порции данных
numbyte   dw  0   ; количество байтов в считываемой порции данных
remline   dw  0   ; количество строк, которые еще не обработаны
```

Значения переменных `iwidth` и `iheight` выбираются из полей заголовка файла, а значения остальных переменных формируются в подпрограмме построения рисунка при выполнении подготовительных действий.

Подпрограмма BigBmp. Текст подпрограммы приведен в примере 7.25. Перед ее вызовом должно быть установлено окно видеопамяти, в котором расположено начало рисунка. Его адрес указывается в регистре `di`, а `es` должен содержать код видеосегмента. Предполагается, что задача предварительно открыла файл для чтения и обработала его заголовок так, как это рекомендовано в приложении А, поэтому известны значения переменных `iwidth` и `iheight` и файл установлен на начало образа рисунка.

Напомним

Перед построением любого рисунка, тем более большого размера, надо удалить с экрана изображение курсора (`call Hidepnt`), а после построения рисунка восстановить его на экране (`call Showpnt`).

Пример 7.25. Построение полноцветного рисунка формата BMP

```
BigBmp:  pusha                  ; сохранение "всех" регистров
         PushReg <fs,Cur_win>   ; сохранение fs и Cur_win
         mov   fs, SwpSeg       ; fs = сегмент буфера обмена
         mov   SwpOffs, 0       ; очистка смещения в сегменте
         mov   dx, iwidth       ; dx = количество точек в строке
         neg   dx               ; dx = – iwidth
         call  calloffs         ; bx = (horsize + iwidth)*bytppnt
         mov   ax, 03           ; ax = 3
         mul   iwidth           ; ax = 3*iwidth, утроенный размер строки
         mov   dx, ax           ; сохраняем его в dx
         add   ax, 03           ; с помощью двух команд округляем
         and   al, 0FCh         ; размер строки до значения, кратного 4
         mov   fwidth, ax       ; fwidth = размер строки в файле
         sub   ax, dx           ; вычисляем добавку к адресу
         mov   rmndr, ax        ; и сохраняем ее в rmndr
         mov   ax, -1           ; ax = 65535
         mov   numbyte, ax      ; numbyte = 65535
         xor   dx, dx           ; очистка старшей части делимого
         div   fwidth           ; ax = 65535 / fwidth (частное от деления)
         mov   part, ax         ; part = число строк в порции для чтения
         sub   numbyte, dx      ; numbyte = numbyte – ax
         mov   ax, iheight      ; ax = количество строк в рисунке
         mov   remline, ax      ; remline – счетчик числа строк
         dec   ax               ; ax = номер последней строки
         mul   bperline         ; ax = (iheight – 1)*bperline
         add   di, ax           ; di = адрес последней строки рисунка
         adc   dx, 00           ; учитываем возможный перенос
         mov   ax, GrUnit       ; ax = GrUnit (единица измерения окон)
         mul   dl               ; вычисляем добавку к номеру окна
         add   Cur_win, ax      ; номер окна для последней строки
         call  Setwin           ; установка окна
NewPart: mov   cx, numbyte      ; cx = количество считываемых байтов
         call  Readf            ; чтение порции в буфер обмена
         jnc   sucread          ; -> чтение прошло без ошибок
;        Здесь должны выполняться действия в случае ошибки при чтении
sucread: mov   cx, part         ; cx = кол-во строк в полной порции
         cmp   remline, cx      ; считана полная порция данных ?
         jae   @F               ; -> да, обходим следующую команду
         mov   cx, remline      ; нет, cx = оставшееся число строк
@@:      sub   remline, cx      ; уменьшаем значение счетчика строк
         xor   si, si           ; si = начало буфера обмена
drwout:  push  cx               ; сохраняем значение счетчика строк
         mov   cx, iwidth       ; cx = размер строки (в точках)
         call  drawline         ; построение очередной строки
```

```
        pop   cx              ; восстанавливаем счетчик строк
        add   si, rmndr       ; корректируем адрес в буфере обмена
        sub   di, bx          ; di = адрес начала предыдущей строки
        jnc   @F              ; -> адрес в пределах текущего окна
        call  PrevWin         ; установка предыдущего окна
@@:     loop  drwout          ; управление циклом построения строк
        cmp   remline, 0      ; остались не обработанные строки ?
        jne   NewPart         ; -> да, на чтение следующей порции
        PopReg <Cur_win,fs>   ; восстановление Cur_win и fs
        popa                  ; восстановление "всех" регистров
        call  Setwin          ; восстановление исходного окна
        ret                   ; возврат из подпрограммы
```

Выполнение примера 7.25 начинается с сохранения в стеке содержимого "всех" регистров. Команда `pusha` не сохраняет сегментные регистры, поэтому содержимое `fs` и переменной `Cur_win` сохраняет макровызов `PushReg`.

Для чтения данных вызывается подпрограмма `Readf`, текст которой приведен в примере 3.23. Она размещает данные в буфере обмена, начиная с адреса, указанного в `SwpOffs`. Для максимального использования пространства буфера эта переменная предварительно очищается, а в регистр `fs` записывается сегмент буфера обмена из `SwpSeg`.

После этого выполняются подготовительные действия, смысл и назначение которых описаны выше. Три первые команды вычисляют константу для коррекции адресов строк, ее значение помещается в регистр `bx` и используется в основном цикле. Следующие восемь команд формируют значения переменных `fwidth` и `rmndr`. Затем шесть команд вычисляют значения переменных `part` и `numbyte`. Последние десять команд формируют адрес начала последней строки рисунка в видеопамяти и устанавливают окно, которому он принадлежит.

Основной цикл имеет метку `NewPart`. Он практически совпадает с одноименным циклом примера 7.24. Отличие заключается в следующем. При переадресации строк видеопамяти вычисляется адрес начала предыдущей строки, поэтому содержимое регистра `bx` вычитается из содержимого регистра `di`, а в случае переполнения результата вычитания устанавливается предыдущее окно видеопамяти. Кроме того, для исключения "лишних" точек адрес буфера обмена, хранящийся в регистре `si`, увеличивается на величину `rmndr`. Ну и, конечно же, имя `drawline` соответствует разным подпрограммам.

Подпрограммы для построения строк. В отличие от основного текста примера 7.25, тексты подпрограмм построения строк существенно зависят от установленного задачей видеорежима. Это связано с необходимостью преобразования исходного формата `bgr` в формат `Hi-Color` или `True Color`. Варианты подпрограмм для обоих режимов показаны в примере 7.26.

Пример 7.26. Варианты подпрограммы построения строки

```
;          Вариант 1 для работы в режимах True Color
drawline: movs  word ptr [di], fs:[si]; копируем коды синего и зеленого
          lods  byte ptr fs:[si]    ; al = код красного цвета
          xor   ah, ah              ; очистка резервного байта
          stosw                     ; записываем старшее слово кода
          or    di, di              ; адрес в пределах сегмента ?
          jnz   @F                  ; -> да, обход команды
          call  NxtWin              ; установка следующего окна
@@:       loop  drawline            ; управление повторами цикла
          ret                       ; возврат из подпрограммы
;        Вариант 2 для работы в 15-разрядных режимах Hi-Color
drawline: mov   al, fs:[si+2]  ; читаем код красного цвета в al
          shr   al, 03         ; сокращаем его до 5-ти разрядов
          mov   bh, fs:[si+1]  ; читаем код зеленого цвета в bh
          shld  ax, bx, 05     ; добавляем в ax код зеленого цвета
          mov   bh, fs:[si]    ; читаем код синего цвета в bh
          shld  ax, bx, 05     ; сдвигаем и дополняем код в ax
          add   si, 03         ; добавляем в ax код синего цвета
          stosw                ; записываем код в видеопамять
          or    di, di         ; адрес в пределах сегмента ?
          jnz   @F             ; -> да, обход команды
          call  NxtWin         ; установка следующего окна
@@:       loop  drawline       ; управление повторами цикла
          ret                  ; возврат из подпрограммы
```

В разделах 7.4.1 и 7.4.2 описано преобразование цветов, хранящихся в палитрах, в форматы `Hi-Color` и `True Color`. Отличие рассмотренного здесь случая в том, что исходные цвета расположены не в палитре, а в строке образа рисунка, находящейся в буфере обмена. Поэтому первый вариант примера 7.26 является циклом примера 7.19, а второй вариант примера 7.26 является циклом примера 7.18.

Таким образом, при построении полноцветных рисунков формата `BMP` текст основной подпрограммы не зависит, а вспомогательной (`drawline`) зависит от видеорежима. Исключить эту зависимость невозможно, но при использовании директив условного ассемблирования можно задавать признак для выбора нужной подпрограммы.

7.5.2. Рисунки, подготовленные в стандарте PCX

В файлах формата `PCX` образы полноцветных рисунков обычно хранятся в упакованном виде. Эффективность принятого в стандарте `PCX` способа упаковки не высока, но иногда получаются удовлетворительные результаты.

В данном разделе описана подпрограмма, выполняющая построение упакованного полноцветного рисунка произвольного размера. Как вы сможете убедиться, она во многом совпадает с подпрограммой аналогичного назначения, описанной в примере 3.26. Это объясняется тем, что вспомогательные действия выполняются в подпрограммах распаковки и построения строк. Поэтому мы начнем с рассмотрения способа распаковки.

Новое поле заголовка. Заголовок файла стандарта PCX имеет фиксированный размер 80h байтов. Назначение его основных полей (байтов и слов) описано в разделе 3.3.3. Здесь нас интересует еще одно поле, которое там не упоминалось.

В байте со смещением 41h указано количество цветов (битовых плоскостей), на которое разложена каждая строка образа рисунка. Если оно равно 3, то мы имеем дело с полноцветным рисунком. Никакого другого признака, указывающего на то, что рисунок полноцветный, не существует.

Напоминаем

В байте со смещением 2 хранится "ключ кодирования", значение которого может быть равно 0 или 1. В первом случае образ рисунка не упакован, а во втором упакован по способу RLE, который описан в разделе 3.3.3.

Способ распаковки строки. Перед упаковкой строка рисунка разлагается на три подстроки, размер каждой из которых равен размеру строки. Одна из них содержит коды красного базового цвета, другая — зеленого и третья — синего. Другими словами, производится разложение строки по компонентам базовых цветов. Затем каждая подстрока упаковывается по способу RLE, и результат записывается в файл. Если исходный рисунок состоял из K строк, содержащих трехбайтовые коды точек, то упакованный рисунок содержит 3K групп однобайтовых кодов цвета. Зачем это делается, описано в следующем разделе, а здесь нас интересует способ распаковки.

Итак, в файле формата PCX, содержащем упакованный полноцветный рисунок, начиная с адреса 80h, записаны группы упакованных однобайтовых кодов базовых цветов в такой последовательности:

коды красного базового цвета первой строки рисунка
коды зеленого базового цвета первой строки рисунка
коды синего базового цвета первой сроки рисунка
коды красного базового цвета второй строки рисунка
коды зеленого базового цвета второй строки рисунка
и т. д. вплоть до
коды синего базового цвета последней строки рисунка

В упакованном виде размеры групп переменные, но после распаковки каждая из них состоит из N байтов, где N — количество точек в строке рисунка. При распаковке первых трех групп будут получены базовые цвета всех точек

первой строки рисунка. При распаковке следующих трех групп — второй строки рисунка и т. д.

Таким образом, для получения цветов всех точек одной строки рисунка цикл распаковки повторяется три раза. Однако если результаты записывать в подряд расположенные байты памяти, то базовые цвета одной точки будут отстоять друг от друга на `N` байтов и это придется учитывать при пересылке распакованного образа рисунка в видеопамять.

Для того чтобы не усложнять подпрограммы построения строки, надо изменить порядок записи результатов распаковки так, чтобы базовые цвета одной точки оказались в трех подряд расположенных байтах оперативной памяти. Другими словами, подпрограмма распаковки должна восстанавливать формат `rgb` или `bgr`, по усмотрению программиста. Мы выберем формат `bgr` для того, чтобы при построении строки можно было использовать подпрограммы примера 7.26.

Подпрограмма Unpack. Текст подпрограммы распаковки строки рисунка с преобразованием в формат `bgr` показан в примере 7.27. Распакованная строка записывается в свободную часть буфера общего назначения, поэтому перед вызовом подпрограммы в регистр `gs` копируется содержимое переменной `GenSeg`. Адрес в буфере общего назначения подпрограмма выбирает из переменной `GenOffs`. Для чтения байтов упакованной строки вызывается вспомогательная подпрограмма `Nxt_sym` (см. пример 3.25, раздел 3.3.3), которая помещает в регистр `al` код очередного байта из буфера обмена и следит за тем, чтобы этот буфер не был пустым.

Пример 7.27. Распаковка строки и преобразование в формат bgr

```
Unpack:  PushReg <ax,dx,cx,di>; сохранение используемых регистров
         mov   di, GenOffs     ; адрес начала строки в GenSeg
         add   di, 02          ; начинаем с записи красных цветов
         mov   cx, 03          ; количество цветов
Unpck1:  PushReg <di,cx>       ; сохранение содержимого di и cx
         mov   dx, fwidth      ; логический размер строки
Unpck2:  call  nxt_sym         ; возвращает в al — текущий код
         mov   cx, 01          ; предполагаем одиночный символ
         cmp   al, 0C0h        ; одиночный символ ?
         jbe   Unpck3          ; => да, на запись символа
         mov   cx, ax          ; пересылка ax в счетчик
         and   cx, 3Fh         ; выделяем число повторов
         call  Nxt_sym         ; читаем повторяемый код
Unpck3:  sub   dx, cx          ; кол-во байтов до конца строки
Unpck4:  mov   gs:[di], al     ; запись кода цвета
         add   di, 03          ; коррекция адреса
         loop  Unpck4          ; управление повторами записи
```

```
        or    dx, dx          ; обработана вся строка ?
        jnz   Unpck2          ; => нет, продолжение обработки
        PopReg <cx,di>        ; восстановление содержимого di и cx
        dec   di              ; для записи зеленых или синих цветов
        loop  Unpck1          ; управление внешним циклом
        PopReg <di,cx,dx,ax>  ; восстановление регистров
        ret                   ; возврат из подпрограммы
```

Подпрограмма примера 7.27 состоит из двух вложенных циклов, внешний имеет метку `Unpck1`, а внутренний `Unpck2`.

Внутренний цикл отличается от аналогичного цикла `Unploop` примера 3.24 тем, что запись результатов распаковки выполняется в цикле, имеющем метку `Unpck4` и состоящем из трех команд. Он усложнен потому, что результаты распаковки записываются в память не подряд друг за другом, а с шагом в 3 байта.

Внешний цикл управляет трехкратным повторением внутреннего. Кроме того, он формирует в регистре `di` адрес для записи результатов так, чтобы строка соответствовала формату `bgr`.

Подпрограмма PackDrw. Перед построением рисунка надо прочитать заголовок файла, проверить его соответствие стандарту `PCX` и наличие в нем полноцветного упакованного рисунка. Затем из полей заголовка выбираются значения переменных `iheight`, `iwidth` и `fwidth`. Если заголовок прочитан полностью (`80h` байтов), то файл установлен на начало образа рисунка.

Текст подпрограммы приведен в примере 7.28. Перед ее вызовом устанавливается окно видеопамяти, в котором расположено начало строящегося рисунка. Адрес начала рисунка указывается в регистре `di`, а регистр `es` должен содержать код видеосегмента (`A000h`). Кроме того, надо удалить с экрана изображение курсора (`call Hidepnt`) и восстановить его на экране (`call Showpnt`) после построения рисунка.

Пример 7.28. Построение упакованного рисунка формата PCX

```
PackDrw: pusha              ; сохранение "всех" регистров
         PushReg <fs,gs,Cur_win>    ; сохранение fs, gs и Cur_win
         mov   gs, GenSeg   ; gs = сегмент общего назначения
         mov   fs, SwpSeg   ; fs = сегмент буфера обмена
         xor   si, si       ; адрес начала буфера обмена
         mov   SwpOffs, si  ; адрес начала буфера обмена
         mov   incount, si  ; incount = 0 — буфер обмена пуст
         mov   dx, iwidth   ; dx = количество точек в строке рисунка
         mov   cx, iheight  ; cx = количество строк в рисунке
         call  calloffs     ; bx = константа для коррекции адресов
```

```
make:   push  cx              ; сохраняем счетчик повторов цикла
        call  Unpack          ; распаковка очередной строки
        PushReg <fs,si>       ; сохраняем содержимое fs и si
        lfs   si, dword ptr GenOffs; fs:si = адрес распакованной строки
        mov   cx, dx          ; cx = количество точек в строке рисунка
        call  drawline        ; построение очередной строки
        add   di, bx          ; адрес начала следующей строки
        jnc   @F              ; -> адрес в пределах текущего сегмента
        call  Nxtwin          ; установка следующего окна
@@:     PopReg <si,fs,cx>     ; восстанавливаем содержимое si, fs и cx
        loop  make            ; управление повторами цикла
        PopReg <Cur_win,gs,fs>      ; восстановление Cur_win, gs и fs
        popa                  ; восстановление "всех" регистров
        call  Setwin          ; восстановление исходного окна
        ret                   ; возврат из подпрограммы
```

Выполнение примера 7.28 начинается с сохранения в стеке содержимого всех регистров, а также сегментных регистров `fs`, `gs` и переменной `Cur_win`. Затем в `gs` и `fs` записываются коды сегментов буферов `GenSeg` и `SwpSeg` и очищаются регистр `si`, переменные `SwpOffs` и `incount`. В регистры `dx` и `cx` копируются размеры рисунка, и вызывается подпрограмма `calloffs` для вычисления константы переадресации строк.

Основной цикл примера 7.28 имеет метку `make`. Для построения каждой строки в нем последовательно вызываются подпрограммы `Unpack` и `drawline`. Перед вызовом `drawline` сохраняется исходное содержимое пары регистров `fs:si`, а в них помещается адрес распакованной строки. После возвращения из `drawline`, как обычно, корректируется адрес видеопамяти, восстанавливается содержимое регистров `si`, `fs`, `cx` и команда `loop make` управляет повторами цикла.

Перед возвратом на вызывающий модуль восстанавливаются все сохраненные в стеке величины и исходное окно видеопамяти. Завершает подпрограмму команда `ret`.

В зависимости от видеорежима в примере 7.28 используется один из вариантов подпрограмм `drawline`, описанных в примере 7.26.

Рисунки, использующие палитру. Подпрограмма `PackDrw` позволяет строить рисунки, подготовленные с применением палитры цветов. Для распаковки таких рисунков в ней вызывается подпрограмма `Unpack`, описанная в примере 3.26, а вариант подпрограммы `drawline` выбирается в зависимости от установленного видеорежима.

При работе в режимах `PPG` для построения распакованной строки вызывается одна из подпрограмм `drawline`, описанных в разделе 3.3.1.

Если же задача работает в одном из режимов `direct color`, то перед построением рисунка хранящаяся в файле палитра преобразуется в таблицу

цветов. В зависимости от видеорежима для этого выбирается одна из подпрограмм 7.17—7.20. Строку рисунка с перекодированием точек по таблице цветов строит подпрограмма примера 7.21.

7.5.3. Способы сжатия полноцветных рисунков

Возможность сжатия образов точечных рисунков была предусмотрена уже в первых стандартах хранения графических данных. Основным критерием при выборе алгоритмов упаковки была простота их программной реализации, а не степень сжатия исходных данных. В то время никто не предполагал, что размеры рисунков смогут достигать нескольких миллионов байтов. И уж тем более трудно было представить, что проблема сжатия графических данных станет настолько важной, что ради ее решения придется жертвовать качеством исходного изображения. Все это стало очевидным намного позже, когда элементная база позволила работать с рисунками большого размера и получили широкое распространение сначала локальные, а затем и глобальные вычислительные сети.

Сжатие по способу RLE. В стандартах `PCX` и `BMP` предусмотрены простые алгоритмы упаковки и распаковки данных, которые называются `Run Length Encoding` или `RLE`. Реализация способа `RLE` в этих стандартах различается в деталях, но суть остается неизменной. Она заключается в том, что группа повторяющихся кодов заменяется двумя байтами, первый из которых содержит количество повторов, а второй — повторяемый код. Чем больше размер группы одинаковых кодов, тем выше степень сжатия.

Способ `RLE` основан на анализе содержимого соседних байтов. У рисунков, подготовленных с применением палитры, в них находятся коды смежных точек. А у полноцветных рисунков в каждой тройке подряд расположенных байтов хранятся базовые цвета точек, и проверять их совпадение не имеет смысла. Коды базовых цветов совпадают только у точек, окрашенных в оттенки серого цвета. Из 16M возможных комбинаций на оттенки серого цвета приходится всего 256!

Для того чтобы сжатие стало возможным, надо изменить порядок анализа байтов строки рисунка. Это можно сделать двумя способами: либо предварительно разложить строку на три группы кодов одноименных базовых цветов и сжимать каждую группу независимо друг от друга, либо трижды повторить процесс упаковки строки, обрабатывая каждый раз коды одноименных базовых цветов, отстоящие друг от друга на три байта. Так или иначе, но при упаковке полноцветных рисунков необходима раздельная обработка трех групп одноименных базовых цветов каждой строки.

Эффективность способа RLE зависит от структуры сжимаемого рисунка. Она тем выше, чем чаще в нем встречаются подряд расположенные одноцветные точки. Оценить ее в общем случае невозможно, поэтому мы огра-

ничимся примерами трех рисунков, характеристики которых приведены в табл. 7.5. Первый рисунок является значком, предназначенным для оформления рабочей области экрана, в нем использованы только оттенки серого цвета. Второй получен в результате сканирования цветной фотографии женского лица, в этом рисунке количество цветов не соответствует качеству изображения, на котором много пятен инородного происхождения. Третий рисунок взят из коллекции Corel PrintHouse, он подготовлен на высоком профессиональном уровне. Это снимок берега моря, уходящего за горизонт, на переднем плане волны прибоя и несколько лодок. У этого рисунка количество цветов соответствует качеству изображения.

Таблица 7.5. *Результат сжатия трех полноцветных рисунков*

Содержание рисунка	Размер рисунка	Количество цветов	Размер файла в байтах		
			BMP	PCX	JPEG
Значок	99×40	251	12 054	12 656	1565
Фотография	184×246	22 158	135 846	90 302	7460
Ландшафт	736×497	41 682	1 097 430	1 079 973	138 964

Табл. 7.5 иллюстрирует недостатки способа сжатия RLE и очевидное преимущество стандарта JPEG. Приведенные в ней данные получены в результате преобразования файлов из формата JPG в форматы BMP (не упакованный файл) и PCX (упаковка по способу RLE).

Из табл. 7.5 видно, что только в одном случае файл формата PCX оказался короче файла формата BMP примерно на 35%. Это произошло потому, что на фотографии женское лицо окружает довольно большое пространство одноцветного фона. В двух остальных случаях сжатие по способу RLE не дает ощутимых результатов. Причина кроется не только в недостатках способа сжатия, но и в самих рисунках.

Применение палитры ограничивало разнообразие кодов точек до 256, поэтому в рисунках неизбежно повторялись одноцветные точки и их комбинации. В этих условиях алгоритм LZW, реализованный в стандарте GIF (см. раздел 3.3.3), обеспечивал вполне удовлетворительные результаты.

У полноцветных рисунков разнообразие цветов (кодов) точек ограничивает только структура изображения. При этом чем больше разнообразие кодов точек, тем меньше вероятность обнаружить в рисунке их одинаковые комбинации. И никакой способ сжатия не будет эффективен до тех пор, пока не будет ограничено количество использованных в рисунке цветов.

Находящееся на экране, или напечатанное на бумаге изображение предназначено для восприятия человеком. Наш орган зрения является не столь

совершенным инструментом, как это кажется. Поэтому правомерна такая постановка вопроса: можно ли изменить цвета в рисунке так, чтобы человеческий глаз этого не заметил, и одновременно появилась возможность сжатия? При определенных условиях ответ на этот вопрос положительный.

Результаты психофизиологических исследований свидетельствуют о том, что мы с вами лучше воспринимаем небольшие изменения яркости, чем небольшие изменения цвета. Незначительное изменение цветов точек при сохранении их яркости может оказаться незаметным для глаза. Но для этого надо иметь возможность независимо изменять яркость и цвет, что не возможно сделать в пространстве цветов RGB.

Цветовое пространство YUV. Трехмерное пространство света, у которого яркость и цвет являются независимыми компонентами, обычно обозначается YUV или YCbCr, в англоязычной технической литературе его называют "luminance/chrominance color space". Точная расшифровка аббревиатуры YUV автору неизвестна. Входящие в нее буквы обозначают следующие величины: Y — светимость (luminance), U — синий цвет, V — красный цвет. Во втором варианте записи Cb является сокращением слов chrominance-blue, а Cr — chrominance-red. Обратите внимание, в данном случае при описании компонент английские слова "яркость" и "цвет" не употребляются, поскольку мы имеем дело с некими абстрактными или обобщенными величинами.

Уравнения, связывающие пространство RGB с пространством YUV, приведены в табл. 7.6. Значения коэффициентов уравнений определяются на основании анализа кривой спектральной чувствительности приемника света, которым в данном случае является человеческий глаз.

Таблица 7.6. *Уравнения для преобразования пространства цветов*

RGB -> YUV	YUV -> RGB
Y = 0,299R + 0,587G + 0,114B	R = Y + 1,134V
U = 0,434B — 0,146R — 0,288G	G = Y — 0,578V — 0,396U
V = 0,617R — 0,517G — 0,100B	B = Y + 2,045U

Простой визуальный анализ показывает, что наибольший вклад в значение компоненты Y вносит зеленый цвет, к которому наиболее чувствителен глаз человека. В значение компоненты U основной вклад вносит синий цвет, а компоненты V — красный. Точкам серого цвета, у которых R=G=B=k соответствуют Y=k, U=V=0. Таким образом, компонента Y соответствует яркости точек серого цвета.

Значения U и V могут быть отрицательными величинами, поэтому в некоторых случаях к ним добавляют такую постоянную величину, при которой ре-

зультаты будут только положительными числами, и учитывают эту величину при обратном преобразовании.

Замечание

В группу прерывания int 10h входит функция, имеющая код 101Bh. Она преобразует цвета R,G,B в оттенки серого по формуле Y = 0,299R + 0,587G + 0,114B.

В описании стандарта VESA перечислено 8 кодов моделей видеопамяти. Как уже говорилось в главе 1 (см. раздел 1.2.2), на практике используются только 4 из них. К числу названных, но не используемых относится и модель YUV. Причем отсутствуют даже намеки на ее возможные характеристики. Это объясняется тем, что в простых видеокартах она не применяется. Зато современные акселераторы, как правило, выполняют преобразования, описанные в табл. 7.6. Они применяются, например, в тех случаях, когда при построении трехмерных графических объектов светимость точек зависит от расстояния до источника света, а цвет при этом не изменяется.

При использовании способа JPEG преобразование RGB -> YUV предшествует сжатию графического изображения, а обратное преобразование выполняется после его распаковки. Покажем, что это дает на двух примерах.

Предположим, что рисунок содержит N точек и занимает в памяти 3N байтов. Если в нем использованы только различные оттенки серого цвета, в пространстве YUV останется только одна компонента Y, а U и V будут содержать нули. То есть еще до сжатия занимаемое изображением пространство сокращается в три раза.

Другой пример, можно усреднить цвета четырех смежных точек, расположенных в квадрате размером 2×2 точки, оставив без изменения их светимость. При этом размеры компонент U и V сократятся в 4 раза, а пространство, занимаемое всем рисунком, сократится в 2 раза. В зависимости от степени различия усредняемых цветов, воспроизведенный рисунок будет несколько отличаться от оригинала. Если оригинал не содержал контрастных точек, то это различие окажется незаметным для человеческого глаза.

Таким образом, пространство цвета YUV позволяет решать те задачи, решение которых при работе с пространством RGB было невозможно.

Общая характеристика способа JPEG. JPEG — это аббревиатура от Joint Photographic Experts Group (объединенная группа экспертов по фотографии) и одновременно обозначение способа сжатия полноцветных рисунков. Для сжатия динамических изображений эта же группа разработала другой не менее распространенный стандарт, который называется MPEG.

Основная особенность JPEG заключается в том, что при сжатии происходит потеря качества исходного изображения. Наибольшим искажениям подвержены его контрастные участки, поэтому JPEG не предназначен для сжатия изображений, содержащих много контрастных линий, их края окажутся

размытыми или зазубренными. К таким изображениям относятся, например, чертежи. Если же резкие контрасты отсутствуют, то вносимые искажения будут мало заметны. Поэтому наиболее подходящими объектами являются рисунки с плавными переходами цветов или света и тени, например результаты ландшафтной фотосъемки.

Количество цветов, использованных в рисунке, сокращается при сжатии и восстанавливается при распаковке. Однако если сравнить результат распаковки и оригинал, то окажется, что коды большинства точек различаются. С формальной точки зрения после распаковки получается совершенно другое изображение. Поэтому имеет смысл говорить только о визуальном совпадении оригинала с результатом распаковки.

Об эффективности способа `JPEG` можно судить по результатам, приведенным в табл. 7.5. Качество воспроизведенного рисунка можно оценить только визуально, точных количественных критериев его оценки не существует. В технической документации принято говорить о сжатии с минимальными (`minimum`), малыми (`low`), средними (`medium`) и большими (`high`) потерями. Обычно рекомендуется сравнить разные степени сжатия и определить оптимальное соотношение между размером сжатого файла и качеством изображения на экране. В табл. 7.7 показано, как изменялся размер исходного файла в зависимости от величины потерь. Преобразования исходного файла формата `BMP` выполнялись с помощью графического редактора PhotoFinish фирмы ZSoft.

Таблица 7.7. *Качество рисунка и размер файла*

Величина потерь качества изображения	Размер файла в байтах
Исходный рисунок	2 476 101
Минимальные потери	244 998
Малые потери	73 654
Средние потери	56 196
Большие потери	43 506

Анализ табл. 7.7 показывает, что при сжатии с минимальными потерями размер исходного файла сократился примерно в 10 раз. Увеличение потерь до малых привело к дополнительному сокращению файла еще в три раза. Дальнейшее увеличение потерь не приводит к существенному сокращению файла, поэтому обычно рекомендуют выбирать минимальные или малые потери качества изображения.

Способы сжатия в JPEG. Стандарт рекомендует два способа сжатия исходного рисунка — основной и дополнительный. Большинство графических редакторов позволяют выбирать параметры обоих способов.

Дополнительный способ заключается в простом усреднении значений компонент U и V, которое было описано выше. При этом можно выбрать усреднение цвета одной, 2- или 4-х смежных точек, значения компоненты Y остаются без изменений. Выбор одной точки означает отказ от усреднения.

Для основного способа можно выбирать только степень потерь, о чем говорилось выше. Он заключается в том, что последовательно выбираются области рисунка размером 8×8 точек. Для каждой группы из 64 точек выполняется преобразование Фурье. Его назначение в том, чтобы отфильтровать (исключить) высокочастотную компоненту из исходных данных. Результаты преобразования квантуются и преобразуются в целые числа, которые кодируются по таблицам Хуффмана (D. A. Huffman) [10] и записываются в выходной массив. После обработки всего изображения выходной массив упаковывается в файл одним из стандартных способов.

При воспроизведении, после распаковки сжатого рисунка, необходимо восстановить исходное количество точек. Поэтому основное преобразование выполняется в обратном порядке. Сначала по таблицам Хуффмана, которые хранятся в файле, вычисляются приближенные результаты преобразования Фурье, а по ним коды точек. Возможно дополнительное сглаживание восстановленных результатов. После этого остается вернуться в пространство RGB.

Основные потери качества изображения происходят при квантовании результатов преобразования Фурье. Чем больше коэффициент квантования, тем больше точек будет отброшено. Точные количественные оценки, как уже говорилось, невозможны.

Сжатие полноцветных рисунков с минимальными потерями качества изображения представляет большой практический интерес. В настоящее время разработано несколько модификаций описанной схемы вычислений, направленных на сокращение потерь при сжатии. Например, в редакции JPEG, выпущенной в конце 1995 года, рекомендуется квантование результатов Фурье анализа с переменным шагом, задаваемым в таблицах. Это позволяет сохранять более подробную информацию о наиболее важных частях сжимаемого рисунка.

Файлы формата JPG. JPEG — это не стандарт хранения графических данных, а способ их сжатия. Поэтому способ хранения данных в файле зависит от стандарта, использующего JPEG для сжатия исходных рисунков. В частности, это один из нескольких способов упаковки данных, принятых в самом универсальном стандарте TIFF.

Среди пользователей более популярен стандарт JFIF, разработанный фирмой C-Cube Microsystems. Это один из основных стандартов для передачи графических данных в сети Internet. Файлы, соответствующие этому стандарту, имеют тип (расширение) JPG.

JFIF расшифровывается как JPEG File Interchange Format (формат для обмена файлами JPEG). В настоящее время наибольшее распространение получила версия JFIF 1.2, опубликованная в сентябре 1992 года. К сожалению, автору не известен перевод описания стандарта на русский язык. В случае необходимости вам придется использовать описание на английском языке, которое можно легко найти в сети Internet.

7.6. Наложение рисунков и спецэффекты

Видеорежимы direct color позволяют решать задачи, постановка которых при работе в режимах PPG была просто невозможна. К ним относится, например, получение различного рода спецэффектов, за счет манипуляций с цветами отдельных точек изображения. В данном разделе описаны способы получения некоторых спецэффектов, широко применяемых в трехмерной графике (3D-графика).

При реализации спецэффектов, как правило, выполняется большой объем арифметических операций, что существенно замедляет процесс получения результата. Поэтому в современной графике арифметические операции либо выполняются с помощью новых команд MMX, появившихся у процессоров Pentium, либо используются акселераторы, выполняющие соответствующие вычисления на аппаратном уровне (см. раздел 1.1.3). Основные графические пакеты поддерживают как программный, так и аппаратный способы вычислений, направленных на достижение нужного спецэффекта.

При выборе конкретного спецэффекта важно знать суть выполняемых преобразований изображения, а конкретный способ реализации зависит от аппаратных и программных средств, имеющихся в вашем распоряжении. Поэтому мы рассмотрим программирование спецэффектов с применением обычных команд микропроцессора и простых видеокарт.

Спрайты (Sprites). В дословном переводе с английского слово "Sprite" означает "фея" или "эльф". В литературе по программированию этот термин обозначает небольшой перемещаемый рисунок, при воспроизведении которого окружающий его фон делается прозрачным. Определение спрайта достаточно расплывчато, ему соответствуют, например, рисунки курсоров и пиктограмм, но фактически имеется в виду более широкий класс рисунков, при построении которых производится динамическое формирование маски.

Для того чтобы перемещение не портило исходную картинку на экране, надо сохранять исходный фон перед построением и восстанавливать его перед удалением спрайта. Способы сохранения и восстановления фона на месте перемещаемого рисунка мы обсуждали при описании вывода информационных строк и работы с курсором. Здесь нас будет интересовать вопрос о том, как получается прозрачный фон при построении спрайта.

Уточним, о каком фоне идет речь. Образ рисунка всегда занимает прямоугольную область, в которой основное изображение может быть окружено

фоном, не имеющим прямого отношения к рисунку. При описании работы с курсором (см. главу 6) говорилось, что одним из способов удаления ненужного фона является маскировка. Например, в файлах, содержащих образы рисунков курсоров и пиктограмм, обязательно находится маска, указывающая, какую часть образа рисунка не надо показывать на экране.

При выводе спрайта на экране получается тот же эффект, что и при наложении маски, но достигается он другим способом. Для этого специально оговаривается, что маскируемый фон должен иметь черный цвет. Если такое соглашение выполнено, то маска не нужна, она формируется программно при построении рисунка. В описании команд MMX, распространяемом фирмой Intel, приводится пример маскировки черного фона с использованием этих команд. Перевод описания на русский язык опубликован в книге [3].

Фильтрация цвета (Chroma Keying). Преобразование черного цвета в прозрачный является частным случаем фильтрации цвета, которая широко применяется в 3D-графике. В новой рекламе технологии MMX, появившейся в связи с выпуском процессора Pentium 3, приводится пример построения изображения с фильтрацией зеленого цвета.

В отличие от построения спрайта при фильтрации решается более общая задача — исключение однородного фона независимо от его цвета. В обоих случаях исключаемый цвет не должен использоваться в основной части рисунка, в противном случае в ней появятся точки с цветом точек подложки, на которую накладывается рисунок.

Для того чтобы цвет с заданным кодом оказался прозрачным, его надо просто не записывать в видеопамять. Это и делает приведенная в примере 7.29 подпрограмма, выполняющая построение строки формата bgr с фильтрацией цвета. Она отфильтровывает цвет, код которого указан в переменной bkgrcod. Эта 32-разрядная переменная должна быть описана в разделе данных задачи. Рабочий видеорежим True Color.

Перед вызовом входные параметры задаются так же, как для всех описанных подпрограмм построения строки (drawline). В регистрах fs:si указывается адрес начала строки в оперативной памяти, в di — адрес ее начала в видеопамяти, в cx — количество точек в строке. Как обычно, должно быть установлено окно видеопамяти, которому принадлежит адрес точки начала строки, а в регистре es должен находиться код видеосегмента.

Пример 7.29. Фильтрация цвета строки формата bgr, режим True Color

```
sprline: lods  word ptr fs:[si]      ; ax = коды синего и зеленого цвета
         push  ax             ; сохраняем ax в стеке
         lods  byte ptr fs:[si]      ; al = код красного цвета
         xor   ah, ah         ; очищаем резервный байт
         shl   eax, 16        ; сдвиг ax в старшую половину eax
         pop   ax             ; восстанавливаем ax из стека
```

```
        cmp   eax, bkgrcod  ; это фильтруемый цвет ?
        je    blcol         ; -> да, обходим запись нового кода
        mov   es:[di], eax  ; нет, записываем код в видеопамять
blcol:  add   di, bytppnt   ; корректируем адрес видеопамяти
        jnz   @F            ; -> адрес в пределах окна
        call  NxtWin        ; установка следующего окна
@@:     loop  splint        ; управление повторами цикла
        ret                 ; возврат из подпрограммы
```

В 1-м варианте примера 7.26 была приведена подпрограмма, выполняющая простое построение строки формата `bgr`. В отличие от нее, в данном случае код очередной точки строки не копируется в видеопамять, а помещается в регистр `eax`. Затем он сравнивается с кодом, указанным в переменной `bkgrcod`, и в случае совпадения не записывается в видеопамять, поэтому цвет соответствующей точки экрана не изменяется.

Описанную подпрограмму можно использовать при построении рисунков, хранящихся в форматах `BMP` (не упакованный) и `PCX` (упакованный). Для этого в примерах 7.25 и 7.28 достаточно просто заменить `call drawline` на `call sprline`. Замена возможна потому, что в обоих случаях используется одна и та же подпрограмма `drawline`.

Переменная `bkgrcod` должна быть описана в разделе данных задачи как двойное слово, содержащее код исключаемого цвета. Вопрос в том, как определить код прозрачного цвета — не задавать же его вручную. Если основное изображение окружает однородный фон, то можно предположить, что первая точка первой или последней строки рисунка имеет цвет фона. В таком случае, перед началом построения надо прочитать (программно) код первой точки образа рисунка и присвоить его значение переменной `bkgrcod`. Если указанное условие не выполнено, то можно с помощью графического редактора присвоить нужной точке цвет фона. В любом случае выбранный рисунок лучше предварительно просмотреть с помощью графического редактора, поскольку точки, которые воспринимаются на глаз как одноцветные, фактически могут быть разноцветными и вам придется редактировать цвет фона.

Смешение цветов (alpha blending). Наиболее общей формой наложения двух изображений является смешение их цветов с разными весовыми коэффициентами. На практике обычно используется так называемое "альфа-смешение", при котором вычисления выполняются по следующей формуле:

```
Z[i,j] = X[i,j] * alpha + Y[i,j] * (1 - alpha)
```

Здесь `X` и `Y` — смешиваемые изображения, а `Z` — результат смешения. Формула применяется к каждому базовому цвету каждой точки смешиваемых изображений, поэтому `i` изменяется от 1 до `N` (`N` — количество точек в рисунке), а `j` — от 1 до 3 (по количеству базовых цветов). Допустимые значения `alpha` находятся в пределах от 0 до 1.

Очевидно, что при alpha = 0 Z[i,j] = Y[i,j], а при alpha = 1 Z[i,j] = = X[i,j], т. е. при граничных значениях alpha в результате смешения получается одно из двух изображений. В остальных случаях результат смешения будет зависеть как от значения alpha, так и от конкретных цветов точек. Вспомните табл. 7.6, при пропорциональном изменении значений базовых цветов точки одновременно изменяются ее яркость и цвет. Предположим, что коды двух смешиваемых точек в формате rgb равны FF, 0, 0 и 0, FF, 0, т. е. одна из них окрашена в ярко-красный цвет, а другая — в ярко-зеленый. В зависимости от значения alpha цвет результата смешения будет изменяться от красного до зеленого, а его яркость сначала будет уменьшаться до 50%, а затем начнет возрастать до максимального значения. При alpha=0,5 получится чисто желтый цвет, но его яркость составит 50% от максимальной.

Особенности программной реализации. Прежде всего, давайте раскроем скобки в приведенной выше формуле и сгруппируем величины, умножаемые на alpha. В результате получится следующий вариант формулы:

```
Z[i,j] = (X[i,j] – Y[i,j]) * alpha + Y[i,j]
```

Преимущество этого варианта записи в том, что на каждом шаге вычислений вместо двух умножений выполняется одно.

Теперь о значениях alpha. Величины X[i,j], Y[i,j] и Z[i,j] являются кодами базовых цветов точек изображений, поэтому их значения заключены в пределах от 0 до 255 (или от 0 до 0FFh). Умножать такие величины на дроби неудобно, поэтому при программной реализации значения alpha задаются в виде целых чисел, изменяющихся в пределах от 0 до 255, а результат умножения делится на 256. Фактически деление на 256 не требуется, достаточно просто использовать старший байт произведения. При таком представлении граничными значениями alpha будут 0 и 255/256 = 0,9961, но с учетом реальной точности вычислений отличие верхнего предела от 1 не имеет существенного значения.

Давайте упростим задачу и рассмотрим частный случай, когда Z = Y и изображение Z находится на экране, а коды его точек, соответственно, в видеопамяти. Это позволит нам рассмотреть простой вариант подпрограммы, которая вместо смешения изображений, хранящихся в двух файлах, выполняет наложение изображения, хранящегося в файле, на изображение, находящееся на экране. Текст подпрограммы, выполняющей наложение строки формата bgr на строку, расположенную в видеопамяти, приведен в примере 7.30, предполагается, что установлен видеорежим True Color.

Входные параметры задаются так же, как для всех описанных подпрограмм построения строки (drawline):

- в регистрах fs:si указывается адрес начала строки в оперативной памяти;
- в регистре di — адрес ее начала в видеопамяти;
- в регистре cx — количество точек в строке.

Как обычно, должно быть установлено окно видеопамяти, а регистр `es` должен содержать код видеосегмента.

Значение `alpha` может изменяться от 0 до 255, оно указывается в одноименной переменной, имеющей *размер слова*, ее надо описать в разделе данных задачи, присвоив конкретное значение, или предусмотреть возможность ввода этого значения перед наложением рисунка.

Пример 7.30. Альфа-наложение строки формата bgr, режим True Color

```
alphamix: PushReg <bx,dx>; сохранение регистров bx, dx
mixpnt:   push  cx       ; сохранение счетчика повторов
          mov   cx, 03   ; для обработки трех базовых цветов
mixcol:   mov   bl, es:[di]  ; bl = код базового цвета Y[i,j]
          lods  byte ptr fs:[si]    ; al = код базового цвета X[i,j]
          xor   ah, ah   ; преобразуем байт в слово
          xor   bh, bh   ; преобразуем байт в слово
          sub   ax, bx   ; X[i,j] = X[i,j]-Y[i,j]
          imul  alpha    ; ax = (X[i,j]-Y[i,j]) * alpha; dx = 0
          xchg  al, ah   ; al = ax/256
          add   al, bl   ; Z[i,j] = (X[i,j]-Y[i,j])*alpha/256+Y[i,j]
          stosb          ; запись Z[i,j] в видеопамять
          loop  mixcol   ; управление повторами цикла
          inc   di       ; пропуск резервного байта
          jne   @F       ; -> адрес в пределах текущего окна
          call  NxtWin   ; установка нового окна
@@:       pop   cx       ; восстановление счетчика повторов
          loop  mixpnt   ; управление повторами цикла
          PopReg <dx,bx> ; восстановление регистров dx, bx
          ret            ; возврат из подпрограммы
```

Подпрограмму `alphamix` можно использовать при построении рисунков, хранящихся в форматах `BMP` (не упакованный) и `PCX` (упакованный). Для этого в примерах 7.25 и 7.28 достаточно просто заменить `call drawline` на `call alphamix`. Замена возможна потому, что в обоих случаях используется одна и та же подпрограмма `drawline`.

Основные действия в примере 7.30 выполняются во внутреннем цикле, имеющем метку `mixcol`, рассмотрим их более подробно. Две первые команды считывают в регистры `bl` и `al` коды смешиваемых базовых цветов. Перед вычитанием значения байты преобразуются в слова. Специальная команда `CBW` (convert byte to word) в данном случае не применима, т. к. она интерпретирует коды со значениями больше чем 127, как отрицательные числа. В примере 7.30 просто очищаются старшие байты регистров `ax` и `bx`, после чего выполняется вычитание (`ax = ax – bx`). Результат вычитания может быть как положительным, так и отрицательным числом, поэтому для его

умножения на значение переменной alpha используется специальная команда imul. В отличие от команды mul она интерпретирует операнды как числа со знаком. Произведение находится в регистрах dx:ax, но поскольку модули значений операндов не превышают 255, dx будет просто очищен, а в ax будет находиться младшая часть произведения. Вместо деления результата на 256 выполняется перестановка байтов регистра ax (с таким же успехом его содержимое можно было сдвинуть на 8 разрядов вправо). Остается сложить содержимое регистров al и bl и записать результат в видеопамять. Последняя команда loop mixcol трижды повторяет выполнение цикла.

После выхода из внутреннего цикла адрес видеопамяти увеличивается на 1 для пропуска резервного байта в коде точки. Как обычно, проверяется принадлежность адреса установленному окну видеопамяти и, в случае необходимости, устанавливается следующее окно. Затем в регистр cx из стека выталкивается содержимое счетчика повторов, и команда loop mixpnt управляет повторами внешнего цикла.

Очевидно, что наложение рисунка будет выполняться дольше, чем его простое построение. Более точное представление о степени замедления дают результаты следующего эксперимента. Один и тот же рисунок формата BMP, размером 640×480 точек просто строился, или накладывался на изображение, уже имеющееся на экране. В обоих случаях основная подпрограмма соответствовала примеру 7.25. Компьютер был укомплектован процессором Pentium с тактовой частотой 100 МГц. Время, затрачиваемое на наложение, увеличилось, по сравнению со временем построения, примерно на 1,1 сек. Это свидетельствует о том, что альфа-наложение можно выполнять с помощью обычных команд, но лучше использовать операции MMX или функции акселератора.

Альфа-смешение — это один из популярных приемов используемых в 3D-графике для получения различных эффектов. Рассмотрим некоторые из них.

Наплыв изображения (Image Dissolve). Если результат выводить на экран и многократно повторять смешение, увеличивая при каждом повторе значение alpha, то на фоне постепенно исчезающего исходного изображения будет все более четко проступать новое изображение. Или наоборот, если значение alpha постепенно уменьшается от 1 до 0, то одно из двух изображений будет исчезать (растворяться в другом изображении). Такие трюки довольно часто используются в оформительских целях не только в компьютерных приложениях, но и на телевидении.

Для получения нужного эффекта картинка должна изменяться медленно, поэтому наплыв небольших рисунков можно выполнять с помощью обычных вычислительных операций. Опишем, как это делается.

Прежде всего, подпрограмма примера 7.30 должна выполнять смешение, а не наложение строк. Для этого первая команда цикла mixcol (mov bl, es:[di]) заменяется на mov bl, gs:[si]. Однако такая замена возможна при условии,

что смешиваемые изображения имеют одинаковый формат. В противном случае при выборке кодов точек нельзя будет использовать один и тот же индексный регистр `es`.

При наплыве или исчезновении небольшого рисунка основное изображение находится на экране, а коды его точек в видеопамяти. Поэтому после выбора изменяемой области экрана надо сохранить копию ее исходного содержимого в сегменте оперативной памяти, указанном в регистре `gs`, в формате налагаемого рисунка. Эта копия пригодится и в том случае, если после наплыва будет производиться "растворение" рисунка.

Рисунок, подлежащий наложению, надо прочитать в сегмент оперативной памяти, указанный в регистре `fs`. Для создания эффекта наплыва организуется цикл последовательного смешения расположенных в памяти изображений с увеличением значения `alpha` от 0 до 255 с выбранным шагом. Для удаления построенного изображения можно использовать тот же цикл, предварительно обменяв содержимое сегментных регистров `gs` и `fs`.

Для упрощения задачи советуем использовать "наплывающий" рисунок формата `BMP`, но предварительно перевернуть в нем строки так, чтобы они располагались в естественном порядке, и исключить дополнительные байты, которые могут быть в конце строк. Специальная программа, выполняющая такое преобразование, может оказаться полезной во многих случаях, поэтому имеет смысл потратить время и усилия на ее составление.

Прозрачная поверхность (Transparent Surface). В этом случае создается зрительный эффект, при котором наблюдаемый объект как бы отгорожен от наблюдателя бесцветной прозрачной перегородкой, например стеклом. Для получения эффекта производится наложение однородного фона белого цвета на наблюдаемый объект. При этом степень прозрачности перегородки тем больше, чем меньше значение `alpha`, если `alpha = 1`, то перегородка превратится в прямоугольник белого цвета.

Однородный фон белого цвета, заполняющий прямоугольник нужного размера, можно подготовить с помощью любого графического редактора. Для наложения строки просто используется подпрограмма примера 7.30. Если фон перегородки однородный, что вовсе не обязательно, то рисунок фона не нужен, можно составить подпрограмму, выполняющую наложение строки заданного цвета. Ее текст приведен в примере 7.31.

Пример 7.31. Наложение прозрачной строки заданного цвета

```
Trnspl:  PushReg <bx, dx>    ; сохранение регистров bx, dx
Tline:   push  cx            ; сохранение счетчика повторов
         mov   cx, 03        ; для обработки 3-х базовых цветов
         lea   si, Trcol     ; si = адрес переменной Trcol
Tpnt:    lodsb               ; al = налагаемый базовый цвет (C[j])
         mov   bl, es:[di]   ; bl = базовый цвет изображения (Y[i,j])
```

```
        xor   ah, ah          ; преобразуем байт в слово
        xor   bh, bh          ; преобразуем байт в слово
        sub   ax, bx          ; ax = C[j] — Y[i,j]
        imul  alpha           ; ax = ax * alpha; dx = 0
        xchg  al, ah          ; al = ax/256
        add   al, bl          ; al = al + Y[i,j]
        stosb                 ; запись результата в видеопамять
        loop  Tpnt            ; трехкратное повторение цикла
        inc   di              ; пропуск резервного байта
        jne   @F              ; -> адрес в пределах текущего окна
        call  NxtWin          ; установка следующего окна
@@:     pop   cx              ; восстановление счетчика повторов
        loop  Tline           ; управление циклом преобразования строки
        PopReg <dx, bx>       ; восстановление регистров dx, bx
        ret                   ; возврат из подпрограммы
```

Все отличия подпрограммы примера 7.31 от примера 7.30 связаны с тем, что код прозрачного цвета выбирается не из файла, а из переменной `Trcol`. При каждом повторении внешнего цикла адрес этой переменной помещается в регистр `si` для использования во внутреннем цикле. `Trcol` должна быть описана в разделе данных задачи, ее размер 3 байта, в которых находятся компоненты налагаемого цвета (коды базовых цветов). Для совместимости с видеорежимом `True Color` их надо расположить в порядке `bgr`.

Следует заметить, что при работе только с белым цветом переменная `Trcol` не нужна, поскольку в этом случае коды трех базовых цветов совпадают и равны `0FFh`. Мы ввели переменную `Trcol` специально для возможности задания любого прозрачного цвета.

Затягивание изображения туманом (Fogging) относится к наиболее сложной категории спецэффектов, создаваемых с помощью альфа-наложения. Из собственного жизненного опыта вы знаете, что в тумане наше поле зрения ограничено тем больше, чем плотнее туман. Кроме того, туман рассеивает свет от различных источников, в результате чего он может казаться цветным.

Формулу для выполнения альфа-наложения тумана однородного цвета можно записать в следующем виде:

```
X[i,j] = Fc[j] * alpha[i] + X[i,j] * (1 — alpha[i])
```

Здесь `Fc` — цвет тумана, а `X` — цвет изображения, затягиваемого туманом. Запись `Fc[j]` означает, что каждая точка тумана окрашена в один и тот же цвет, это позволяет избавиться от файла, содержащего изображение тумана. В отличие от ранее приведенных формул, в данном случае у коэффициента `alpha` появился индекс `i`, т. е. значений `alpha` должно быть столько, сколько точек в основном рисунке (который затуманивается). О том, как подготовить массив значений `alpha`, мы поговорим после описания программной реализации формулы.

Если у значений `alpha` убрать индекс `i`, то получится формула для наложения прозрачного цвета, которая была реализована в подпрограмме примера 7.31. Следовательно, надо несколько изменить внешний цикл этой подпрограммы, добавив в него выборку значения `alpha` для каждой точки изображения, что и сделано в примере 7.32. Предполагается, что затягиваемое туманом изображение находится на экране.

Перед вызовом подпрограммы в регистре `di` указывается адрес начала строки рисунка в видеопамяти, а в `fs:si` — адрес начала строки коэффициентов `alpha` в буфере обмена. В `cx` задается количество точек в строке.

Пример 7.32. Наложение строки тумана заданного цвета

```
Foglin:  PushReg <bx, dx>    ; сохранение регистров bx, dx
fog_1:   push   cx           ; сохранение счетчика повторов
         lods   byte ptr fs:[si]    ; al = alpha[i]
         mov    byte ptr falpha, al ; falpha = al запоминаем alpha[i]
         push   si           ; сохраняем содержимое si
         lea    si, fogcol   ; si = адрес переменной fogcol
         mov    cx, 03       ; для обработки трех базовых цветов
fog_2:   lodsb               ; al = базовый цвет тумана (F[j])
         mov    bl, es:[di]  ; bl = базовый цвет точки (Y[i,j])
         xor    ah, ah       ; преобразуем байт в слово
         xor    bh, bh       ; преобразуем байт в слово
         sub    ax, bx       ; ax = F[j] - Y[i,j]
         imul   falpha       ; ax = ax * alpha[i]; dx = 0
         xchg   al, ah       ; al = ax/256
         add    al, bl       ; al = al + Y[i,j]
         stosb               ; сохраняем результат в видеопамяти
         loop   fog_2        ; трехкратное повторение цикла
         inc    di           ; пропуск резервного байта
         jne    @F           ; -> адрес в пределах текущего окна
         call   NxtWin       ; установка следующего окна
@@:      pop    si           ; восстановление адреса в si
         pop    cx           ; восстановление счетчика повторов
         loop   fog_1        ; управление повторами цикла
         PopReg <dx, bx>     ; восстановление регистров dx, bx
         ret                 ; возврат из подпрограммы
```

Значения `alpha` изменяются от 0 до 255, для сокращения размера файла их лучше хранить в виде байтов, но при умножении коэффициент `alpha` должен иметь размер слова. Поэтому в примере 7.32 введена специальная переменная `falpha`, имеющая размер слова. Ее надо описать в разделе данных задачи, присвоив нулевое значение.

Во внешнем цикле примера 7.32, по сравнению с примером 7.31, появились три новые команды. Первая из них считывает очередное значение `alpha[i]`

в регистр al (и увеличивает содержимое si на 1). Вторая копирует содержимое al в младший байт переменной falpha, а третья сохраняет в стеке адрес, находящийся в регистре si. Таким образом, в результате выполнения этих трех команд получено очередное значение alpha и освобожден регистр si. Теперь в него можно записать адрес переменной, содержащей цвет тумана, для использования во внутреннем цикле, это же делалось и в примере 7.31.

После выхода из внутреннего цикла восстанавливается находившийся в si адрес, и при повторе внешнего цикла будет выбрано следующее значение alpha из буфера обмена.

Для использования описанной подпрограммы надо подготовить файл, содержащий значения коэффициентов alpha. От их расположения в файле зависит способ построения рисунка, поэтому первыми должны располагаться коэффициенты для его верхней, а последними — для нижней строки. В таком случае построение будет производиться в направлении слева направо и сверху вниз, т. е. по наиболее простой схеме. Способ чтения файла в буфер обмена зависит от количества значений alpha. Например, для затягивания туманом всего изображения при разрешении 640×480 точек файл будет содержать 307 200 байтов и его придется считывать по частям. Следовательно, перед построением рисунка надо вычислить размер считываемой порции данных (количество строк и байтов в порции). Мы это делали неоднократно.

Вычисление значений alpha. Вам понадобится специальная программа, формирующая файл, содержащий значения alpha для каждой точки затуманиваемого изображения. Значения alpha зависят от расстояния (r[i]) между текущей точкой (i) и точкой принятой за начало отсчета (точкой наблюдения). В линейном двухмерном дискретном пространстве это расстояние вычисляется как квадратный корень из суммы квадратов приращений значений координат:

```
r[i] = sqrt { (x[i] – x[0])**2 + (y[i] – y[0])**2}
```

Здесь ** обозначают степень, sqrt — квадратный корень, x[0] и y[0] — значения координат в точке наблюдения, x[i] и y[i] — текущие значения координат.

В общем случае значения alpha описываются некоторой функцией (F), зависящей от расстояния, т. е. alpha[i] = F(r[i]). Независимо от природы функции F значения alpha равны нулю в точке наблюдения и увеличиваются по мере удаления от точки наблюдения, не превышая при этом значение 255 (которое принято за 1).

Советуем вам начать с простого случая, когда F является нормированной линейной функцией, у которой за норму принято значение r в максимально удаленной точке, т. е. alpha[i] = (r[i] / rmax)*255. В результате получится файл, содержащий значения коэффициентов для тумана, плотность кото-

рого возрастает пропорционально расстоянию. Эксперименты с файлом позволят вам оценить степень соответствия линейной модели ожидаемым результатам.

Кривая чувствительности человеческого глаза к яркости света нелинейная, она близка к логарифмической зависимости. Поэтому для получения более правдоподобного эффекта обычно используют экспоненциальное изменение плотности тумана. В таких случаях значения `alpha` вычисляются так:

```
alpha[i] = exp (-k*r[i]) или alpha[i] = exp {-(k*r[i])**2}
```

Замечание

Программировать вычисление значений `alpha` на языке ассемблера вовсе не обязательно. Можно использовать любой удобный для вас язык. Главное, чтобы файл имел нужную структуру и содержание.

Выбор точки отсчета зависит от характера изображения, затягиваемого туманом. Например, если это фотография какого-то объекта, ландшафта и т. п., то точку наблюдения лучше совместить с точкой фотосъемки. Обычно для выбора окончательного варианта надо произвести несколько проб, подбирая точку отсчета, цвет и характер изменения плотности тумана.

Общий случай смешения. Затягивание туманом является частным случаем смешения двух изображений с переменными значениями `alpha`. Рассмотрим простой пример. Предположим, что у вас есть рисунок сада и рисунок окна (или части комнаты с окном) и вы хотите изобразить на экране окно, выходящее в сад, причем часть окна открыта, а часть закрыта. Можно, конечно, смонтировать нужное изображение с помощью графического редактора, но нас интересует другой способ достижения результата.

Предположим, что изображение сада уже находится на экране и на него накладывается изображение окна. В таком случае нужен массив значений `alpha` для каждой точки изображения окна. Не прозрачным точкам, например рамам, подоконнику и т. п., должны соответствовать значения `alpha=1`, для полностью прозрачных точек, соответствующих открытой части окна, `alpha=0`, частично прозрачные точки (стекло) предварительно окрашиваются в белый цвет, а значение `alpha` для них подбирается в пределах от 0,25 до 0,5. Оставим в стороне вопрос о том, как составить массив значений `alpha`, предположим, что он существует.

Работать с двумя файлами не очень удобно, поэтому на практике используются комбинированные файлы, в которых коды базовых цветов каждой точки дополнены значениями `alpha` (от 0 до 255). Код точки при этом занимает не 24, а 32 разряда. В специальной литературе вы можете встретить выражение альфа-канал (Alpha Channel), которое обозначает некий источник кодов точек, в которых базовые цвета дополнены значениями `alpha`. Некоторые модели акселераторов имеют специальный альфа-буфер, для размещения

рисунков с 32-разрядными кодами точек. При работе с обычными видеокартами прикладные пакеты (DirectX и Open GL) используют такие комбинированные файлы.

Для наложения строки комбинированного файла надо внести небольшие дополнения в подпрограмму `Alphamix`, описанную в примере 7.30. В начале внешнего цикла перед командой, имеющей метку `mixcol`, надо вставить следующие две команды:

```
mov       al, es:[si+3]         ; al=значение альфа для данной точки
mov       byte ptr alpha, al    ; сохраняем его в переменной alpha
```

Кроме того, после обработки кода точки во внешнем цикле, например перед командой `inc di`, надо вставить команду `inc si` для пропуска байта, содержащего значение `alpha`.

Мы не будем обсуждать способ создания комбинированного файла, только отметим, что основная задача заключается в вычислении массива значений коэффициентов `alpha`. А добавить эти значения к кодам точек образа рисунка не составит особого труда.

Заключение. Данный раздел завершает основную часть книги. В нем описаны далеко не все спецэффекты, применяемые в современной компьютерной графике. Выбраны только те из них, способы получения которых можно отнести к основам компьютерной графики. При реализации более сложных спецэффектов используются специфические объекты трехмерной графики — треугольники, текстуры и пр. Описание работы с такими объектами выходит за рамки данной книги. Как говорил небезызвестный Козьма Прутков: "Нельзя объять необъятное".

ПРИЛОЖЕНИЕ А

Рисунки в файлах BMP

Точечные или растровые рисунки являются одной из распространенных форм представления графической информации. Существует множество различных стандартов хранения таких рисунков на внешних носителях в виде файлов. В пятерку наиболее популярных входят BMP, PCX, TIFF, GIF и JPG. Большинство современных графических редакторов поддерживает работу с файлами перечисленных форматов и позволяет преобразовывать их из одного формата в другой. Кроме того, существуют программы, предназначенные только для просмотра изображений и преобразования форматов файлов.

BMP и PCX создавались для хранения рисунков, использующих палитру цветов. Со временем в них были внесены изменения, позволяющие хранить полноцветные рисунки, но соответствующие файлы имеют большие размеры и не подходят для передачи по компьютерным сетям. В форматах GIF и JPG применяются более эффективные способы сжатия рисунков, подготовленных с использованием палитры (GIF) и без нее (JPG). Эти два формата приняты в качестве своеобразного стандарта передачи графической информации в компьютерной сети Internet.

В прикладных графических задачах целесообразно работать с файлами BMP и PCX, поскольку хранящееся в них изображение либо не упаковано (BMP), либо распаковывается достаточно просто (PCX). Немаловажен и тот факт, что в этих форматах хранится множество рисунков, предназначенных для оформления рабочей области экрана. Если же выбранный рисунок хранится в файле, имеющем тип (расширение) GIF или JPG, то его можно преобразовать в используемый задачей формат с помощью графического редактора.

BMP является основным форматом графических файлов для Windows и ее приложений, поэтому автор счел целесообразным вынести описание работы с ним в данное приложение и обсудить особенности файлов, встречающихся на практике. Имена полей заголовка взяты из справочника Борна [4].

A.1. Общая характеристика стандарта

BMP является сокращением слова Bitmap, в такой записи оно переводится как "точечное изображение", а запись bit map обычно переводят как "карта битов". В соответствии с описанием стандарта файл состоит из четырех частей: Bitmap_file, Bitmap_info, RGB_QUAD и Bitmap. Смысл этих названий следующий — данные о файле, данные об изображении, палитра используемых цветов и образ рисунка. Условимся рассматривать две первые части как заголовок файла. В таком случае файл состоит из заголовка, палитры и образа рисунка.

Существует несколько разновидностей стандарта BMP, основными из них являются стандарты для Windows и для OS/2. Они различаются размером полей, их расположением в области Bitmap_Info и способом записи кодов базовых цветов в палитре. Образ рисунка хранится в одинаковой форме.

A.1.1. Заголовок файла для Windows

Первые точечные рисунки хранились в аппаратно-зависимом формате DDB (Device Dependent Bitmap). Такое представление было крайне неудобным, и разработчики Windows 3.0 отказались от него. Для нас оно не представляет интереса, поскольку не предназначено для современных видеосистем.

Семейства Windows 3.X, 9X и 2000 используют аппаратно-независимый формат DIB (Device Independent Bitmap), заимствованный из OS/2 с некоторыми изменениями. Структура заголовка показана в табл. A.1.

Таблица A.1. *Заголовок BMP-файла для Windows*

Смещение поля	Размер в байтах	Имя поля	Назначение поля
00 (00h)	2	bfType	Метка "BM"
02 (02h)	4	bfSize	Размер файла в байтах
06 (06h)	2	Reserved	Резервное поле
08 (08h)	2	Reserved	Резервное поле
10 (0Ah)	4	BfOffBits	Смещение области данных
14 (0Eh)	4	BiSize	Размер области информации
18 (12h)	4	BiWidth	Ширина рисунка в точках
22 (16h)	4	BiHeight	Высота рисунка в точках
26 (1Ah)	2	BiPlanes	Количество плоскостей (всегда 1)
28 (1Ch)	2	BiBitCnt	Количество бит на точку

Таблица А.1 *(окончание)*

Смещение поля	Размер в байтах	Имя поля	Назначение поля
30 (1Eh)	4	BiCompr	Тип сжатия данных
34 (22h)	4	BiSizeImage	Размер данных в байтах
38 (26h)	4	biXPels/M	Количество точек на метр по оси X
42 (2Ah)	4	biYPels/M	Количество точек на метр по оси Y
46 (2Eh)	4	BiClrUsed	Количество используемых цветов
50 (32h)	4	BiClrImprt	Количество цветов в палитре

Назначение, содержание и применение основных полей заголовка будет описано в следующих подразделах, здесь мы опишем только три поля, обычно не используемые при выводе изображения на экран.

Поле biPlanes указывает количество плоскостей в рисунке. Трудно сказать, с какой целью его ввели. По крайней мере, при выводе рисунка на экран или на печать поле biPlanes не используется.

Большинство рисунков получается путем оцифровки рукотворных картинок, фотографий, слайдов, кино или видеокадров. При выводе таких рисунков на печать важно знать размер оригинала. Необходимые для этого данные находятся в полях biXPels/M и biYPels/M, где Pels означает Picture Elements (элемент рисунка или просто точка). Если содержащиеся в них величины умножить на 25,4 и разделить на 1000, то получится разрешающая способность устройства, с помощью которого была оцифрована картинка. Например, при сканировании с разрешением 300 точек на дюйм по обеим координатам на один метр приходится приблизительно $300 \cdot 1000/25{,}4 = 11\,811$ точек. Этому числу соответствует код 2E23h, который и будет храниться в описываемых полях. Учитывая возможные ошибки вычислений, реальная величина может незначительно отличаться от вычисленной.

А.1.2. Заголовок файла для OS/2

Оболочка Presentation Manager операционной системы OS/2 использует другой формат заголовка BMP-файла. Разработчики OS/2 учли избыточность стандарта BMP для Windows и сократили заголовок до минимально необходимых размеров. Его структура показана в табл. А.2. Для упрощения ссылок в ней сохранены те же обозначения полей, что и в табл. А.1, но в соответствии со стандартом OS/2 в именах полей, начиная с адреса 14 (0Eh), префикс bi изменяется на bc.

Таблица А.2. *Заголовок BMP-файла для OS/2*

Смещение поля	Размер в байтах	Имя поля	Назначение поля
00 (00h)	2	bfType	Метка "BM"
02 (02h)	4	BfSize	Размер файла в байтах
06 (06h)	2	Reserved	Резервное поле
08 (08h)	2	Reserved	Резервное поле
10 (0Ah)	4	BfOffBits	Смещение области данных
14 (0Eh)	4	BcSize	Размер области информации
18 (12h)	2	BcWidth	Ширина рисунка в точках
20 (14h)	2	BcHeight	Высота рисунка в точках
22 (16h)	2	BcPlanes	Количество плоскостей (всегда 1)
24 (18h)	2	BcBitCnt	Количество бит на точку

Информационная часть заголовка сократилась до 12 байтов, в то время как в стандарте Windows она занимала 40 байтов. Мы подчеркиваем этот факт потому, что размер информационной части заголовка является единственным критерием для определения типа BMP-файла в программе.

После заголовка располагается палитра, а затем образ самого рисунка, т. е. в целом структуры обоих типов BMP-файлов (для Windows и для OS/2) идентичны, что упрощает задачу программиста. Остается только гадать, почему программа bitmap.exe, входящая в состав Norton Commander (NC), не обрабатывает BMP-файлы для OS/2.

A.1.3. Образ рисунка в файле

Исторически стандарт BMP предназначался для Windows, а в ней при построении изображений "по умолчанию" начало координат расположено в нижнем левом углу экрана. Значения по оси X возрастают слева направо, а по оси Y — снизу вверх. На первый взгляд ничего особенного в этом нет, именно так расположены и направлены оси координат при черчении или рисовании различных графиков на бумаге. Однако это только на первый взгляд.

Расположение строк. При таком расположении осей координат последняя строка рисунка оказывается первой, а его первая строка — последней. Обычно образ рисунка записывают в файл так, чтобы его было удобно вос-

производить на экране. Разработчики BMP так и поступили — образ рисунка хранится в файле в *перевернутом виде*, сначала записана его последняя строка, за ней предпоследняя и так вплоть до первой строки, которая записана последней. В таком случае, для построения рисунка снизу вверх строки из файла считываются последовательно друг за другом.

Следует отметить, что в перевернутом виде изображение хранится во всех графических форматах, предназначенных для использования Windows и ее приложениями. В частности, в разделе 6.1.1 данной книги уже говорилось, что так хранятся рисунки курсоров (файлы типа cur) и пиктограмм (файлы типа ico).

На практике эта особенность форматов для Windows приводит к необходимости применения специальных подпрограмм, переворачивающих образ рисунка в процессе его воспроизведения.

Упаковка кодов точек. Если в образе рисунка использовано 2 или 16 цветов, то для сокращения размера файла он хранится в упакованном виде.

У 16-цветных рисунков значения кодов точек изменяются от 0 до 0Fh, поэтому в одном байте можно записать коды двух подряд расположенных точек. Код левой точки записывается в старшую тетраду, а код правой — в младшую тетраду байта.

У 2-цветных рисунков значения кодов точек изменяются от 0 до 1 и в одном байте можно записать коды восьми подряд расположенных точек. Код левой точки группы записывается в старший (седьмой), а код правой точки группы — в младший (нулевой) разряд байта.

Такой способ упаковки точек рисунков, содержащих небольшое количество цветов, применяется не только в формате BMP, но также в PCX, GIF и других форматах. В разделе 3.3.1 были приведены примеры подпрограмм 3.17 и 3.18, выполняющих распаковку в процессе построения строки рисунка.

Сжатие образа рисунка. Образы рисунков, содержащих более 2-х цветов, могут быть подвергнуты сжатию по способу RLE (Run-Length-Encoding). Прежде всего, отметим, что сжатие возможно только в формате для Windows, в формате для OS/2 оно просто не предусмотрено.

В формате для Windows (см. табл. А.1) имеется поле BiCompr, в котором указано состояние образа рисунка. Если в этом поле указан 0, то образ рисунка хранится в естественном виде. BiCompr=1 означает, что рисунок, содержащий 256 цветов, сжат по способу RLE. BiCompr=2 означает, что рисунок, содержащий 16 цветов, предварительно упакован по 2 точки в байт, а затем сжат по способу RLE.

Алгоритм декомпрессии файла, сжатого по способу RLE, следующий:

1. Если значение текущего (первого) байта отлично от нуля, то оно указывает, сколько раз надо повторить в выходном массиве код, находящийся

в следующем байте. В противном случае проверяется код следующего байта.

2. Если он больше двух (от 3 до 255), то соответствующее количество последующих байтов просто копируется в выходной массив, т. к. они не упакованы.
3. Значения второго байта 0, 1 и 2 являются признаками конца строки (0), конца рисунка (1) и изменения текущих координат (2). В последнем случае в двух следующих байтах указаны приращения значений координат X и Y, которые надо прибавить к их текущим значениям.

Таким образом, в основе упаковки по способу RLE лежит замена группы подряд расположенных одинаковых кодов двумя байтами, в первом указывается количество повторов, а во втором — повторяемый код. Сама по себе эта идея не принадлежит разработчикам стандарта BMP, они только выбирали конкретную реализацию. Аналогичный способ используется и в стандарте PCX, но его реализация несколько проще. Мы обсуждали ее в разделах 3.3.3 и 7.4.2 основной части книги.

Замечание

Автор исследовал достаточно много файлов формата BMP, но не обнаружил ни одного сжатого рисунка. Напомним также, что в формате для OS/2 сжатие просто исключено. Остается только гадать, зачем надо было описывать способ сжатия и не использовать его на практике? Ведь аналогичный способ сжатия применяется при подготовке файлов формата PCX. Мы не будем рассматривать программирование распаковки именно по причине отсутствия упакованных рисунков.

A.2. Общая схема обработки заголовка файла

Для построения на экране рисунка, хранящегося в BMP-файле, надо выполнить следующие действия:

- ввести спецификацию и открыть файл для чтения;
- прочитать заголовок и выбрать из него данные о рисунке и файле;
- установить или преобразовать палитру используемых в рисунке цветов;
- записать образ рисунка в видеопамять.

Мы уже рассматривали такую последовательность действий при описании построения рисунков, хранящихся в файлах формата PCX. Она характерна для любого файла, в структуре которого можно выделить заголовок, палитру и образ рисунка. От конкретных особенностей формата зависят лишь способы работы с отдельными частями файла.

А.2.1. Возможные отклонения от стандарта

По-видимому, стандарты пишут для того, чтобы их не соблюдали. Анализ большого количества файлов формата BMP приводит именно к такому печальному выводу. Потенциальные возможности для отклонений заложены в самом стандарте для Windows, который явно избыточен — многие величины можно просто вычислить исходя из значений других величин, а не выбирать их из полей заголовка. Именно так и поступают в большинстве случаев.

Наиболее пунктуально заполняют заголовок программы оцифровки изображений, например, выполняющие сканирование рисунков. Однако после оцифровки "оригинальные" файлы проходят достаточно сложный путь, связанный с различными преобразованиями. Очень часто рисунки редактируются с помощью графических редакторов, которые могут изменить заголовок файла по "своему усмотрению". Например редактор Paintbrush для Windows просто очищает в заголовке байты с 30-го по 50-й, что при этом теряется — можно увидеть в табл. А.1. Если такой файл импортировать в CorelDraw 4.0, то рисунок окажется искаженным, но сам Paintbrush воспроизводит его нормально. Вероятно, разработчики CorelDraw не учли возможность несоответствия заголовка файла стандарту BMP.

"Оригинал" мог быть подготовлен в одном графическом стандарте, а затем преобразован в другой, например, в стандарт BMP какой-либо программой. Такое преобразование может оказаться причиной более тонких ошибок, связанных с различием способов хранения самого рисунка. Например, по требованию стандарта PCX количество точек в строке должно быть кратно 16. Такое требование не противоречит стандарту BMP явно, но и не оговорено в нем. Если при преобразовании в BMP сохранить указанную особенность формата PCX, что обычно и делается, то рисунок будет искажен.

Стандарт для OS/2 оставляет меньше возможностей для отклонений, но одна все таки есть, — в двойном слове bfSize должен храниться размер файла. Содержимое этого слова лучше не использовать в программе. Слишком часто встречаются файлы с "потерянной" старшей частью двойного слова.

Как учесть в программе максимум возможных отклонений от стандарта? Для этого надо использовать только те величины, которые указаны в табл. А.2 (кроме bfSize) и без которых воспроизвести рисунок просто невозможно, а все остальные вычислять. Способы вычислений описаны ниже.

А.2.2. Ввод спецификации и открытие файла

Для упрощения программирования работы с файлами DOS выполняет много полезных функций. Их использование возможно только при соблюдении определенных правил манипуляций с файлами. Одно из них заключается в том, что для работы с существующим файлом его надо предваритель-

но открыть и получить идентификатор файла, необходимый для выполнения последующих действий.

Открытие существующего файла. DOS исполняет специальную функцию, предназначенную для открытия существующего файла (`Open File`). Она имеет код `3Dh` и вызывается через прерывание `int 21h`. Перед вызовом функции в регистрах `ds:dx` указывается адрес начала строки, содержащей спецификацию файла. В регистр `al` помещается код режима открытия: 0 — для чтения, 1 — для записи, 2 — для чтения и записи (для редактирования), а в регистре `ah` указывается код функции `3Dh`.

Если файл существует, то он открывается. При возврате из DOS признак переполнения (разряд `Carry` регистра флагов) очищен, а в регистре `ax` находится идентификатор файла (`file handle`), который надо сохранить в специально выделенной переменной (мы обозначали ее `handle`). Идентификатор нужен DOS для работы с конкретным файлом. По существу, это порядковый номер файла, открытого задачей.

Если при возврате из DOS флаг переполнения установлен, то файл не был открыт по каким-то причинам. Наиболее вероятно из-за ошибки в тексте спецификации. Поэтому задача должна проверять состояние разряда C и выводить аварийное сообщение, если он установлен.

Напомним, что спецификация должна содержать данные, необходимые для поиска файла на диске, например, C:\Windows\Clouds.bmp. При обработке спецификации DOS преобразует строчные буквы в заглавные, поэтому можно использовать буквы любого размера. Если путь для поиска не указан, то DOS ищет файл в текущем каталоге. Признаком конца спецификации является пустой байт.

Подпрограмма открытия файла. Способ ввода спецификации зависит от установленного видеорежима и от формы диалога, поддерживаемого задачей. Предположим, что задача установила один из видеорежимов VESA, а для диалога с оператором используются информационные строки. То есть на экран выводится подсказка оператору, и задача переходит в режим ввода данных с клавиатуры. Подпрограммы, необходимые для вывода информационных строк и ввода данных с клавиатуры рассмотрены во второй части главы 5 данной книги. Поэтому мы возьмем за основу описанный там пример 5.29 и дополним его действиями, необходимыми для открытия файла.

Текст подпрограммы приведен в примере А.1. Перед ее вызовом в регистрах `ds:si` надо указать адрес подсказки оператору, содержащей текст типа `"Введите спецификацию файла >"`, он должен заканчиваться пустым байтом. После возврата из подпрограммы проверяется состояние C-разряда и содержимое регистра `ax`. Если C-разряд очищен, а содержимое `ax` отлично от 0, то оно является идентификатором файла.

Пример А.1. Подпрограмма ввода спецификации и открытия файла

```
GetSpec: push   Cur_win      ; сохранение исходного значения Cur_win
         mov    ax, Inflinw  ; ax = номер окна информационной строки
         mov    Cur_win, ax  ; Cur_win = ax
         call   Savinfo      ; сохранение исходного фона
         jmp    short outstr ; переход на выборку первого символа
out1:    call   outsgn       ; вывод на экран очередного символа
outstr:  lodsb               ; al = код очередного символа (al = ds:si)
         or     al, al       ; конец выводимого текста ?
         jne    out1         ; -> нет, переход на метку out1
         call   Inline       ; ввод строки теста с клавиатуры
         call   Delinfo      ; удаление информационной строки с экрана
         pop    Cur_win      ; восстановление исходного значения Cur_win
         call   setWin       ; восстановление исходного окна
         mov    al, Linbuf   ; al = первый байт строки
         or     al, al       ; спецификация введена?
         jnz    OpenFr       ; -> нет, пустая строка
         xor    ah, ah       ; очистка регистра ah
         ret                 ; возврат из подпрограммы
OpenFr:  lea    dx, Linbuf   ; dx = адрес начала спецификации файла
         mov    ax, 3D00h    ; al = 0, ah = 3D — код функции
         int    21h          ; обращение к DOS для открытия файла
         ret                 ; возврат из подпрограммы
```

Первые 13 команд подпрограммы повторяют текст примера 5.29. В этой части на экран выводится подсказка оператору, а в буфер `Linbuf` записываются вводимые с клавиатуры символы. После нажатия оператором на клавишу <Enter> в `Linbuf` записывается пустой байт, ввод прекращается, подсказка и ответ оператора удаляются с экрана, а на их месте восстанавливается исходное изображение.

Дополнительные действия начинаются с анализа содержимого первого байта буфера `Linbuf`. Если он пуст, то просто очищается регистр `ax` и происходит возврат из подпрограммы. Это предусмотрено на тот случай, если оператор раздумает вводить спецификацию и просто нажмет клавишу <Enter>.

Если первый байт `Linbuf` не пустой, то предполагается, что оператор ввел спецификацию. В таком случае в регистр `dx` загружается адрес `Linbuf`, в регистр `ax` записываются код функции `3Dh` и признак открытия файла для чтения (`00`). После этого происходит обращение к DOS через прерывание `int 21h` и возврат из подпрограммы.

Если файл успешно открыт, то после возврата из подпрограммы C-разряд очищен, а содержимое `ax` отлично от нуля, его надо сохранить в переменной `handle` и можно читать заголовок файла.

A.2.3. Чтение заголовка файла и палитры

Действия, предшествующие построению рисунка, описаны ниже по шагам. На каждом шаге приводится обоснование действий и их программная реализация. Это сделано для того, чтобы описание и программирование конкретных действий были расположены близко друг к другу.

Описание двух первых шагов. После того как файл открыт, в буфер обмена считываются его заголовок и палитра используемых цветов. В разделе 3.3.2 мы условились, что сегмент, содержащий буфер обмена, хранится в переменной `SwpSeg`, а смещение в нем — в переменной `SwpOffs`. Чтение порции данных в буфер обмена выполняет подпрограмма `Readf`, описанная в примере 3.23. Она располагает считанные данные в сегменте `SwpSeg`, начиная с адреса (смещения), указанного в `SwpOffs`. Чтение заголовка и палитры производится в два приема.

Шаг 1. Считываем из файла первые 16 байтов и располагаем их в начале буфера обмена, начиная с адреса `0`. Проверяем содержимое двух первых байтов буфера. В них должны находиться коды заглавных латинских букв `BM` (`42h` и `4Dh`). Если это не так, то файл не соответствует стандарту `BMP` и его обработка не имеет смысла.

Если в начале файла находятся буквы `BM`, то слово 10 (`0Ah`) указывает смещение образа рисунка от начала файла (`biOffBits`). Напомним, что образ рисунка расположен после палитры, поэтому значение, указанное в слове 10, можно использовать для вычисления размера второй порции данных.

Шаг 2. Читаем следующие 16 байтов (`biOffBits`) и располагаем их в буфере обмена начиная с 16-го байта (счет начинается с нуля), т. е. сразу после данных, прочитанных на первом шаге. После чтения в буфере находятся заголовок файла и палитра. В зависимости от принадлежности файла стандарту Windows или OS/2, размер и структура заголовка соответствуют табл. A.1 или A.2.

Программная реализация двух описанных шагов показана в примере A.2. Это начало большой подпрограммы, которую мы будем описывать по частям. Входные параметры у нее отсутствуют, но перед обращением должен быть открыт файл, содержащий выбранный вами рисунок, а его идентификатор помещен в переменную `handle`, которая используется в подпрограмме `Readf`.

Пример A.2. Начало обработки заголовка BMP-файла

```
BitMap:  mov   cx, 16          ; размер порции для чтения
         mov   SwpOffs. 0      ; адрес начала считываемых данных
         call  Readf           ; чтение первых 16-ти байтов файла
         jnc   FilType         ; -> чтение без ошибок
         ret                   ; возврат при ошибке чтения
```

```
FilType: xor   si, si           ; очистка регистра si
         mov   fs, SwpSeg       ; fs = сегмент буфера обмена
         mov   ax, fs:[si]      ; ax = метка файла ("BM")
         cmp   ax, 4D42h        ; файл типа BMP?
         je    bmpfil           ; -> да
         stc                    ; нет, установка C-разряда
         ret                    ; возврат, файл не BMP
bmpfil:  mov   cx, fs:[si+0Ah]  ; cx = смещение области данных
         sub   cx, 16           ; cx = cx — 16, порция для чтения
         mov   SwpOffs, 16      ; адрес начала считываемых данных
         call  Readf            ; чтение остатка заголовка и палитры
         jnc   part_2           ; -> чтение без ошибок
         ret                    ; возврат при ошибке чтения
```

В примере А.2 выполняются достаточно простые действия, поэтому мы не будем останавливаться на их подробном описании.

Теперь нужно определить величины, необходимые для построения рисунка, и выполнить манипуляции, связанные с обработкой палитры. Вообще говоря, порядок дальнейших действий не имеет принципиального значения, но мы начнем с определения характеристик изображения.

А.2.4. Анализ основных полей заголовка

Для построения рисунка надо знать его ширину, высоту и размер строки в файле. Следует подчеркнуть, что ширина рисунка и размер строки в файле являются разными величинами и их значения чаще всего не совпадают.

Будем считать, что в разделе данных задачи описаны переменные:

```
iwidth   dw 0  ; количество точек в строке рисунка
iheight  dw 0  ; количество строк в рисунке
fwidth   dw 0  ; количество байтов в строке в файле
rmndr    dw 0  ; количество дополнительных байтов в строке файла
bitcnt   db 0  ; количество разрядов в коде точки
```

Первые четыре переменные мы использовали в основной части книги при описаниях построения различных рисунков. Пятая переменная нужна для временного хранения размера кода точки, чтобы не выбирать его каждый раз из заголовка файла. При описании переменные очищаются, их конкретные значения определяются в процессе обработки заголовка. В нем явно указаны значения только трех переменных, а `fwidth` и `rmndr` приходится вычислять.

Описание следующего шага. Расположение полей, содержащих значения для переменных `iwidth`, `iheight` и `bitcnt`, зависит от того, какому формату соответствует заголовок — Windows или OS/2. Напомним, что эти форматы раз-

личаются размером информационной части заголовка, указанным в поле biSize или bcSize (его смещение 0Eh).

Шаг 3. Выбираем содержимое полей заголовка biWidth, biHeight и biBitCnt с учетом значения, указанного в поле biSize, и сохраняем выбранные величины в переменных iwidth, iheight и bitcnt. В заключение проверяем значение bitcnt, и если оно равно 18h (24), то обработка заголовка закончена, и выполнение подпрограммы Bitmap завершается. В соответствии со стандартом BMP поле biBitCnt может содержать значения 1, 4, 8 или 18h. Последнее значение соответствует полноцветным рисункам, не использующим палитру цветов, поэтому никакие манипуляции с палитрой не требуются, что и позволяет просто завершить подпрограмму.

Полученные на этом шаге величины содержат исчерпывающую исходную информацию и позволяют вычислить все, что нужно для построения рисунка. В конце раздела А.2.1 говорилось, что при работе с BMP-файлами доверять можно только этим величинам. Значения, содержащиеся в других полях заголовка, могут быть ошибочными.

Программная реализация описанного шага показана в примере А.3, который является продолжением примера А.2.

Пример А.3. Формирование исходных значений основных переменных

```
Part_2: mov   ax, fs:[si+12h]  ; ax = biWidth, формат Windows и OS/2
        mov   bx, fs:[si+16h]  ; bx = biHeight, формат Windows
        mov   cx, fs:[si+ 1Ch] ; cx = biBitCnt, формат Windows
        cmp   byte ptr fs:[si+0Eh], 28h; заголовок формата Windows ?
        je    @F               ; -> да, переходим на запись значений
        mov   bx, fs:[si+14h]  ; bx = biHeight, формат OS/2
        mov   cx, fs:[si+18h]  ; cx = biBitCnt, формат OS/2
@@:     mov   iwidth, ax       ; iwidth = biWidth
        mov   fwidth, ax       ; fwidth = biWidth
        mov   iheight, bx      ; iheight = biHeight
        mov   bitcnt, cl       ; bitcnt = biBitCnt
        cmp   cl, 18h          ; cl = 18h ?
        jne   Part_3           ; -> нет, переход на продолжение
        ret                    ; возврат из подпрограммы
```

При выполнении примера А.3 переменным iwidth и fwidth присваиваются одинаковые значения, но в общем случае они различаются. Поэтому цель дальнейших шагов — уточнить значение fwidth.

Здесь уместно напомнить, что в стандарте PCX значение переменной fwidth хранится в заголовке файла и вычислять его не надо (см. раздел 3.3.3).

Наиболее простой способ определения значения fwidth следующий. В соответствии со стандартом в полях заголовка хранятся размер файла и адрес

начала образа рисунка. Размер образа рисунка в байтах равен разности указанных величин. При делении этой разности на количество строк получается размер строки в байтах.

К сожалению, этот способ на практике не применим. Проверка достаточно большого количества файлов показала, что поле `bfSize` (его смещение 02), в котором должен находиться размер файла в байтах, может содержать ошибочное значение или нуль. Размер файла можно определить программно, но для этого придется обращаться к DOS со специальными запросами.

Следующие 4 шага. Мы опишем более простой способ, основанный на том, что адрес начала строки в файле должен быть кратен четырем. Поэтому если количество точек в строке рисунка не кратно четырем, то при записи в файл после каждой строки добавляется необходимое количество байтов, содержимое которых не определено. При построении образа рисунка эти байты не используются, их просто пропускают. Таким образом, одна из причин различия значений переменных `iwidth` и `fwidth` связана с возможным наличием дополнительных байтов в строке файла. Кроме того, вследствие упаковки точек 2- и 16-цветных рисунков, строка в файле будет в 2 или в 8 раз короче строки рисунка (без учета дополнительных байтов).

Для получения нужного результата выполняются следующие 4 шага.

Шаг 4. Предполагаем, что `fwidth = iwidth`. Округляем это значение до ближайшего целого числа, кратного четырем. Очищаем константу сдвига. Если `bitcnt = 8`, то переходим к шагу 7, иначе следующий шаг.

Шаг 5. Полученное на предыдущем шаге значение округляем до ближайшего целого, кратного 8. Присваиваем константе сдвига значение 1. Если `bitcnt = 4`, то переходим к шагу 7, иначе следующий шаг.

Шаг 6. Если мы дошли до этого шага, то `bitcnt = 1`, т. е. рисунок двухцветный (но не обязательно черно-белый). Округляем полученное на предыдущем шаге значение `fwidth` до ближайшего целого, кратного 32. Константе сдвига присваиваем значение 3.

Шаг 7. Округленное значение сдвигаем вправо на константу, значение которой выбиралось на предыдущих шагах, и присваиваем его переменной `fwidth`. Вычисляем количество дополнительных байтов в строке файла (значение переменной `rmndr`). Нужные величины сформированы.

Программная реализация описанных шагов показана в примере А.4, который является продолжением примеров А.2 и А.3.

Пример А.4. Вычисление значений переменных fwidth и rmndr

```
Part_3:  xor   cl, cl        ; cl = 0, значение константы сдвига
         add   ax, 03        ; ax = ax + 3 (ax содержит iwidth)
```

```
        and   al, 0FCh      ; очищаем 2 младших разряда ax
        cmp   bitcnt, 08    ; bitcnt = 8 ?
        je    @F            ; -> да, переход на локальную метку @@
        add   ax, 04        ; ax = ax + 4
        and   al, 0F8h      ; очищаем 3 младших разряда ax
        inc   cl            ; cl = 1, значение константы сдвига
        cmp   bitcnt, 04    ; bitcnt = 4 ?
        je    @F            ; -> да, переход на локальную метку @@
        add   ax, 18h       ; ax = ax + 24
        and   al, 0E0h      ; очищаем 5 младших разрядов ax
        mov   cl, 03        ; cl = 3, значение константы сдвига
@@:     mov   bx, ax        ; bx = ax (округленное значение iwidth)
        sub   bx, iwidth    ; bx = bx — iwidth
        shr   bx, cl        ; сдвиг bx вправо на содержимое cl
        mov   rmndr, bx     ; количество дополнительных байтов
        shr   ax, cl        ; сдвиг ax вправо на содержимое cl
        mov   fwidth, ax    ; сохраняем значение fwidth
```

Рисунок не помещается на экране. Вполне вероятно, что размеры (или один из размеров) рисунка превышают размеры рабочей области экрана, соответствующие установленному видеорежиму. В таких случаях возможны, по крайней мере, три варианта действий, не считая отказа от построения рисунка:

- ❐ выводится только часть рисунка, помещающаяся на экране;
- ❐ увеличивается логический размер строки (раздел 1.2.2, функция `4F06h`);
- ❐ устанавливается видеорежим с большим геометрическим разрешением.

В двух первых случаях на экране будет видна только часть, а в третьем случае — все изображение, если удастся подобрать подходящий видеорежим. Наиболее универсален второй способ, но для получения всех его преимуществ в задачу надо включить механизм перемещения области видеопамяти, отображаемой на экране. В приложениях для Windows таким механизмом являются горизонтальный и вертикальный "лифты".

Изменять установленный видеорежим или его характеристики в описываемой здесь подпрограмме не целесообразно, это надо делать на более высоком уровне. Если же вас устраивает построение части рисунка, размер которой зависит от установленного видеорежима, то в данной подпрограмме можно принудительно изменить значения `iwidth` и `iheight`.

Дополнение к примеру А.3. Для выполнения описанных действий, в тексте примера А.3 команда, выполняющая проверку размера кода точки (`cmp cl, 18h`), заменяется группой команд, приведенных в примере А.5.

Пример А.5. Ограничение значений переменных iwidth и iheight

```
        mov   bx, horsize    ; bx = размер экрана по горизонтали
        cmp   bx, ax         ; iwidth > horsize ?
        jae   @F             ; -> нет
        mov   iwidth, bx     ; iwidth = horsize (уменьшаем iwidth)
@@:     mov   bx, versize    ; bx = размер экрана по вертикали
        cmp   bx, iheight    ; iheight > versize ?
        jae   @F             ; -> нет
        mov   iheight, bx    ; iheight = versize (уменьшаем iheight)
@@:     cmp   cl, 18h        ; cl = 18h ?
```

При построении рисунка с таким ограничением значений `iwidth` и `iheight` на экране будет видна его левая нижняя часть размером `horsize*versize`. Мы не включили эти команды в текст примера А.3 потому, что ограничение размера рисунков не относится к основным действиям, выполняемым при обработке заголовка файла.

А.2.5. Манипуляции с палитрой

Для окончания обработки заголовка надо извлечь из него величины, необходимые для установки или преобразования палитры цветов.

Палитра применяется в тех случаях, когда цвет не указан в коде точки (`biBitCnt` = 1, 4 или 8), она содержит описание используемых в рисунке цветов и состоит из строк, количество которых изменяется от 2 до 256. Строка содержит полный код цвета и занимает 3 (формат OS/2) или 4 (формат Windows) байта. Базовые цвета расположены в строке в следующем порядке: синий, зеленый, красный, в формате Windows к ним добавлен пустой байт, т. е. строки хранятся в форматах `bgr` или `bgr0`.

Характеристики палитры. Для работы с палитрой надо вычислить значения трех величин, которые мы обозначим как `paddr` — адрес начала палитры в буфере обмена, `pnline` — количество строк (цветов) в палитре и `pbpl` — количество байтов в строке палитры. Они определяются так:

Шаг 8. Вычисляем адрес начала палитры (`paddr`) как сумму значения поля `iSize` и смещения этого поля (оно равно `0Eh`). Размер палитры в байтах вычисляется как разность содержимого поля `bfOffBits` (его смещение `0Ah`) и `paddr`. В зависимости от формата заголовка (Windows или OS/2) присваиваем переменной `pbpl` значение 4 или 3. Для определения формата заголовка проверяем значение `paddr`, если оно равно `36h`, то это формат Windows, в противном случае — OS/2. Вычисляем `pnline = paddr/pbpl`.

Вычисление характеристик палитры показано в примере А.6.

Пример А.6. Определение характеристик палитры

```
      mov   di, fs:[si+0Eh]  ; di = размер Bitmap_info
      add   di, 0Eh          ; di = di + 0Eh, di содержит paddr
      mov   ax, fs:[si+0Ah]  ; ax = адрес начала образа рисунка
      sub   ax, di           ; ax = размер палитры в байтах
      mov   bx, 04           ; полагаем pbpl = 4
      cmp   di, 36h          ; заголовок формата Windows ?
      jz    @F               ;-> да
      dec   bl               ; pbpl = 3
@@:   xor   dx, dx           ; перед делением очищаем dx
      div   bx               ; ax = количество строк в палитре
      mov   paddr, di        ; !! сохраняем paddr (не обязательно)
      mov   pnline, ax       ; !! сохраняем pnline (не обязательно)
      mov   pbpl, bx         ; !! сохраняем pbpl (не обязательно)
```

Вычисленные величины сохраняются только в том случае, если работа с палитрой будет производиться в другой подпрограмме. При этом переменные `paddr`, `pnline` и `pbpl` надо описать в разделе данных задачи. Если же работа с палитрой выполняется в продолжении описываемой подпрограммы, то перечисленные переменные не нужны, мы их вводили только для удобства описания.

После выполнения примера А.6 из заголовка получена информация, необходимая для построения рисунка и работы с палитрой. Если вы не планируете включить установку или преобразование палитры в описываемую здесь подпрограмму, то в конец текста примера А.6 добавьте команду `ret`.

Дальнейшие манипуляции с палитрой зависят от установленного задачей видеорежима. Если это один их режимов `PPG`, то ее надо установить, а если это один из режимов `direct color`, то преобразовать в таблицу цветов, которая нужна для построения рисунка.

Установка палитры заключается в записи перечисленных в ней цветов в регистры цвета видеокарты (`DAC`-регистры). Эти регистры доступны только при работе в видеорежимах `PPG`. При простой установке в регистры записываются коды всех цветов без их предварительного анализа. Недостаток такого способа в том, что в результате установки палитры добавляемого рисунка небольшого размера могут измениться цвета точек основного изображения, находящегося на экране. Для более рационального использования регистров видеокарты перед установкой палитры могут выполняться анализ и преобразование добавляемых цветов. Эти вопросы подробно обсуждались в разделах 4.3, 4.4 и 4.5 основной части книги. Здесь мы ограничимся простым примером.

В разделе 4.4 описан способ простой установки палитры, хранящейся в файлах формата `PCX`. Рассмотрим аналогичные действия для `BMP`-файлов.

Напомним, что для записи кодов базовых цветов в регистры цвета видеокарты используется одна из функций прерывания `int 10h`. Существуют две ее разновидности — для записи кода в один регистр и для записи кодов в группу регистров. Нас будет интересовать последняя, поскольку она позволяет установить всю палитру за одно обращение к `BIOS`.

Подробное описание функций `BIOS` приведено в разделе 4.3. Нужная нам функция имеет код `12h`. Перед ее вызовом надо подготовить палитру так, чтобы в каждой ее строке было записано по 3 байта, содержащих базовые цвета, расположенные в порядке `rgb`, причем коды цветов должны быть сокращены до шести разрядов (два старших разряда кода `BIOS` игнорирует).

Программная реализация преобразования и простой установки палитры показана в примере А.7, который является продолжением примера А.6.

Пример А.7. Преобразование и установка палитры

```
convert: mov   cx, ax         ; cx = количество строк в палитре
         mov   si, di         ; si = di адрес начала палитры
         push  es             ; сохраняем содержимое es
         mov   es, SwpSeg     ; es = сегмент буфера обмена
         push  di             ; сохраняем адрес начала палитры
         push  cx             ; сохраняем количество строк
s_lp:    mov   al, fs:[si]    ; al = код синего цвета
         xchg  al, fs:[si+2]  ; перестановка кодов красного и синего
         shr   al, 02         ; сокращаем код красного до 6 разрядов
         stosb                ; es:di = al, di = di + 1
         mov   ax, fs:[si+1]  ; ax = коды зеленого и синего цвета
         shr   ax, 02         ; сокращаем коды до 6 разрядов
         and   al, 3Fh        ; очищаем в al 2 старших разряда
         stosw                ; es:di = ax, di = di + 2
         add   si, bx         ; si = si + pbpl (pbpl = 3 или 4)
         loop  s_lp           ; управление повторами цикла
setpal:  pop   cx             ; восстанавливаем количество строк
         pop   dx             ; восстанавливаем адрес палитры
         xor   bx, bx         ; номер первого DAC-регистра
         mov   ax, 1012h      ; ax = код запрашиваемой функции
         int   10h            ; BIOS устанавливает палитру
         pop   es             ; восстанавливаем содержимое es
         ret                  ; возврат из подпрограммы
```

Напомним, что после выполнения примера А.6 в регистрах находятся следующие величины: `ax` — количество строк в палитре, `bx` — размер строки в байтах, `di` — адрес начала палитры. Выполнение примера А.7 начинается с формирования содержимого регистров, используемых при преобразовании

палитры. Для этого в cx копируется содержимое ax, а в si — содержимое di. Исходное содержимое es сохраняется и в него записывается код сегмента буфера обмена. Адрес и количество строк будут нужны при установке палитры, поэтому содержимое регистров cx и di сохраняется в стеке.

Цикл преобразования имеет метку s_lp. В нем производятся перестановка красного и синего и сокращение кодов базовых цветов до 6-ти разрядов. Первая команда цикла считывает код синего базового цвета в регистр al, а вторая команда переставляет содержимое регистра al и байта со смещением 2. В результате в al оказывается код красного цвета, а в байте со смещением 2 — код синего цвета. Содержимое регистра al сдвигается на 2 разряда вправо (сокращается до 6-ти разрядов) и записывается в буфер обмена. Зеленый и синий цвета обрабатываются совместно. Их коды считываются в регистр ax, его содержимое сдвигается на 2 разряда вправо, и очищаются два старших разряда al. Результат записывается в буфер обмена. Описанные действия повторяются для каждой строки палитры.

Преобразованная палитра помещается на место исходной. Если заголовок файла соответствует формату Windows, то она окажется короче исходной за счет исключения пустых байтов.

После цикла преобразования, начиная с команды, имеющей метку setpal, выполняется установка палитры. Размер палитры выталкивается из стека в регистр cx, а адрес ее начала в dx. Для записи, начиная с нулевого DAC-регистра, bx очищается. В регистр ax помещается код запроса и происходит обращение к BIOS для установки палитры. После возврата из BIOS из стека восстанавливается исходное значение es и подпрограмма BitMap завершается, поскольку она выполнила все предусмотренные действия.

Построение таблицы цветов. Если задача работает в одном из режимов direct color, то регистры цвета видеокарты не используются. В таких случаях исходная палитра преобразуется в таблицу, содержащую коды базовых цветов точек в формате, соответствующем установленному видеорежиму. Другими словами, надо преобразовать строки исходной палитры, упаковав базовые цвета в 16-разрядное слово для режимов Hi-Color или в 32-разрядное слово для режимов True Color. Таблица цветов используется для преобразования кодов точек при построении рисунка.

Способы построения таблицы цветов описаны в разделах 7.3.1 (для режимов Hi-Color) и 7.3.2 (для режимов True Color). Здесь мы ограничимся примерами для режимов True Color.

В указанных разделах предполагалось, что таблица цветов размещается в буфере общего назначения. Код сегмента, содержащего этот буфер, хранится в переменной GenSeg, а адрес свободного пространства в нем находится в переменной GenOffs. Таким образом, пара переменных GenSeg и GenOffs задает полный адрес начала таблицы цветов в оперативной памяти.

Если заголовок файла соответствует формату Windows, то исходная палитра просто копируется в буфер общего назначения. А если заголовок файла соответствует формату OS/2, то при копировании палитры к базовым цветам каждой строки надо добавить пустой байт.

Программная реализация построения таблицы цветов показана в примере А.8, который является продолжением примера А.6. Предполагается, что в регистрах находятся следующие данные: `ax` — количество строк в палитре, `bx` — размер строки в байтах, `di` — адрес начала палитры.

Пример А.8. Построение таблицы цветов для режимов True Color

```
TabCol: mov   si, di           ; si = адрес начала палитры
        push  es               ; сохраняем содержимое es
        les   di, dword ptr GenOffs; es:di = адрес таблицы цветов
        sub   bx, 03           ; уменьшаем содержимое bx на 3
        mov   cx, ax           ; cx = количество строк в палитре
modcol: movs  word ptr [di], fs:[si]; копируем 2 младших байта
        lods  byte ptr fs:[si] ; читаем в регистр al третий байт
        xor   ah, ah           ; очищаем старший байт регистра ax
        stosw                  ; записываем 2 старших байта
        add   si, bx           ; корректируем адрес для чтения
        loop  modcol           ; управление повторами цикла
        pop   es               ; восстановление содержимого es
        ret                    ; возврат из подпрограммы BitMap
```

Пример А.8 является несколько измененным вариантом примера 7.19. При выполнении подготовительных действий добавлено формирование содержимого регистров `bx`, `cx` и `si`, а в цикл построения таблицы включена команда `add si, bx` для пропуска пустого байта, если заголовок соответствует формату Windows.

Если ваша задача устанавливает один из режимов `Hi-Color`, то возьмите за основу пример 7.18 и внесите в него аналогичные изменения.

Заключение. Основные действия, выполняемые при обработке заголовка файла, описаны в примерах А.2—А.4 и А.6. Если манипуляции с палитрой будет выполнять специализированная подпрограмма, то после выполнения примера А.6 надо завершить выполнение подпрограммы `BitMap`. Если же `BitMap` устанавливает палитру или формирует таблицу цветов, а это желательно, то она завершается после выполнения соответствующих действий. В таком случае при выходе из `BitMap` использованы все данные, находящиеся в буфере обмена, и его содержимое больше не нужно.

A.3. Построение рисунков, использующих палитру

Если задача выполнила действия, описанные в предыдущих разделах, то остается только указать адрес начала рисунка в видеопамяти и можно начинать процедуру его построения. При работе с файлами формата BMP достаточно иметь одну универсальную процедуру, выполняющую построение изображения снизу вверх. Тем не менее, мы рассмотрим вариант построения сверху вниз с одновременным переворотом рисунка.

Нас будут интересовать универсальные процедуры построения рисунка, текст которых не зависит от установленного в задаче видеорежима. В первую очередь от него зависит размер кодов точек и, как следствие, размер строки и значение константы переадресации строк видеопамяти. В разделе 7.2 были введены следующие переменные, содержащие характеристики видеорежима:

```
bperline - размер отображаемой на экране строки в байтах
bytppnt  - размер кода точки, выраженный в байтах
wrdppnt  - размер кода точки, выраженный в словах
```

Они используются в описанных ниже подпрограммах для автоматической настройки на установленный задачей видеорежим.

От видеорежима зависит не только размер кода точки, но и расположение в нем базовых цветов. Учет этих двух факторов будет производиться в подпрограммах нижнего уровня, выполняющих запись кодов точек рисунка в видеопамять. За счет этого достигается универсальность подпрограмм, выполняющих как построение рисунка в целом, так и его отдельных строк.

A.3.1. Построение рисунка сверху вниз

Здесь описан способ построения рисунка, при котором строки его образа выбираются в обратном порядке (начиная с последней строки), а на экран выводятся в естественном порядке сверху вниз.

Исходные предпосылки. Сразу отметим, что такой способ построения не является универсальным. Его можно использовать для воспроизведения рисунков небольшого размера, образы которых помещаются в одном сегменте оперативной памяти (не превышают 65 536 байтов).

При работе с большими файлами его использовать не целесообразно по следующей причине. Файл является структурой с последовательным доступом к данным. Это значит, что для обработки конкретной строки надо либо прочитать все предыдущие строки, либо просто пропустить их, выполнив принудительное позиционирование файла на начало нужной строки. Позиционирование связано с дополнительными обращениями к DOS. При обра-

ботке строк в обратном порядке потребуется многократное позиционирование файла, что существенно замедлит построение рисунка. Поэтому большие рисунки лучше строить по другой схеме.

При выборке строк из оперативной памяти в обратном порядке каждый раз надо вычислять адрес начала предыдущей строки. Возможны разные способы таких вычислений. Например, можно зарезервировать специальную переменную, содержащую адрес последней обработанной строки и уменьшать этот адрес для доступа к новой строке. В таком случае для вычисления адреса нужны три команды (пересылка, вычитание, пересылка). Мы покажем, как можно обойтись без специальной переменной.

Подпрограмма Smlbmp. Текст подпрограммы, выполняющей построение небольшого рисунка формата `BMP` описанным способом, приведен в примере А.9. Перед ее вызовом адрес левого верхнего угла рисунка в видеопамяти помещается в регистр `di` и устанавливается окно, которому принадлежит этот адрес. Регистр `es` должен содержать код видеосегмента. В регистре `si` указывается размер образа рисунка в байтах.

Пример А.9. Построение рисунка формата BMP сверху вниз

```
Smlbmp:  pusha               ; сохранение "всех" регистров
         PushReg <fs,gs,Cur_win>; сохранение fs, gs, Cur_win
         mov   ax, horsize   ; ax = horsize
         sub   ax, iwidth    ; ax = horsize – iwidth
         mul   bytppnt       ; ax = (horsize – iwidth)* bytppnt, dx = 0
         mov   dx, ax        ; dx = ax, для коррекции адреса строки
         mov   cx, si        ; cx = si, размер образа рисунка
         mov   SwpOffs, 0    ; начало считываемых данных
         call  readf         ; чтение образа рисунка
         jnc   ok            ;-> чтение без ошибок
;        Здесь должны выполняться действия в случае ошибки при чтении
ok:      mov   fs, SwpSeg    ; fs = сегмент буфера обмена
         mov   gs, GenSeg    ; !! gs = сегмент таблицы цветов
         mov   cx, iheight   ; cx = количество строк в рисунке
invout:  push  cx            ; сохраняем счетчик повторов
         mov   cx, iwidth    ; cx = размер строки рисунка
         sub   si, fwidth    ; адрес начала новой строки
         call  drawline      ; !! или call bx – построение строки
         pop   cx            ; восстанавливаем счетчик повторов
         add   si, rmndr     ; учитываем "лишние байты"
         sub   si, fwidth    ; адрес начала построенной строки
         add   di, dx        ; коррекция адреса видеопамяти
         jnc   @F            ; -> адрес в пределах сегмента
         call  NxtWin        ; установка следующего окна
```

```
@@:     loop  invout              ; управление повторами цикла
        PopReg <Cur_win,gs,fs>    ; восстановление Cur_win, gs, fs
        popa                      ; восстановление "всех" регистров
        call  setwin              ; восстановление исходного окна
        ret                       ; возврат из подпрограммы
```

Выполнение примера А.9 начинается с сохранения в стеке содержимого регистров общего назначения, сегментных регистров `fs`, `gs` и переменной `Cur_win`. Следующие четыре команды вычисляют константу для коррекции адресов строк видеопамяти по способу, описанному в примере 7.13.

Для чтения образа файла в регистр `cx` помещается его размер, очищается переменная `SwpOffs` и происходит обращение к подпрограмме `readf`. Если чтение прошло без ошибок, то команда `jnc ok` обойдет строку, состоящую только из комментария. Что делать при ошибках чтения, решать вам.

В подпрограммах построения строк (`drawline`) регистр `fs` используется при чтении кодов точек из оперативной памяти, а регистр `gs` — при работе с таблицей цветов. Поэтому перед началом основного цикла в них записываются коды соответствующих сегментов.

Цикл построения рисунка имеет метку `invout`. Работа с адресами в нем организована так, что при каждом повторе регистр `di` содержит адрес очередной строки видеопамяти, а регистр `si` — адрес последней обработанной строки образа рисунка (при первом входе это размер рисунка в байтах).

Перед вызовом подпрограммы `drawline` содержимое `si` уменьшается на размер строки в файле (`fwidth`) и указывает начало очередной строки. При выполнении подпрограммы `drawline` регистр `si` увеличится на `iwidth`. Поэтому после возврата из `drawline` значение `si` увеличивается на `rmndr` (количество лишних байтов) и уменьшается на `fwidth`. В результате регистр `si` будет содержать адрес начала последней обработанной строки.

Адрес начала следующей строки видеопамяти, как обычно, увеличивается на константу переадресации, которая хранится в регистре `dx`. Если при сложении (`add di, dx`) происходит переполнение, то устанавливается следующее окно видеопамяти.

Команда `loop invout` повторяет выполнение цикла нужное количество раз, после чего восстанавливаются сохраненные в стеке величины, исходное окно видеопамяти и происходит возврат на вызывающий модуль.

По сравнению с обычным циклом построения рисунка (см. пример 7.23) в данном случае добавились три команды, корректирующие значение регистра `si`. При желании их количество можно сократить до двух. Подумайте, как это сделать, но не забывайте, что `fwidth – rmndr` не обязательно равно `iwidth`. Вспомните, как вычислялись эти величины.

Замечание

Напомним, что имя подпрограммы построения строки не обязательно указывать в явном виде. Команду `call drawline` можно изменить на `call bx`, а перед обращением формировать в регистре `bx` адрес подпрограммы построения строки (см. раздел А.3.3).

А.3.2. Построение рисунка снизу вверх

При работе с `BMP`-файлами произвольного размера основным способом является построение изображения снизу вверх. В таком случае строки образа рисунка считываются в порядке их расположения в файле, а выводятся на экран начиная с последней строки рисунка.

Такой способ уже использовался для построения полноцветных рисунков формата `BMP`, он описан в разделе 7.5.1, а текст соответствующей подпрограммы показан в примере 7.25. Отличие рассматриваемого здесь случая только в том, образ рисунка использует палитру, а коды его точек могут занимать 1, 4 или 8 разрядов.

Описание предварительных действий. Перед началом построения рисунка подпрограмма вычисляет адрес начала последней строки в видеопамяти, размер считываемой из файла порции данных в байтах, количество содержащихся в ней строк образа рисунка и значение константы для переадресации строк видеопамяти. Опишем, как это делается.

Адрес начала последней строки рисунка в видеопамяти вычисляется следующим способом. Расстояние между первой и последней строкой равно `(iheight – 1)*bperline` байтов. В общем случае произведение занимает два слова. Содержимое младшего слова прибавляется к адресу начала первой строки. Содержимое старшего слова преобразуется в номер окна, который прибавляется к номеру окна, в котором находится первая строка. Вычисленное значение окна надо установить.

Количество строк в порции считываемых данных (`part`) вычисляется как частное от деления числа 65 535 на размер строки в файле (`fwidth`). Умножив `part` на `fwidth`, получим размер порции для чтения в байтах.

Замечание

Вместо умножения можно вычесть остаток от деления из числа 65 535.

Константа переадресации строк видеопамяти. После записи кодов точек текущей строки в видеопамять определяется адрес начала предыдущей. Для этого текущий адрес видеопамяти уменьшается на величину, вычисляемую по формуле `(iwidth + horsize)*bytppnt`. Если при вычитании вырабатывается признак переноса, то устанавливается предыдущее окно видеопамяти, в противном случае текущее окно не изменяется.

Подпрограмма BigBmp. Текст подпрограммы, выполняющей построение рисунка описанным способом, приведен в примере А.10. Перед ее вызовом адрес левого верхнего угла рисунка помещается в регистр `di` и устанавливается окно видеопамяти, которому принадлежит этот адрес. Регистр `es` должен содержать код видеосегмента. Если в тексте вместо `call drawline` записана команда `call bx`, то в регистре `bx` указывается адрес подпрограммы построения строки (см. раздел А.3.3).

Пример А.10. Построение рисунка формата BMP снизу вверх

```
BigBmp:  pusha                ; сохранение "всех" регистров
         PushReg <gs,fs,Cur_win>; сохранение gs, fs и Cur_win
         mov   ax, iheight    ; ax = iheight, количество строк в рисунке
         mov   remline, ax    ; remline = ax, количество строк в рисунке
         dec   ax             ; учет нумерации строк с нуля
         mul   bperline       ; dx:ax = (iheight — 1)*bperline
         add   di, ax         ; di = адрес последней строки рисунка
         adc   dx, 00         ; учитываем возможный перенос
         mov   ax, GrUnit     ; ax = GrUnit (единица измерения окон)
         mul   dl             ; вычисляем добавку к номеру окна
         add   Cur_win, ax    ; номер окна для последней строки
         call  Setwin         ; установка вычисленного окна
         mov   ax, -1         ; ax = 65535
         xor   dx, dx         ; очистка старшей части делимого
         div   fwidth         ; ax = 65535 / fwidth (частное от деления)
         mov   part, ax       ; part = число строк в порции для чтения
         mul   fwidth         ; ax = ax*fwidth
         mov   numbyte, ax    ; размер порции считываемой в байтах
         mov   ax, iwidth     ; ax = iwidth
         add   ax, horsize    ; ax = iwidth + horsize
         mul   bytppnt        ; ax = (iwidth + horsize)*bytppnt, dx = 0
         mov   dx, ax         ; сохраняем для коррекции адресов
         mov   fs, SwpSeg     ; fs = сегмент буфера обмена
         mov   SwpOffs, 0     ; очистка смещения в сегменте
         mov   gs, GenSeg     ; !! gs = сегмент таблицы цветов
NewPart: mov   cx, numbyte    ; cx = количество считываемых байтов
         call  Readf          ; чтение порции в буфер обмена
         jnc   sr             ; -> чтение прошло без ошибок
;        Здесь должны выполняться действия при ошибке чтения
sr:      mov   cx, part       ; cx = кол-во строк в полной порции
         cmp   remline, cx    ; считана полная порция данных ?
         jae   @F             ; -> да, обходим следующую команду
         mov   cx, remline    ; нет, cx = оставшееся число строк
@@:      sub   remline, cx    ; уменьшаем значение счетчика строк
         xor   si, si         ; si = начало буфера обмена
```

```
drwout:  push  cx              ; сохраняем значение счетчика строк
         mov   cx, iwidth      ; cx = размер строки (в точках)
         call  drawline        ; !! или call bx — построение строки
         pop   cx              ; восстанавливаем счетчик строк
         add   si, rmndr       ; корректируем адрес в буфере обмена
         sub   di, dx          ; di = адрес начала предыдущей строки
         jnc   @F              ; -> адрес в пределах текущего окна
         call  PrevWin         ; установка предыдущего окна
@@:      loop  drwout          ; управление циклом построения строк
         cmp   remline, 0      ; остались не обработанные строки ?
         jne   NewPart         ; -> да, на чтение следующей порции
         PopReg <Cur_win,fs,gs>; восстановление Cur_win, fs и gs
         popa                  ; восстановление "всех" регистров
         call  Setwin          ; восстановление исходного окна
         ret                   ; возврат из подпрограммы
```

Выполнение подпрограммы примера А.10 начинается с подготовительных действий, смысл и назначения которых описаны выше. Основной цикл имеет метку `NewPart`. Если вы сравните его с одноименным циклом примера 7.25, то убедитесь в их полном совпадении, поэтому мы не будем повторять описание выполняемых действий.

Таким образом, основной цикл построения `BMP`-файлов произвольного размера *не зависит* от того, как подготовлен образ исходного рисунка, — с использованием или без использования палитры цветов. От этого зависят только подпрограммы `drawline`, вызываемые для построения строк рисунков.

А.3.3. Универсальная процедура построения рисунка

В данном разделе описана универсальная процедура построения рисунка формата `BMP`, использующего палитру цветов. Двухцветные и 16-цветные рисунки хранятся в упакованном виде, поэтому процедура выбирает нужную подпрограмму для их распаковки. Будем так же считать, что в зависимости от размера файла выбираются подпрограммы из примеров А.9 или А.10.

Текст процедуры приведен в примере А.11, а использованные в ней подпрограммы описаны ниже.

Пример А.11. Начало построения рисунка формата BMP

```
BmpShow: lea   bx, mode_8     ; bx = адрес подпрограммы mode_8
         cmp   bitcnt, 08     ; bitcnt = 8 ?
         je    @F             ; -> да, переход на локальную метку
```

```
        lea   bx, mode_4     ; bx = адрес подпрограммы mode_4
        cmp   bitcnt, 04     ; bitcnt = 4 ?
        je    @F             ; -> да, переход на локальную метку
        lea   bx, mode_2     ; bx = адрес подпрограммы mode_2
@@:     mov   ax, fwidth     ; ax = fwidth, размер строки в файле
        mul   iheight        ; dx:ax = fwidth*iheight
        or    dx, dx         ; образ рисунка помещается в сегменте ?
        jne   @F             ; -> нет, файл большого размера
        mov   si, ax         ; si = размер образа рисунка
        jmp   Smlbmp         ; переход на Smlbmp
@@:     jmp   BigBmp         ; переход на BigBmp
```

Для того чтобы выполняемые в примере А.11 действия имели смысл, в текстах примеров А.9 и А.10 надо заменить `call drawline` на `call bx`, как указано в комментарии. Первые 7 команд примера А.11 формируют в `bx` адрес подпрограммы обработки строки (примеры А.12—А.14), в зависимости от размера кода точки образа рисунка. После этого вычисляется размер образа рисунка, и если он помещается в одном сегменте (`dx=0`), то выбирается подпрограмма `Smlbmp`, в противном случае `BigBmp`.

Таким образом, для построения рисунка формата `BMP` задача должна обращаться к процедуре `BmpShow`, как к подпрограмме. Перед ее вызовом в регистре `di` указывается адрес левого верхнего угла рисунка в видеопамяти и устанавливается окно, которому принадлежит этот адрес. Регистр `es` должен содержать код видеосегмента (хранящийся в `Vbuff`).

Подпрограммы обработки строк. Для того чтобы тексты подпрограмм построения строк не зависели от установленного задачей видеорежима, преобразование кодов точек и их запись в видеопамять вынесены в подпрограммы `outpnt`, которые будут описаны ниже.

В примере А.12 приведена подпрограмма, выполняющая построение строки рисунка, у которого код точки занимает 1 байт. Ее отличие от описанных ранее вариантов только в том, что для записи кода точки вызывается вспомогательная подпрограмма `outpnt`.

Пример А.12. Вывод строки формата 8 бит на точку (256 цветов)

```
mode_8: lods  byte ptr fs:[si]  ; al = код очередной точки
        call  outpnt            ; обращение к подпрограмме записи
        loop  mode_8            ; управление повторами цикла
        ret                     ; возврат из подпрограммы
```

В примере А.13 приведен текст подпрограммы, выполняющей распаковку точек 16-цветного рисунка в процессе построения строки.

Пример А.13. Вывод строки формата 4 бита на точку (16 цветов)

```
mode_4: lods  byte ptr fs:[si]  ; al = код очередных 2-х точек
        push  ax                ; сохраняем содержимое ax
        shr   al, 04            ; выделяем код старшей тетрады
        call  outpnt            ; обращение к подпрограмме записи
        pop   ax                ; восстанавливаем содержимое ax
        dec   cx                ; cx = cx — 1, счетчик точек в строке
        je    @F                ; -> все точки выведены
        and   al, 0Fh           ; выделяем код младшей тетрады
        call  outpnt            ; обращение к подпрограмме записи
        loop  mode_4            ; управление повторами цикла
@@:     ret                     ; возврат из подпрограммы
```

Сравните текст примера А.13 с текстом подпрограммы `drwlin4` (см. пример 3.17). В данном случае он упростился за счет использования вспомогательной подпрограммы `outpnt`.

Замечание

Напомним, что дополнительная коррекция и проверка значения счетчика повторов цикла (`cx`) нужна потому, что при нечетном количестве точек в рисунке последний байт содержит код только одной точки (младшую тетраду выводить на экран нельзя).

В примере А.14 приведен текст подпрограммы, выполняющей распаковку 2-цветных рисунков в процессе построения строки.

Пример А.14. Вывод строки формата 1 бита на точку (2 цвета)

```
mode_2: lods  byte ptr fs:[si]  ; al = код очередных восьми точек
        mov   ah, 80h           ; ah = константа для выделения разряда
md_21:  push  ax                ; сохраняем содержимое ax
        and   al, ah            ; выделяем текущий разряд
        je    md_22             ; если нуль, то обходим одну команду
        mov   al, 01            ; иначе записываем в al единицу
md_22:  call  outpnt            ; обращение к подпрограмме записи
        pop   ax                ; восстанавливаем содержимое ax
        dec   cx                ; cx = cx — 1, счетчик точек в строке
        je    md_23             ; -> все точки выведены
        shr   ah, 01            ; изменяем константу выделения
        jne   md_21             ; -> если обработаны не все точки
        jmp   short mode_2      ; -> если обработано 8 точек
md_23:  ret                     ; возврат из подпрограммы
```

Сравните текст примера А.14 с текстом подпрограммы `drwlin1` (см. пример 3.18).

Замечание

Напомним, что дополнительная коррекция и проверка значения счетчика повторов цикла (`cx`) нужна потому, что в зависимости от количества точек в строке последний байт может быть заполнен частично.

Подпрограммы записи кодов точек существенно зависят от видеорежима. Если задача установила один из видеорежимов `PPG`, то в самом простом случае выполняются действия, показанные в примере А.15.

Пример А.15. Простой вывод точки в режимах PPG

```
outpnt:  stosb          ; запись кода точки в видеопамять
         or    di, di   ; адрес в пределах сегмента?
         jne   @F       ; -> да
         call  NxtWin   ; нет, установка следующего окна
@@:      ret            ; возврат из подпрограммы
```

Пример А.15 рассчитан на те случаи, когда при записи кодов точек в видеопамять не требуются никакие дополнительные действия. К ним относится перекодировка точек, которая может потребоваться, по крайней мере, в двух случаях.

При работе в режимах `PPG` палитра может быть установлена с изменением исходного расположения базовых цветов. В разделе 4.5 мы подробно обсуждали, в каких случаях и почему используется такая установка палитры. При любом изменении расположения цветов, описанных в палитре, требуется изменение кодов точек при их записи в видеопамять. Один из вариантов перекодировки был показан в примере 4.8.

При работе в режимах `direct color` перекодировка производится в тех случаях, когда используется таблица цветов. Вариант подготовки такой таблицы показан в примере А.8. Размер строки таблицы, а следовательно, и действия, выполняемые при перекодировке, зависят от установленного видеорежима. При режимах `Hi-Color` строка таблицы занимает 2 байта, а при режимах `True Color` — 4 байта. В примере А.16 приведена подпрограмма, выполняющая перед записью в видеопамять перекодировку точек по таблице цветов.

Пример А.16. Перекодировка точек в режимах direct color

```
outpnt:  push  eax           ; сохранение содержимого eax
         and   eax, 0FFh     ; очистка старших разрядов eax
         shl   ax, wrdppnt   ; учет размера строки таблицы
```

```
add   ax, GenOffs    ; ax = смещение начала таблицы
mov   ax, gs:[eax]   ; !! или mov eax, gs:[eax] для True Color
stosw                ; !! или stosd для режимов True Color
pop   eax            ; восстановление содержимого eax
ret                  ; возврат из подпрограммы
```

Текст примера А.16 является простым повторением текста примера 7.22. Дополнительные подробности вы найдете в разделе 7.4.3.

Заключение. Мы завершили описание работы с файлами формата BMP. В данном приложении отсутствует описание построения полноцветных рисунков. Это объясняется тем, что оно приведено в разделе 7.5.1 основной части книги вместе с текстом соответствующей подпрограммы (см. пример 7.25).

ПРИЛОЖЕНИЕ Б

Оперативная память

Оперативная память (ОЗУ, RAM) является одним из важнейших ресурсов персонального компьютера. В англоязычной технической литературе вы можете встретить три термина, характеризующие тип памяти, а именно: `conventional memory`, `extended memory` и `expanded memory`. У современных ПК они относятся к разным частям одного физического устройства и являются характеристиками способа доступа к этим частям. Различие способов доступа к отдельным частям памяти является специфической особенностью (родимым пятном) и одним из существенных недостатков семейства IBM PC. В чем именно оно заключается, описано в данном приложении.

Предельно допустимый объем памяти зависит от системной (материнской) платы ПК, точнее от набора микросхем (`chip set`), на базе которого она собрана. Реально существующий объем выводится на экран монитора в процессе загрузки ПК, когда `BIOS` проверяет (тестирует) память. В процессе работы ПК можно с помощью специальных задач узнать объем и текущее распределение пространства ОЗУ. Например, в состав DOS входит задача `mem.exe`, а в состав Norton Commander — `sysinfo.exe`. Прикладные задачи, нуждающиеся в больших объемах памяти, должны самостоятельно определять размер ее доступного пространства.

Б.1. Обычная память (Conventional Memory)

`Conventional` — общепринятый, обычный. Так называют младшую часть ОЗУ, занимающую первые 640 Кбайт пространства адресов и имеющую большое значение в обеспечении работоспособности ПК. В ней хранятся векторы прерываний, данные, используемые `BIOS` и DOS, рабочая область и резидентная часть DOS, драйверы внешних устройств, а также другие программы и данные, необходимые для поддержки нормальной работы ПК. Наконец, DOS загружает прикладные задачи только в обычную память. Если

при отсутствии или неисправности дополнительной памяти ПК может выполнять свои функции, то при неисправности младшей части памяти его работа просто невозможна.

Б.1.1. Сегменты оперативной памяти

Первые компьютеры IBM PC имели 16-разрядную шину данных, что соответствовало пространству адресов 64 Кбайт. У IBM PC/XT, собранного на базе процессора Intel 8086, адресная шина увеличилась на 4 разряда и появилась возможность работы с пространством адресов размером в 1 Мбайт. Однако основные регистры процессора остались 16-разрядными, поэтому 20-разрядные адреса в них просто не помещались. Для работы с пространством адресов в 1 Мбайт потребовалось создание специального механизма.

Разработчики выбрали не самое лучшее из возможных решений, ограничив предельный размер пространства адресов значением 1 Мбайт. Возможно, им казалось, что такой объем памяти удовлетворит нужды пользователей на многие годы. Так или иначе, но заведомо ограниченное решение стало стандартом и используется во всех без исключения компьютерах семейства IBM PC.

Режимы работы микропроцессора. Компьютеры IBM PC/AT, собранные на базе процессора Intel 80286, уже обеспечивали доступ к пространству адресов размером в 16 Мбайт. Однако для реального использования такого пространства требовалось новое программное обеспечение. К этому времени было создано много системных и прикладных программ, ориентированных на возможности процессора Intel 8086, и изменять способ работы с адресами было слишком поздно. Разработчики пошли по другому пути. Начиная с модели 80286, все микропроцессоры Intel поддерживают два режима работы. Один из них называется *реальным режимом* (`real-address mode`), а другой *защищенным режимом* (`protected-address mode`).

Реальным режим назван потому, что используются физические адреса оперативной памяти и внешних устройств, значения которых не превышают 1 Мбайт. При включении ПК любой современный микропроцессор переключается в реальный режим. В нем он работает в точности как Intel 8086, только значительно быстрее.

Работа с дополнительным пространством адресов возможна только в защищенном режиме. При этом микропроцессор контролирует допустимость значений адресов, откуда и произошло название режима. Для распределения и контроля расширенного пространства адресов в состав микропроцессоров включены специальные регистры и команды для их обслуживания. Те и другие доступны только при работе микропроцессора в защищенном режиме.

У микропроцессора Intel 386 шины адреса и данных стали 32-разрядными, а размер пространства адресов увеличился до 4 Гбайт. Кроме того, был введен

еще один виртуальный режим (`virtual-address mode`), иногда его обозначают `V86`. Он позволяет при работе в защищенном режиме эмулировать процессор 8086. Отличие от реального режима в том, что адреса не являются физическими (реальными), а отображаются на конкретную часть 32-разрядного пространства. Это позволяет современным операционным системам "параллельно" выполнять несколько задач, рассчитанных на работу в реальном режиме, загружая их в свободные адреса ОЗУ.

Таким образом, работа любого современного IBM PC начинается в реальном режиме, затем его можно перевести в защищенный режим, а из последнего — в виртуальный. Изменение текущих режимов работы микропроцессора обычно выполняет операционная система. Прикладные задачи могут это делать только при работе в монопольном режиме, на практике такие случаи встречаются редко.

Сегменты оперативной памяти. В реальном режиме работы процессора пространство оперативной памяти делится на сегменты, размер которых не превышает 64 Кбайт, а адрес начала обязательно кратен 16.

При выборке команд и операндов микропроцессор вычисляет абсолютный адрес, исходя из значения (кода) сегмента и смещения (относительного адреса) в нем. Для этого он сдвигает значение сегмента на 4 разряда вправо и прибавляет смещение к результату сдвига. Табл. Б.1 иллюстрирует схему формирования 20-разрядного адреса.

Таблица Б.1. *Схема формирования 20-разрядного адреса*

				15	14	13	...	**с**	**м**	**е**	**щ**	**е**	**н**	**и**	**е**	...	2	1	0
15	14	13	12	...	**с**	**е**	**г**	**м**	**е**	**н**	**т**	...	2	1	0				
19	18	17	...	**п**	**о**	**л**	**н**	**ы**	**й**		**а**	**д**	**р**	**е**	**с**	...	2	1	0

При записи на бумаге или на экране монитора сегмент и смещение разделяет символ "двоеточие". Например, записи `1111:2222` соответствует адрес 11110 + 2222 = 13332. Такая форма записи адресов используется при работе с большинством отладчиков.

Один и тот же полный адрес может быть задан различными способами. Например, область данных `BIOS` расположена в оперативной памяти, начиная с абсолютного адреса `400h`. Его можно записать в виде `0000:0400`, `0040:0000`, `0020:0200` и т. п.

Оперативная память занимает не все пространство адресов, а только младшие 640 Кбайт. Ее можно разделить на 10 сегментов максимального размера (64 Кбайт), коды которых будут изменяться от `0000` до `9000h`. Обычно далеко не все сегменты имеют максимальный размер, поэтому их количество быва-

ет больше 10, но в любом случае сегмент с кодом A000h уже не относится к оперативной памяти, обычно это код видеосегмента при работе в графических режимах. И вообще, пространство от 640 Кбайт до 1 Мбайт выделено для размещения BIOS, ее дополнений, расположенных на платах контроллеров внешних устройств, адресов, через которые происходит доступ к этим устройствам, и т. п.

Расположение адресов в регистрах. Микропроцессор выбирает части адреса из двух разных регистров. Коды сегментов хранятся в специальных сегментных регистрах, которые предназначены только для этих целей. Начиная с модели 80386, у микропроцессоров Intel таких регистров 6. На языке ассемблера они имеют имена CS, DS, ES, FS, GS, SS. По умолчанию, т. е., если явно не указано другое, регистр CS используется при выборке команд, DS — при выборке данных, а SS — при работе со стеком. Умолчаний для ES, FS и GS не существует, они всегда указываются явно.

Смещения в сегментах (относительные адреса) процессор может выбирать из шести регистров, которые на языке ассемблера имеют имена BX, BP, IP, SP, DI, SI. Первый (BX) называется регистром базы, относится к регистрам общего назначения и делится на два байта. Остальные пять на байты не делятся, три из них имеют фиксированное назначение. В SP хранится указатель стека, с ним работают команды push, pop и др. Регистр BP используется для доступа обычных команд (mov, add и пр.) к области стека (см. приложение В). Из IP процессор выбирает относительный адрес очередной выполняемой команды. Индексные регистры DI и SI используют строковые операции, а в остальных случаях они не имеют фиксированного назначения.

Б.1.2. Сегментирование текстов программ

Составленные на языке Макроассемблера программы обязательно сегментируются. В простейшем случае вся программа может состоять только из одного сегмента. Подобная программа была приведена в примере 4.1.

Описание и расположение сегментов. В большинстве случаев текст программы состоит, по крайней мере, из трех сегментов, содержащих область стека, раздел данных и коды. В примере Б.1 показана общая структура программы, содержащей три основных сегмента.

Пример Б.1. Структура программы, состоящей из трех сегментов

```
stack  Segment word stack "stack"; начало стекового сегмента
       db 100h dup (?) ; размер области стека 100h байтов
stack  Ends            ; конец стекового сегмента
data   Segment         ; начало сегмента данных
```

```
;       В этом сегменте располагается описание
;       используемых в программе данных
data    Ends            ; конец сегмента данных
code    Segment         ; начало кодового сегмента
;       В этом сегменте располагается текст основной
;       программы и входящих в нее подпрограмм
code    Ends            ; конец кодового сегмента
        END             ; конец текста программы
```

Из текста примера Б.1 видно, что описание сегмента открывает директива `Segment`, а закрывает `Ends`. Перед обеими директивами указывается одинаковая метка, символ "двоеточие" в данном случае отсутствует. Метка указывается дважды для того, чтобы при работе с вложенными сегментами Макроассемблер мог определить, какой из них закрывает данная директива. Формально вложенные сегменты допустимы, но особых преимуществ их использование не дает.

После директивы `Segment` могут располагаться параметры, которые обычно не нужны. Исключением является стековый сегмент, если не указать параметр `stack`, то компоновщик (`link.exe`) выдаст сообщение об его отсутствии в теле задачи. Это не аварийное, а предупреждающее сообщение, т. к. стековый сегмент действительно может отсутствовать. Однако если он явно описан, а компоновщик его не обнаружил, значит, в тексте программы допущена ошибка, и ее надо исправить.

Последовательность расположения сегментов в тексте программы не имеет значения ни для Макроассемблера, ни для компоновщика, ни для DOS. Поэтому разработчик располагает их в нужном ему порядке, а основным критерием является наглядность текста программы и удобство его анализа. Правда в некоторых случаях бывает важно знать, какой сегмент расположен последним в тексте построенной задачи. Такой случай будет рассмотрен в следующем разделе.

Расположение сегментов в задаче. Макроассемблер может располагать сегменты в тексте будущей задачи либо в порядке их описания в исходной программе, либо по именам в алфавитном порядке. Вариант расположения можно выбрать с помощью директив `.SEQ` и `.ALPHA` (в обоих случаях указание точки обязательно), которые располагаются перед описанием первого сегмента. Директива `.SEQ` устанавливает расположение сегментов в порядке их описания, а директива `.ALPHA` — в алфавитном порядке. Обычно по умолчанию сегменты располагаются в порядке их описания.

Реальное расположение сегментов в построенной задаче компоновщик выводит в файл, имя которого надо указать в ответ на подсказку "`List file >`", по умолчанию такой файл имеет тип `map`. Если в ответ на подсказку вы введете конкретное имя, то получите файл, содержащий имена, адреса и размеры описанных в программе сегментов.

Специальные директивы описания сегментов. В документации на Макроассемблер для описания сегментов, содержащих константы, данные, коды и стек, рекомендуется использовать специальные директивы: `.CONST`, `.DATA`, `.CODE`, `.STACK` (указание точки обязательно), которые автоматически формируют все параметры сегмента. Им обязательно должна предшествовать директива `.MODEL` с указанием названия используемой модели памяти (пример Б.2). Допустимо использование следующих имен: `small`, `compact`, `medium`, `large` и `huge`.

Перечисленные типы моделей памяти поддерживают компиляторы таких языков программирования, как Паскаль, Си, Фортран и некоторые другие, что и объясняет введение директивы `.MODEL`. В MASM 6.0 специально для защищенного режима введена еще одна модель — `flat`.

Пример Б.2. Специальные директивы описания основных сегментов

```
        Dosseg        ; задает расположение сегментов
        .Model small  ; описание модели памяти
        .Stack 100h   ; описание стекового сегмента
        .Data         ; начало сегмента данных
;       В этом сегменте располагается описание
;       используемых в программе данных
        .Code         ; начало сегмента кодов
;       В этом сегменте располагается текст основной
;       программы и входящих в нее подпрограмм
        END           ; конец текста программы
```

В примере Б.2 директивы `.Stack`, `.Data`, `.Code` указывают начало соответствующего сегмента, директива `Ends` при этом не нужна. Признаком конца текущего сегмента является начало следующего или директива `END`, завершающая текст программы, поэтому вложение сегментов исключено. В тексте программы, кроме указанных сегментов, могут использоваться другие, описанные обычным способом.

Перед специальными директивами имена сегментов не указываются, но они могут понадобиться для использования в программе. Макроассемблер присваивает стековому сегменту имя `stack`, сегменту данных — имя `_data` (нижняя черта обязательна). Имя сегмента кодов зависит от модели памяти, например, при моделях `small` и `compact` оно будет `_text`. Присвоенные сегментам имена можно посмотреть в карте памяти, которую формирует компоновщик при указании файла с типом `map` (см. выше).

Для управления последовательностью расположения сегментов в примере Б.2 использована директива `Dosseg`, которая размещает сегменты в соответствии с соглашениями DOS. Это значит, что первым в теле задачи будет расположен кодовый, а последним — стековый сегмент. Если сегментов

всего три, то данные будут размещены после кодов, перед областью стека. Дополнительные сегменты располагаются между кодами и данными.

Работа с именами сегментов. Макроассемблер рассматривает имя сегмента как константу (а не как переменную), ее прямая пересылка в сегментный регистр невозможна, поэтому приходится использовать регистр-посредник. Например, в большинстве случаев в начале выполнения задачи надо определить содержимое сегментного регистра `DS`, поместив в него код сегмента данных. Это можно сделать с помощью двух следующих команд:

```
start:  mov   ax, data    ; ax = код сегмента данных
        mov   ds, ax      ; ds = ax
```

Этот пример составлен исходя из предположения, что сегмент данных описан так, как приведено в примере Б.1. Если же он описан, как в примере Б.2, то в первой команде надо изменить имя `data` на `_data` или `@data`.

Макроассемблер позволяет определить код сегмента тремя разными способами: указать его явно в виде числа, указать имя сегмента или указать имя метки, расположенной в нужном сегменте. В последнем случае перед именем метки помещается оператор `seg`. Следующие три команды иллюстрируют сказанное:

```
mov       ax, 0400h       ; указание конкретного числового значения
mov       ax, job         ; явное указание имени сегмента
mov       ax, seg buf     ; косвенная ссылка на сегмент по имени метки
```

Команды первого типа нужны при обращении к сегментам, расположенным вне тела программы, например к области данных `BIOS`.

Команды второго типа применяются при работе с сегментами, описанными в тексте программы. Точное значение таких сегментов заранее неизвестно, его вычисляет DOS при загрузке программы в память для выполнения.

Команды третьего типа полезны в тех случаях, когда метка является внешним именем, не описанным в программе, и нужно узнать, какому сегменту она принадлежит.

После того, как с помощью одной из этих команд значение сегмента окажется в регистре `ax`, оно копируется в нужный сегментный регистр командой пересылки, как это было показано выше на примере определения содержимого регистра `ds`.

Резервирование пространства памяти. Большинство графических задач нуждается в определенном пространстве ОЗУ для размещения буферов различного назначения. В простейшем случае для резервирования требуемого пространства памяти в текст программы включаются дополнительные сегменты.

Это можно сделать, например, так:

```
Job     Segment             ; начало сегмента
buffer  db 65536 dup (?)    ; директива резервирует 65536 байтов
Job     Ends                ; конец сегмента
```

В приведенном примере сегмент становится частью тела задачи и увеличивает его размер на соответствующее количество байтов. Поэтому такой способ распределения пространства ОЗУ является вспомогательным и применять его можно только в порядке исключения.

Более гибкий и универсальный вариант заключается в том, что для размещения буферов различного назначения пространство оперативной памяти не резервируется в исходном тексте программы, а распределяется в процессе выполнения задачи.

Б.1.3. Динамическое управление памятью

Перед началом выполнения задачи DOS выделяет для нее всю свободную часть пространства обычной памяти. Задача может произвольно распоряжаться выделенным пространством ОЗУ, но она не должна выходить за его пределы, т. к. это приведет к непредсказуемым результатам. В данном разделе будет описан один из способов работы задачи с обычной памятью.

Блок задачи. Пространство, выделенное DOS для выполнения задачи, в технической документации принято называть блоком задачи. Он состоит из трех основных частей. Первые `100h` (256) байтов блока занимает специальная структура данных — префикс программного сегмента (`PSP`). В нем хранятся величины, которые могут быть нужны при выполнении задачи. Сразу после `PSP` в памяти расположены сегменты, описанные в исходном тексте программы.

Как уже говорилось, порядок расположения сегментов в блоке задачи зависит от их имен и от наличия директив `.Alpha`, `.Seq` или `Dosseg`, которые могут указываться в начале исходного текста программы. После последнего сегмента в блоке задачи находится свободное пространство, которым задача может распоряжаться по своему усмотрению (по усмотрению программиста). Но для доступа к этому пространству надо знать его размер и адрес начала, точнее, сегмент, с которого оно начинается. При составлении программы эти величины неизвестны, поскольку их формирует DOS, исходя из реально имеющихся ресурсов на момент загрузки задачи для выполнения.

Адрес свободного пространства. Для определения адреса свободного пространства надо знать, где заканчивается последний сегмент задачи. Для этого, в свою очередь, необходимо выяснить, какой из сегментов, описанных в исходном тексте программы, окажется последним в теле задачи.

Иногда рекомендуют включать в исходный текст задачи пустой сегмент, имя которого начинается на букву `Z`, например:

```
Zero   Segment    ; начало сегмента Zero
Zero   Ends       ; конец сегмента Zero
```

Если при этом задана директива `.Alpha`, т. е. надежда, что сегмент `Zero` будет расположен в теле задачи последним. В таком случае он и является началом свободного пространства в блоке задачи.

Несмотря на очевидную простоту, этот способ не универсален, поскольку сегмент `Zero` не всегда оказывается последним. Напомним, что имена могут состоять не только из букв. Например, если кодовый сегмент имеет имя `_text`, то сегмент `Zero` будет расположен перед ним. Поэтому нужен более надежный способ определения последнего сегмента.

Если указана директива `Dosseg`, то последним в теле задачи будет расположен стековый сегмент. Тот же результат получается при использовании специальных директив, показанных в примере Б.2. Учитывая, что они рекомендованы разработчиками в качестве основных, имеет смысл исходить из допущения, что последним в теле задачи расположен стековый сегмент.

Стековый сегмент имеет определенный размер, который надо учесть при вычислении первого свободного сегмента в блоке задачи. При входе в задачу регистр `SP` содержит адрес верхушки, который равен размеру стека, выраженному в байтах. Его надо преобразовать в параграфы (разделить на 16) и сложить с кодом стекового сегмента, хранящимся в регистре `SS`. Таким образом, адрес начала свободного пространства (код свободного сегмента) вычисляется по формуле:

```
freeseg = [ss] + ([sp] /16) + 1
```

В этой формуле квадратные скобки указывают на то, что используется содержимое регистров `ss` и `sp`. При программировании деление `[sp]` на 16 заменяется сдвигом на 4 разряда вправо. Для того чтобы не потерять один параграф, размер стека должен быть кратен 16-ти, в противном случае его надо округлить в сторону увеличения. Прибавление 1 нужно потому, что свободный сегмент должен начинаться после стекового, не перекрывая его.

Замечание

Какой бы сегмент вы не выбрали в качестве точки отсчета, советуем при построении задачи обязательно задать файл листинга (карту памяти) и убедиться в том, что выбранный сегмент расположен в теле задачи последним.

Размер свободного пространства. В `PSP` слово со смещением 2 содержит последний доступный для задачи адрес оперативной памяти, выраженный в параграфах, т. е. это код последнего доступного сегмента. После загрузки задачи DOS помещает код сегмента, содержащего `PSP` в регистры `es` и `ds`.

Содержимое регистра ds изменяется первыми командами задачи, а es можно использовать для чтения указанной величины. Если ее уменьшить на freeseg, то получится размер свободного пространства, выраженный в параграфах. Теперь надо проверить, достаточно ли выделенное пространство для нужд задачи, и если да, то его можно использовать.

Вычисление SwpSeg и GenSeg. В приведенных в основной части книги примерах использовались буфер обмена и буфер общего назначения. Мы предполагали, что код сегмента, содержащего буфер обмена, хранится в переменной SwpSeg, а буфер общего назначения — в переменной GenSeg. Покажем, как можно сформировать значения этих переменных после вычисления размера и адреса начала свободного пространства описанным выше способом.

В примере Б.3 приведен фрагмент начала программы, в котором выполняются все необходимые вычисления. Для описания сегментов в нем использованы обычные директивы (см. пример Б.1).

Пример Б.3. Вычисление значений переменных SwpSeg и GenSeg

```
        .Alpha              ; порядок расположения сегментов
        Dosseg              ; порядок расположения сегментов
stack   Segment word stack "stack" ; начало стекового сегмента
        db 200h dup (?)     ; размер области стека 200h байтов
stack   Ends                ; конец стекового сегмента
data    Segment             ; начало сегмента данных
prmpt   db 0Dh,0Ah,'! Для выполнения задачи не хватает памяти !$'
freeseg dw 0                ; первый свободный сегмент
msize   dw 0                ; размер памяти в параграфах
needm   dw 2000h            ; требуемый размер памяти
SwpOffs dw 0                ; смещение в буфере обмена
SwpSeg  dw 0                ; сегмент буфера обмена
GenOffs dw 0                ; смещение в буфере
GenSeg  dw 0                ; сегмент буфера общего назначения
;       Далее описываются другие используемые данные
data    Ends                ; конец сегмента данных
code    Segment             ; начало кодового сегмента
        .386                ; набор команд процессора
start:  mov   ax, data      ; ax = код сегмента данных
        mov   ds, ax        ; ds = код сегмента данных
        mov   bx, sp        ; bx = размер стека в байтах
        shr   bx, 04        ; превращаем его в параграфы
        mov   ax, ss        ; ax = код стекового сегмента
        add   bx, ax        ; bx = последний параграф стека
        inc   bx            ; bx = первый свободный сегмент
        mov   freeseg, bx   ; freeseg = bx
```

```
        mov   ax, es:[02] ; ax = последний доступный сегмент
        sub   ax, bx      ; ax = ax – bx
        mov   msize, ax   ; размер памяти в параграфах
        cmp   ax, needm   ; памяти достаточно ?
        jae   @F          ; -> да
        lea   dx, prmpt   ; dx = адрес аварийного сообщения
        mov   ah, 09      ; ah = код функции DOS
        int   21h         ; вывод текста сообщения
        mov   ax, 4C00h   ; ah = 4C, код функции DOS
        int   21h         ; завершение выполнения задачи
@@:     mov   SwpSeg, bx  ; SwpSeg = bx
        add   bx, 1000h   ; bx = bx + 65536/16
        mov   GenSeg, bx  ; GenSeg = bx
;       Далее расположен текст основной программы и подпрограмм
code    Ends              ; конец кодового сегмента
        END   start       ; конец текста программы
```

В начале примера Б.3 подряд расположены директивы `.Alpha` и `Dosseg`. При таком их сочетании стековый сегмент будет расположен в теле задачи последним, независимо от имен других сегментов (по крайней мере, так его располагает MASM 5.1).

Директива `.386` определяет набор команд, которые можно использовать в программе. Если ее не указать, то по умолчанию будет выбран набор команд микропроцессора Intel 8086. В таком случае Макроассемблер обнаружит ошибку в записи команды `shr bx, 04`. Подробнее о назначении и месте расположения этой директивы сказано в приложении В.

После загрузки задачи DOS передает управление на метку `start`, для этого ее имя указано не только перед первой командой, но и после директивы `END`. Первые две команды записывают в регистр `ds` код сегмента данных, этот вопрос мы уже обсуждали. Шесть следующих команд вычисляют в регистре `bx` код первого свободного сегмента и сохраняют его в переменной `freeseg`. Затем в регистр `ax` считывается из 2-го слова `PSP` код последнего доступного для задачи сегмента, вычисляется размер свободной памяти в параграфах и сохраняется в переменной `msize`.

Реальный размер памяти сравнивается с необходимым (`needm`), и если он достаточен для выполнения задачи, то произойдет переход на локальную метку `@@`. В противном случае на экран будет выведено аварийное сообщение и прекратится выполнение задачи. Способы вывода текстовых сообщений описаны в разделе 5.1.3 основной части книги, а завершение выполнения задачи описано в разделе 6.3.1.

Если памяти достаточно, то в переменную `SwpSeg` копируется код первого доступного сегмента. Для буфера обмена отведено 65 536 байтов, что в параграфах составляет `1000h`. Эта величина прибавляется к содержимому реги-

стра bx, и результат записывается в GenSeg. Значения переменных SwpOffs и GenOffs определяются в процессе выполнения задачи.

После выполнения команд примера Б.3 размер буфера обмена ограничен, поскольку после него расположен буфер общего назначения, поэтому при работе с SwpSeg нельзя выходить за пределы 65 536 байтов. Во всех ранее приведенных примерах такое ограничение нас вполне устраивало. Размер буфера общего назначения пока ограничен величиной (msize – 1000h)*16 байтов. В зависимости от конкретных особенностей задачи это пространство может быть разделено на блоки меньшего размера или использовано как большой буфер общего назначения.

При работе с блоками большого размера задача должна контролировать текущий адрес ОЗУ и при достижении границы 65 536 байтов изменять код в сегментном регистре, который используется для доступа к блоку (увеличивать его содержимое на 1000h). Необходимость работы с блоками ОЗУ большого размера возникает, например, при сохранении и восстановлении содержимого всей рабочей области экрана.

Мы описали простой пример размещения блоков в ОЗУ. Для выполнения более сложных функций, связанных с распределением памяти в текст задачи, придется включать специальные подпрограммы и поддерживать структуру данных, описывающих свободное и использованное пространство ОЗУ. Альтернативой является обращение к DOS для выполнения действий, связанных с распределением пространства оперативной памяти.

Б.1.4. Использование функций DOS

Для того чтобы DOS могла распределять оперативную память, ей надо вернуть все свободное пространство, расположенное за пределами сегментов, образующих тело задачи. Иначе говоря, надо сократить размер блока выделенного для выполнения задачи до ее реальных размеров.

Определение размера задачи. Будем предполагать, что стековый сегмент расположен в теле задачи последним. В таком случае нас интересует расстояние от начала PSP до конца стекового сегмента, выраженное в параграфах. Напомним, что код сегмента, содержащего PSP, находится в регистре es, код стекового сегмента — в ss, а если стек еще не использовался, то его размер в байтах содержится в регистре sp.

Размер тела задачи, выраженный в параграфах, вычисляется так:

```
tasksize = [ss] – [es] + [sp] /16
```

В этой формуле квадратные скобки указывают на то, что при вычислении используется содержимое регистров ss, es и sp. При программировании деление [sp] на 16 заменяется сдвигом на 4 разряда вправо. Для того чтобы не

потерять один параграф, размер стека должен быть кратен 16-ти, в противном случае его надо округлить в сторону увеличения.

Зная реальный размер задачи, можно запросить у DOS сокращение ее блока. В результате появится свободное пространство памяти, которое можно использовать по запросам задачи.

Функции DOS. По запросам прикладных программ DOS выполняет несколько функций, связанных с распределением оперативной памяти. Нас будут интересовать только три из них.

Все обращения к DOS происходят через прерывание `int 21h`, при этом код запрашиваемой функции указывается в регистре `ah`, а входные и выходные параметры располагаются в регистрах общего назначения и иногда в сегментных регистрах `ds` или `es`.

Функция `48h Allocate Memory` выделяет запрашиваемое пространство памяти. Требуемый размер памяти, выраженный в параграфах, указывается в регистре `bx`. Если такое пространство существует, то при возврате из DOS признак переполнения отсутствует, содержимое `bx` не изменяется, а в `ax` находится код сегмента, с которого начинается выделенное пространство памяти. Если свободное пространство нужного размера отсутствует, то при возврате из DOS вырабатывается признак переполнения (устанавливается в 1 С-разряд регистра флагов).

Функция `49h Free Allocated Memory Block` открепляет выделенный ранее для задачи блок памяти, после этого задача не может с ним работать. Код сегмента, начиная с которого расположен освобождаемый блок, помещается в регистр `es`, размер блока указывать не требуется, т. к. DOS хранит его в своей области данных. Если функция выполнена успешно, т. е. блок освобожден, то при возврате из DOS признак переполнения отсутствует. Если он установлен, то наиболее вероятно, что в регистре `es` был неверно указан сегмент блока.

Функция `4Ah Shrink or Expand a Memory Block` урезает или расширяет существующий блок памяти. Код сегмента, начиная с которого расположен изменяемый блок, указывается в регистре `es`, а новый размер блока — в регистре `bx`. Если функция выполнена успешно, т. е. размер указанного блока изменен, то при возврате из DOS признак переполнения отсутствует. Если он установлен, то блок не может быть расширен или сокращен. В этом случае в регистре `bx` возвращается наибольший (последний) доступный блок памяти.

Вычисление SwpSeg и GenSeg. В примере Б.4 показан фрагмент начала текста программы, в котором производится сокращение размера блока задачи и выделение двух блоков для размещения буферов обмена и общего назначения. Для описания основных сегментов в примере использованы специальные директивы (см. пример Б.2).

Пример Б.4. Получение от DOS значений переменных SwpSeg и GenSeg

```
        dosseg              ; задаем расположение сегментов
        .model small        ; выбор модели памяти
        .stack 200h         ; задаем стековый сегмент
        .data               ; начало сегмента данных
SwpOffs dw 0                ; смещение в буфере обмена
SwpSeg  dw 0                ; сегмент буфера обмена
GenOffs dw 0                ; смещение в буфере
GenSeg  dw 0                ; сегмент буфера общего назначения
;       Далее располагаются другие данные, используемые в задаче
        .code               ; начало кодового сегмента
        .386                ; набор команд процессора
start:  mov  ax, @data      ; ax = код сегмента данных
        mov  ds, ax         ; ds = ax
        mov  ax, ss         ; ax = код стекового сегмента
        mov  bx, es         ; bx = код сегмента, содержащего PSP
        sub  ax, bx         ; ax = ax — bx
        mov  bx, sp         ; bx = размер стека в байтах
        shr  bx, 04         ; превращаем его в параграфы
        add  bx, ax         ; bx = bx + ax, размер задачи
        mov  ax, 4A00h      ; код запроса на сокращение блока
        int  21h            ; DOS сокращает блок задачи
        mov  bx, 1000h      ; bx = размер блока в параграфах
        mov  ax, 4800h      ; код запроса на выделение блока
        int  21h            ; обращение к DOS
        mov  SwpSeg, ax     ; SwpSeg = код сегмента блока
        mov  bx, 1000h      ; bx = размер блока в параграфах
        mov  ax, 4800h      ; код запроса на выделение блока
        int  21h            ; обращение к DOS
        mov  GenSeg, ax     ; GenSeg = код сегмента блока
; Далее следует продолжение текста программы
        END  start          ; конец текста программы
```

Первые две команды примера Б.4 записывают в регистр `ds` код сегмента данных, который в этом случае имеет имя `@data`. Следующие 6 команд формируют в регистре `bx` значение `tasksize`, затем происходит вызов функции DOS `4Ah` для сокращения размера блока задачи. При выполнении этой функции неиспользуемая память возвращается DOS, и теперь ее можно запрашивать для размещения дополнительных блоков.

В примере Б.4 сначала запрашивается место для размещения буфера обмена, а затем буфера общего назначения. В обоих случаях в регистре `bx` указывается размер блока в параграфах (`1000h`), а в регистре `ah` — код функции `48h` (`Allocate memory`) и происходит обращение к DOS. После возврата из DOS

код выделенного сегмента сохраняется в переменных `SwpSeg` или `GenSeg`. В отличие от примера Б.3 размер буфера общего назначения в данном случае ограничен величиной 65 536 байтов.

В примере Б.3 перед вычислением значений переменных `SwpSeg` и `GenSeg` производилась проверка достаточности объема памяти для выполнения задачи. В данном случае распределением памяти занимается DOS, поэтому указанная проверка выполняется иначе.

Контроль выполнения запросов. При описании функций говорилось, что при возврате из DOS состояние C-разряда указывает, успешно или неудачно завершилось выполнение запроса. Контролировать результат выполнения функции `4Ah` не имеет смысла, лучше внимательно проверить запись команд, предшествующих обращению к DOS.

Результат исполнения функции `48h` надо обязательно проверять. Признак переполнения (`C=1`) при возврате из DOS означает, что в памяти нет места для размещения блока нужного размера (если исключены ошибки в тексте программы). Что делать в таких случаях?

Выдача аварийного сообщения и прекращение выполнения задачи является самым плохим способом реагирования на недостаток памяти. Всегда можно найти вариант продолжения нормального выполнения задачи. Если освободить место для блока в основной памяти невозможно, то его надо разместить в расширенной памяти, тем более что это никак не скажется на дальнейшей работе с блоком.

Освобождение блоков. Освободившийся блок возвращается DOS по запросу `49h` (`Free Allocated Memory Block`), но с этим действием можно не спешить. В среде DOS задача, как правило, выполняется в монопольном режиме, и кроме нее претендентов на свободную память нет. Перед завершением выполнения задачи освобождать использованные блоки не обязательно. После завершения задачи DOS обязательно освобождает все пространство памяти, которое было за ней закреплено.

Заключение. Мы описали три варианта выделения нужного пространства в основной памяти ПК: явное описание в исходном тексте задачи, размещение в свободной части блока задачи и его запросы у DOS. Первый способ предельно упрощает программирование, но значительно увеличивает размер файла, содержащего задачу. Два других варианта примерно равноценны, они не существенно увеличивают количество вспомогательных действий и почти не увеличивают размер задачи.

Размер основной памяти ПК невелик, поэтому при выборе места для размещения дополнительного пространства предпочтение следует отдавать расширенной памяти, которая описана в следующем разделе.

Б.2. Расширенная память (Expanded Memory)

Очень скоро после начала массового производства IBM PC XT стало ясно, что заложенная в нем возможность работы с адресным пространством в 1 Мбайт является серьезным препятствием для создания все более усложняющегося программного обеспечения. К этому времени технические средства уже позволяли преодолеть барьер 1 Мбайт, но для их использования надо было радикально изменить некоторые фундаментальные концепции, положенные в основу DOS, чего явно не хотели делать ни IBM, ни Microsoft. В то время многие фирмы предлагали различные компромиссные решения в виде аппаратных и программных средств, позволяющих работать с дополнительной памятью без изменения структуры DOS и BIOS. В конечном итоге только одно из них стало стандартом при работе с дополнительной памятью, не потому, что оно было самым лучшим, а потому что его разработку, массовый выпуск и дальнейшую поддержку взяли на себя такие крупные корпорации, как Lotus, Intel и Microsoft.

Б.2.1. Спецификация расширенной памяти

Спецификация расширенной памяти (Expanded Memory Specification или EMS) содержит перечень требований к оборудованию, предназначенному для работы с дополнительным пространством адресов ОЗУ, и совокупность правил, которых надо придерживаться при работе с этим оборудованием. Если в ПК используется микропроцессор Intel 386 и выше, то необходимое оборудование заведомо существует и нет необходимости в приобретении и установке каких-либо дополнительных устройств. В данном разделе нас будет интересовать возможность его использования.

Основная концепция EMS. Дополнительная память занимает физические адреса от A0000h и далее вплоть до верхнего предела. Если на ПК установлено 16 Мбайт ОЗУ, то последнему байту соответствует физический адрес 0FFFFFFh. Напомним, что для DOS и BIOS последний физический адрес ОЗУ равен 9FFFFh, а адреса от 0A0000h до 0FFFFFh к оперативной памяти не относятся. Таким образом, при работе в DOS прямой доступ к расширенной памяти невозможен и применяется косвенный доступ.

В доступном для DOS пространстве адресов выделяется окно (сегмент), на которое отображается фрагмент расширенной памяти. Отображение окна на пространство физических адресов и его перемещение выполняет специальная программная компонента (менеджер). Ее присутствие является необходимым условием для работы с расширенной памятью.

Системные и прикладные задачи получают доступ к расширенной памяти, издавая специальные команды, которые исполняет менеджер. В соответст-

вии с идеологией DOS и BIOS они оформляются в виде программных запросов.

Впервые EMS была опубликована сразу в виде версии 3.0 в 1985 году, в ее подготовке приняли участие только Lotus и Intel. Примерно в это же время Microsoft в связи с разработкой Windows заинтересовалась EMS и принимала активное участие в разработке ее последующих версий. В 1987 году была опубликована версия 4.0, которая предусматривала возможность размещения и выполнения нескольких задач в расширенной памяти. Автору неизвестно существование более поздних версий EMS, но для изложения материала это не существенно, т. к. соблюдается строгая преемственность версий и то, что описано в данном разделе может использоваться в ваших программах.

Менеджер расширенной памяти (Expanded Memory Manager или EMM) оформлен в виде драйвера, который располагается в оперативной памяти при загрузке DOS и остается резидентным до выключения ПК. Обычно его имя EMM386.EXE, оно обязательно указывается в файле config.sys, например, так:

```
DEVICE = C:\DOS\EMM386.EXE
```

В данном случае предполагается, что файл EMM386.EXE расположен на диске C, в каталоге DOS. Драйвер многофункциональный, поэтому после его имени в командной строке могут указываться параметры, которые используются при загрузке. Описание всех параметров вы найдете в файле HELP, входящем в комплект DOS.

Важно

Один из параметров, а именно NOEMS запрещает поддержку описываемых ниже функций драйвера, поэтому его указание в командной строке недопустимо.

EMS 4.0 исполняет 30 различных функций, разделенных на три группы: стандартные (standard), расширенные (advanced) и для многозадачных режимов (OS). Нас будут интересовать только стандартные функции, необходимые для работы с данными, расположенными в расширенной памяти, их всего 7. Описание остальных функций вы найдете в TECH HELP или в специальной литературе по работе с DOS.

Доступ к драйверу осуществляется через прерывание int 67h. Перед выдачей запроса код функции помещается в регистр ah (старший байт регистра ax). Регистр al используется либо для уточнения запрашиваемой функции (для advanced и OS), либо для указания параметров функции. В случае успешного исполнения запроса EMM возвращает в регистре ah 0, а случае неудачного — 1.

EMS 4.0 фиксирует 36 различных ошибок, им присваиваются коды от 80h до 0A4h включительно. Полный список ошибок содержится в TECH HELP, некоторые из них будут названы при описании функций. Ошибки возникают по

разным причинам. На стадии отладки они, чаще всего, вызваны некорректностью программы. Если корректность программы не вызывает сомнений и задача неоднократно выполнялась успешно, то имеет смысл посмотреть, какие параметры EMM386 заданы в файле config.sys. Например, если указан параметр NOEMS, то при запросах памяти обязательно будет возникать "внутренняя ошибка EMM драйвера", имеющая код 80h.

Описание стандартных функций. К категории стандартных относятся следующие функции драйвера EMM.

Функция 40h Get EMM Status предназначена для получения информации о состоянии драйвера. Входные параметры отсутствуют. Результат проверки возвращается в регистре ah. Если он очищен, то все в порядке. Код 81h означает неисправность расширенной памяти, других кодов ошибки в данном случае не должно быть. В случае ошибки надо прекратить выполнение задачи и проверить состояние системного программного обеспечения.

Функция 41h Get Physical Segment Address of EMS Frame возвращает значение сегмента, используемого для доступа к расширенной памяти. Входные параметры отсутствуют, код сегмента указан в регистре bx. Его надо сохранить и при чтении или записи данных помещать в один из сегментных регистров. Код сегмента зависит от конфигурации оборудования, установленного на компьютере, в процессе выполнения задачи он не может и не должен изменяться. Размер сегмента 64 Кбайт, он делится на 4 страницы по 16 Кбайт каждая. Адреса (смещения) страниц равны 0, 4000h, 8000h и 0C000h.

Функция 42h Get EMS Memory size позволяет определить общее и свободное пространство expanded memory. Входные параметры у нее отсутствуют. Общий размер памяти возвращается в регистре dx, а размер свободного пространства — в bx. Обе величины указаны в виде количества страниц, размер страницы равен 16 Кбайт.

Обычно не вся внешняя память ПК используется как expanded memory. Часть ее занимает DOS и другие резидентные программы. Кроме того, при установке EMM (в файле config.sys) может быть указан размер доступного для него пространства ОЗУ. Поэтому не следует ожидать, что находящееся в регистре dx количество страниц всегда соответствует размеру старшей памяти ПК.

Для выполнения задачи важно знать размер свободной части памяти, т. е. количество страниц, указанное в регистре bx. При программировании задачи необходимо предусмотреть проверку доступного размера памяти и выбор дальнейших действий в зависимости от его значения.

Функция 43h Allocate Memory and Open EMM handle запрашивается для выделения требуемого пространства расширенной памяти. Входным параметром является необходимое количество страниц, указываемое в регистре bx. При успешном выполнении функции в регистре dx возвращается EMM

`Handle` — идентификатор выделенного блока, который используется при запросах других функций драйвера. Его обязательно надо сохранить.

Два специальных кода ошибок означают, что запрошено слишком много страниц. Код `87h` выдается, если общее пространство `expanded memory` меньше запрошенного размера, а код `88h` означает, что недостаточно свободного пространства памяти. Если вы предварительно проверяли свободную память с помощью функции `42h`, то эти две ошибки не должны возникать, в противном случае надо прекратить выполнение задачи.

Выделенное пространство закреплено за задачей, но перед записью или чтением данных надо отображать его конкретную часть на рабочий сегмент, значение которого возвращает функция `41h`. Как уже говорилось ранее, в этом сегменте размещается 4 страницы. Для отображения 16 Кбайт ОЗУ на одну из этих страниц предназначена функция `44h`.

Функция `44h Map Memory` связывает физическую страницу с логической. Входными параметрами являются номера физической (`al`) и логической (`bx`) страниц и `EMM handle` (`dx`). Номер физической страницы может изменяться от 0 до 3, а номер логической страницы — от 0 до `n-1`, где `n` — количество страниц, выделенное функцией `43h`.

Возможные ошибки имеют следующие коды: `83h` — неверно задан идентификатор блока (`EMM handle`), `8Ah` — логическая страница вне диапазона значений, выделенных функцией `43h`, `8Bh` — недопустимая физическая страница.

После отображения логическая страница доступна для использования. Для работы с полным сегментом (65 536 байт) функция `44h` выполняется 4 раза. При этом не обязательно указывать физические и логические страницы в порядке увеличения их номеров.

Функция `45h Release Memory` освобождает пространство памяти, выделенное для задачи функцией `43h`. Входным параметром является идентификатор блока, указываемый в регистре `dx`. Отказ от выполнения функции может быть связан только с его неправильным заданием.

Важно

Данная функция *обязательно* должна вызываться перед завершением выполнения задачи, в противном случае выделенная для завершенной задачи память будет недоступна для других задач вплоть до перезагрузки системы.

Функция `46h Get EMM version number` позволяет определить версию менеджера `EMM`. Входные параметры отсутствуют, номер версии в двоично-десятичном коде возвращается в регистре `al`. Например, версии 6.2 соответствует код `62h`.

Таким образом, при использовании стандартного набора функций драйвера `EMM` задача получает возможность резервирования нужного пространства в дополнительной памяти ПК и размещения в нем данных.

Б.2.2. Использование функций драйвера

Для корректной работы с расширенной памятью задачи должны выполнять определенную последовательность действий, а именно:

1. Проверить наличие менеджера, поддерживающего функции EMS.
2. Получить код сегмента, на который отображается расширенная память.
3. Определить наличие требуемого пространства расширенной памяти.
4. Отобразить часть пространства ОЗУ на физические страницы.
5. В пределах сегмента работать с расширенной памятью как с обычной.
6. При достижении границ сегмента повторять пункты 4 и 5.
7. Перед завершением задачи вернуть память менеджеру.

Выполнение первого пункта списка является скорее данью традиции, чем необходимостью, поскольку ПК на базе Intel 386 и всех последующих моделей обязательно имеют оборудование для доступа к расширенной памяти. Для проверки можно, например, с помощью функции 40h определить статус, а с помощью функции 46h — номер версии драйвера и убедиться, что он не меньше чем 4.0. Остальные пункты списка обязательно должны выполняться, причем в той последовательности, в которой они перечислены.

Специальные переменные. При работе с Expanded memory обязательно используются код сегмента расширенной памяти и идентификаторы выделенных блоков. В некоторых случаях могут быть нужны номера последних логических страниц в выделенных блоках и другие величины. Для их хранения в разделе данных задачи надо выделить специальные переменные, количество которых зависит от количества открытых блоков.

Если задача запрашивает у драйвера только один блок большого размера, то для работы с ним нужны следующие переменные:

```
EBuff   dw 0    ; код сегмента для доступа к расширенной памяти
Ehndlr  dw 0    ; идентификатор блока выделенного для задачи
Curpg   db 0    ; номер текущей логической страницы блока
Lastpg  db 0    ; номер последней логической страницы блока
```

Номера текущей и последней страниц нужны при работе с большими блоками, размер которых превышает 65 536 байтов (4 страницы). При работе с ними приходится многократно отображать его логические страницы на физические. Если же размер блока не превышает стандартного сегмента ОЗУ, то он отображается на физические страницы только один раз и при дальнейшей работе номера страниц не нужны.

При работе с несколькими блоками в разделе данных задачи можно организовать простую таблицу, состоящую из строк фиксированного размера, содержащих характеристики каждого блока. В таком случае характеристики

выбираются из таблицы по порядковому номеру блока, и сокращается количество имен переменных.

Резервирование блока. Предположим, что для выполнения задачи требуется непрерывное пространство расширенной памяти (блок) размером в 1 Мбайт. Для резервирования такого блока задача должна запросить у драйвера исполнение функции `43h`, указав в регистре `bx` 64 страницы (размер страницы составляет 16 Кбайт). Если это первый запрос, обращенный к драйверу, то предварительно надо выполнить функцию `41h` для определения состояния драйвера и получения кода сегмента для доступа к памяти.

Фрагмент программы, выполняющий резервирование блока размером в 1 Мбайт, приведен в примере Б.5. Его надо включить в ту часть задачи, где выполняются подготовительные действия. Например, сразу после команд, приведенных в примерах Б.3 или Б.4.

Пример Б.5. Создание в расширенной памяти блока размером 1 Мбайт

```
      mov   ax, 4100h  ; код функции запроса сегмента
      int   67h        ; обращение к драйверу
      or    ah, ah     ; функция выполнена ?
      je    @F         ; -> да
      jmp   emmerr     ; -> ошибка при исполнении функции
@@:   mov   EBuff, bx  ; сохраняем код сегмента
      mov   bx, 64     ; размер запрашиваемого блока
      mov   ax, 4300h  ; код функции выделения памяти
      int   67h        ; обращение к драйверу
      or    ah, ah     ; блок выделен ?
      je    @F         ; -> да
      jmp   emmerr     ; -> ошибка при выделении блока
@@:   mov   Ehndlr, dx ; сохраняем идентификатор блока
      mov   Curpg, 0   ; номер текущей страницы блока
      mov   Lastpg, 63 ; номер последней страницы блока
;     Продолжение текста программы
```

Выполнение примера Б.5 начинается с запроса кода сегмента для доступа к расширенной памяти. Если при возврате из драйвера регистр `ah` очищен, то запрос исполнен успешно, в противном случае произойдет переход на метку `emmerr`, для вывода аварийного сообщения.

На следующем шаге выдается запрос на выделение блока размером в 64 страницы. Если после возврата из драйвера регистр `ah` очищен, то блок выделен, в противном случае происходит переход на метку `emmerr`.

В случае успешного выделения блока формируются `Curpg` и `Lastpg` и на этом выполнение фрагмента завершено.

В примере Б.5 отсутствуют команды, обработки аварийных ситуаций. Предполагается, что первая из них имеет метку `emmerr`. Что делать в случае

ошибки, решать вам, например, можно вывести на экран текст аварийного сообщения и завершить выполнение задачи. На стадии отладки полезно предусмотреть вывод кода ошибки, который находится в регистре ah.

Напоминаем, что после выполнения команд примера Б.5 блок только закреплен за задачей, но не доступен для записи или чтения. Для работы с его конкретными страницами их надо отобразить на физические страницы сегмента EMS. В этом заключается одно из существенных отличий доступа к блокам расширенной памяти от доступа к блокам обычной памяти.

Отображение страниц. Для отображения логической страницы блока на одну из физических страниц сегмента EMS запрашивается функция 44h. Мы рассмотрим универсальный вариант подпрограммы отображения страниц.

В зависимости от логики выполняемых в задаче действий может потребоваться отображение от одной до четырех страниц. Поэтому подпрограмма, текст которой приведен в примере Б.6, имеет две точки входа.

При вызове call mapseg отображаются четыре подряд расположенные страницы. Предварительно в регистре bx указывается номер первой логической страницы, а в регистре dx — идентификатор блока, которому принадлежат отображаемые страницы. Вариант вызова подпрограммы mapseg показан в примере Б.7 (см. раздел Б.2.3).

При вызове call maplp дополнительно (кроме заполнения регистров bx и dx) в регистре ax указывается номер первой физической страницы, а в регистре cx — количество страниц.

Если отображение страниц выполнено успешно, то при возврате из подпрограммы отсутствует признак переполнения (C-разряд очищен), в противном случае он установлен. Следует отметить, что при работе отлаженной задачи аварийные ситуации исключены.

Пример Б.6. Подпрограмма отображения 4-х страниц сегмента EMS

```
mapseg: mov   cx, 04    ; количество повторов цикла
        xor   al, al    ; нулевая физическая страница
maplp:  mov   ah, 44h   ; код функции отображения памяти
        int   67h       ; обращение к драйверу
        or    ah, ah    ; отображение выполнено?
        jne   @F        ; -> нет, переход на локальную метку
        inc   ax        ; следующая физическая страница
        inc   bx        ; следующая логическая страница
        loop  maplp     ; управление повторами цикла
        clc             ; очистка C-разряда
        ret             ; возврат на основную программу
@@:     stc             ; установка C-разряда
        ret             ; возврат на основную программу
```

При входе в точку `mapseg` задается 4 повтора цикла отображения начиная с нулевой физической страницы. После этого выполняется цикл отображения, имеющий метку `maplp` (вторая точка входа).

Если отображение происходит без ошибок, то номера физической и логической страницы увеличиваются на 1 и цикл повторяется до тех пор, пока не будет отображено заданное количество страниц.

При возникновении ошибки ее код возвращается в регистре `ah`. В этом случае выполнение цикла прекращается и происходит переход на локальную метку, что приведет к возврату из подпрограммы с установленным признаком переполнения.

Освобождение памяти. Для освобождения выделенного задаче блока выполняются следующие действия:

```
mov    dx, Ehndlr      ; dx = идентификатор блока
mov    ax, 4500h       ; ax = код функции освобождения блока
int    67h             ; обращение к драйверу
```

При исполнении запроса драйвер открепляет пространство блока от задачи, и оно становится общедоступным.

Как уже говорилось, перед завершением задачи освобождение выделенной для нее расширенной памяти *обязательно*. В противном случае это пространство окажется недоступным для других претендентов и будет освобождено только при выключении или перезагрузке компьютера.

Освобождаются все блоки, затребованные задачей. Для этого описанные команды повторяются, с указанием в регистре `dx` идентификаторов разных блоков, закрепленных за задачей.

Б.2.3. Работа с расширенной памятью

После резервирования блока и отображения части или всех его логических страниц с расширенной памятью могут работать все без исключения команды микропроцессора. В данном разделе описан пример пересылки большого массива данных и обсуждается возможность одновременного использования двух блоков, расположенных в расширенной памяти.

Способ пересылки большого блока. При выполнении графических задач может потребоваться сохранение содержимого всего рабочего пространства видеопамяти. Учитывая его большие размеры, сохранение в обычной памяти либо не целесообразно, либо просто невозможно, для этого лучше подходит расширенная память.

Размер рабочего пространства видеопамяти зависит от установленного видеорежима, он вычисляется умножением размера строки в байтах на количество строк на экране (`bperline * versize`). Для временного хранения про-

изведения нужна переменная, состоящая из двух слов (но не одно двойное слово), назовем ее `blsize`. В первом слове будет храниться старшая часть произведения, а во втором слове — младшая. Старшая часть указывает количество полных окон видеопамяти, а младшая часть — количество байтов в последнем окне, оно может быть равно нулю.

Если в результате умножения в первом слове `blsize` окажется число `N`, а во втором `blsize+2` число `M`, то размер отображаемой части видеопамяти составляет `N*65536 + M` байтов, т. е. надо переслать `N` полных сегментов и еще `M` байтов. Поэтому перед пересылкой очередного фрагмента данных необходимо уточнять его размер.

Кроме того, перед пересылкой каждого фрагмента производится отображение очередной группы логических страниц блока, выделенного в расширенной памяти на физические с помощью подпрограммы `mapseg`, приведенной в примере Б.6 Для исключения лишних проверок можно каждый раз отображать по 4 страницы. В процессе отображения `mapseg` изменяет номера логических страниц (содержимое регистра `bx`), поэтому при работе с ней достаточно задать номер исходной страницы и в дальнейшем просто не изменять текущее содержимое регистра `bx`.

Для упрощения и ускорения пересылки нужен микропрограммный цикл на основе строковой операции `movs`, работающей с двойными словами.

Подпрограмма пересылки блока. Программная реализация пересылки показана в примере Б.7. Перед вызовом подпрограммы в регистрах `es` и `fs` указываются коды видеобуфера и сегмента `EMS`, в примере Б.8 показано, как это делается. Предполагается, что в разделе данных задачи описана переменная `blsize`, а в расширенной памяти зарезервирован блок, размер которого не меньше размера отображаемой части видеопамяти.

Пример Б.7. Пересылка содержимого рабочей области экрана

```
movebl:  push  Cur_win       ; сохраняем текущее окно видеопамяти
         pusha               ; сохраняем "все" регистры
         mov   ax, BaseWin   ; ax = BaseWin (или ax = 0)
         mov   Cur_win, ax   ; Cur_win = BaseWin
         call  Setwin        ; установка нулевого окна видеопамяти
         mov   ax, bperline  ; ax = размер строки в байтах
         mul   versize       ; dx:ax = bperline * versize
         mov   blsize, dx    ; blsize = число полных окон
         mov   blsize+2, ax  ; blsize +2 = размер последнего окна
         xor   bx, bx        ; bx = исходная логическая страница
         mov   dx, Ehndlr    ; dx = идентификатор файла
         xor   di, di        ; di = 0 исходный адрес
         xor   si, si        ; si = 0 исходный адрес
```

```
mloop:  mov   cx, 4000h      ; 0,25 размера стандартного сегмента
        dec   blsize         ; уменьшаем количество сегментов
        jns   sc_1           ; -> пересылка полного окна
        mov   cx, blsize+2   ; cx = размер последнего окна
        shr   cx, 02         ; уменьшаем его в 4 раза
        je    sc_2           ; -> окно пустое, пересылка окончена
sc_1:   call  mapseg         ; отображаем очередные 4 страницы
        rep movs dword ptr [di], fs:[si]; цикл пересылки
        call  Nxtwin         ; следующее окно видеопамяти
        cmp   blsize, -1     ; пересылка завершена ?
        jne   mloop          ; -> нет, продолжаем пересылку
sc_2:   pop   Cur_win        ; исходное значение видеоокна
        popa                 ; восстановление "всех" регистров
        call  Setwin         ; восстановление исходного окна
        ret                  ; возврат из подпрограммы
```

Выполнение примера Б.7 начинается с сохранения исходного окна видеопамяти, содержимого всех регистров и установки базового окна. Если переменная `BaseWin` в задаче не используется, то надо просто установить нулевое окно. Затем вычисляется размер отображаемой области видеопамяти, и результат сохраняется в словах `blsize` и `blsize+2`. В регистр `bx` помещается номер нулевой логической странице, а в `dx` — идентификатор блока. Содержимое этих двух регистров использует только подпрограмма `mapseg`. Подготовка оканчивается очисткой содержимого индексных регистров `si` и `di`.

Основной цикл имеет метку `mloop`. Его первые шесть команд определяют размер фрагмента пересылаемых данных. Он составляет 16 384 двойных слова, если окно заполнено полностью, или равен значению слова `blsize+2`, уменьшенному в 4 раза, если окно заполнено частично.

Команда, имеющая метку `sc_1`, отображает очередные четыре страницы блока на сегмент `EMS`. Затем микропрограммный цикл пересылает очередной фрагмент данных. Он выполняет основную работу, все остальные команды примера Б.7 являются вспомогательными.

После пересылки очередного фрагмента проверяется содержимое `blsize`, и работа подпрограммы продолжается до тех пор, пока его значение не окажется равным "−1". В этом случае из стека выталкиваются значения переменной `Cur_win` и сохраненных регистров, восстанавливается исходное окно видеопамяти и происходит возврат на вызывающий модуль.

Сохранение и восстановление рабочей области экрана. В примере Б.7 основные действия выполняет строковая операция `movs`, у которой расположение источника задает регистр `fs`, а приемника — `es`. Следовательно, для сохранения содержимого видеопамяти в расширенной памяти в регистр `fs` надо записать код видеосегмента, а в `es` — код сегмента `EMS`. Для восстановления

содержимого видеопамяти, сохраненного в расширенной памяти в регистре fs, указывается код сегмента EMS, а в es — код видеобуфера. Формирование нужных значений в регистрах es и fs выполняют подпрограммы, приведенные в примере Б.8. Для сохранения содержимого видеопамяти используется обращение к подпрограмме scrsave, а для восстановления — к scrrest. Входные параметры отсутствуют.

Пример Б.8. Сохранение или восстановление рабочей области экрана

```
scrsave: PushReg <fs,es,vbuff,ebuff>; размещение в стеке
         jmp   short @F      ; обход макровызова
scrrest: PushReg <fs,es,ebuff,vbuff>; размещение в стеке
@@:      call  Hidepnt       ; удаление изображения курсора
         PopReg <es,fs>      ; формируем содержимое es и fs
         call  Movebl        ; перемещение блока
         PopReg <es,fs>      ; восстановление содержимого es и fs
         call  Showpnt       ; вывод курсора на экран
         ret                 ; возврат из подпрограммы
```

При вызове scrsave в стеке сохраняется исходное содержимое регистров fs, es и переменных vbuff, ebuff. При обращении к scrrest порядок записи в стек переменных ebuff, vbuff противоположный. После размещения в стеке нужных величин выполняется общая часть обеих подпрограмм.

Прежде всего надо удалить изображение курсора с экрана, иначе при сохранении оно станет частью общей картины, а при восстановлении на экране могут появиться два курсора, или при перемещении на месте курсора окажется прямоугольник другого цвета.

После этого в регистры fs и es выталкиваются из стека нужные величины, происходит обращение к подпрограмме movebl и восстанавливается исходное содержимое регистров fs и es.

Выполнение подпрограмм заканчивается восстановлением изображения курсора на экране. Напомним, что варианты подпрограмм Hidepnt и Showpnt описаны в главе 6.

Несколько блоков в расширенной памяти. При работе с обычной памятью каждому блоку соответствует свой уникальный код сегмента. В отличие от обычной, при работе с расширенной памятью доступ ко всем зарезервированным задачей блокам осуществляется через один и тот же сегмент EMS. В таком случае по коду сегмента невозможно определить блок, к которому происходит обращение. Для этой цели можно использовать только смещение (адрес), по которому выбираются или записываются данные. Для того чтобы понять, к чему это приводит, рассмотрим простой вариант работы с двумя блоками.

Предположим, что доступны два блока, и надо переписать данные из одного в другой. Если один из блоков расположен в видео или в обычной памяти, а другой в расширенной, то размер пересылаемой порции данных ограничен адресным пространством сегмента, т. е. величиной 65 536 байтов. В этом вы могли убедиться на примере Б.7.

Если же оба блока расположены в расширенной памяти, то пространство сегмента `EMS` придется разделить пополам и использовать младшие адреса для работы с одним из блоков, а старшие — с другим. В результате этого в каждом блоке будет доступно пространство размером 32 768 байтов. Никаких других ограничений нет.

В таком случае перед копированием порции данных придется дважды обратиться к подпрограмме отображения страниц, описанной в примере Б.6, через точку входа `maplp`. Сначала отображаются две очередные логические страницы блока 1 на физические страницы 0 и 1, затем две очередные логические страницы блока 2 на физические страницы 2 и 3. Блок 1 начинается с нулевого адреса сегмента `EMS`, а блок 2 с адреса `8000h` того же сегмента. Предельный размер доступного пространства в обоих блоках составляет `8000h` или 32 768 байтов. После этого можно использовать любые команды для работы с отображенным пространством, например, строковую операцию `movs`, выполняющую перемещение данных из блока в блок.

Если особенности алгоритма требуют одновременной работы с тремя или четырьмя блоками, то доступное пространство будет ограничено размером одной страницы, т. е. величиной 16 384 байта.

Здесь уместно отметить, что в набор `Advanced Functions` драйвера `EMM` включена специальная функция, предназначенная для перемещения или обмена содержимого (перестановки) двух блоков данных размером до 1 Мбайт, ее код `57h`. Блоки могут располагаться в обычной или расширенной памяти. Перед вызовом функции `57h` в регистре `al` указывается 0 для пересылки или 1 для перестановки блоков. Кроме того, в регистрах `ds:si` указывается адрес начала специальной структуры данных, содержащей размер блока и данные об источнике и приемнике. Описание этой функции вы можете найти, например, в `Tech Help`, нам важно было напомнить о ее существовании.

Заключение. Мы закончили описание основных видов оперативной памяти, поэтому можно подвести общий итог. При работе в среде DOS для программ доступна как основная, так и дополнительная память. Реальные размеры последней существенно больше размеров первой, поэтому при разработке задач следует отдавать предпочтение расположению больших блоков в расширенной памяти, а блоки небольшого размера размещать в обычной памяти. Кроме того, следует избегать одновременного использования нескольких блоков, расположенных в расширенной памяти, т. к. это связано с ограничением доступного пространства адресов.

Б.3. Расширенная память (Extended Memory)

Термин Extended memory относится к тому же пространству памяти, которое описано в предыдущем разделе, но обозначает другой способ доступа, а именно, непосредственную работу с его адресами. Такой способ доступа возможен при работе микропроцессора в *защищенном режиме* (protected-address mode). Напомним, что свое название режим получил потому, что микропроцессор контролирует адреса при любых обращениях к внешним устройствам, в том числе и к оперативной памяти.

Начиная с модели Intel 386, в защищенном режиме микропроцессоры оперируют 32-разрядными адресами, что соответствует пространству в 4 Гбайт или 4096 Мбайт. Это очень большое пространство, для рационального использования и контроля допустимости адресов оно делится на страницы размером по 4 Кбайт. Учитывая, что реальный объем оперативной памяти намного меньше 4 Гбайт, предусмотрен механизм подкачки страниц.

Как и при работе в реальном режиме, адрес ОЗУ состоит из двух частей, одна из которых находится в сегментном регистре, а другая — в индексных регистрах, регистрах указателях или в регистрах общего назначения. Отличие в том, что доступное пространство может быть больше чем 65 536 байтов, но оно всегда ограничено конкретной величиной, иначе будет невозможен контроль адресов. Также изменяется содержимое сегментных регистров (cs, ds, es, fs, gs, ss), в них кроме кода сегмента хранятся его характеристики, необходимые микропроцессору для контроля адресов.

DOS сама не использует и не поддерживает выполнение прикладных задач в защищенном режиме. Тем не менее задача может самостоятельно перевести микропроцессор в защищенный режим, а после выполнения, восстановить реальный режим перед возвратом в DOS. Однако в таком случае в ней придется выполнять много специфических действий, которые обычно возлагаются на операционные системы. Для выполнения таких действий предназначены расширители (DOS extenders), которые подключаются к прикладной задаче и создают на время ее выполнения вычислительную среду, необходимую для работы в защищенном режиме. Наиболее известными из них являются DOS4GW, DOS32A, PMODE/W.

В некоторых случаях DOS и служебные программы все же переключаются в защищенный режим для использования Extended memory. Поэтому BIOS выполняет простейшую форму поддержки работы в защищенном режиме. В данном разделе приведена ее краткая характеристика.

Менеджер Extended memory. В состав DOS входит драйвер, хранящийся в файле himem.sys, его спецификация обязательно указывается в первой строке файла config.sys. Этот драйвер выполняет несколько функций, связанных с доступом к дополнительной памяти в режиме Extended memory.

Одна из них заключается в тестировании и определении объема дополнительной памяти, сообщение о том, что himem тестирует память, можно увидеть в процессе загрузки DOS. Тестирование можно запретить, указав в config.sys ключ /testmem:off. Основное назначение himem.sys заключается в загрузке в дополнительную память резидентной части DOS.

В дополнительную память могут загружаться и драйверы различного назначения. В autoexec.bat признаком этого является команда LH, которая предшествует спецификации файла драйвера. В config.sys в таком случае вместо команды DEVICE используется DEVICEHIGH. При первоначальной установке DOS на компьютере все драйверы загружаются в обычную память. После того как файлы autoexec и config окончательно сформированы (завершено конфигурирование системы), выполняется специальная задача memmaker.exe, которая перемещает драйверы в старшую память, для увеличения свободного пространства в обычной памяти.

Драйвер himem.sys только загружает резидентные задачи в старшую память. Для выполнения таких задач надо либо переводить микропроцессор в защищенный режим, либо вызывать их так, как будто они находятся в Expanded memory. При работе в среде DOS используется второй способ.

Поддержка BIOS. После выпуска микропроцессора Intel 286 в состав BIOS была включена группа функций с названием AT Services, доступных через прерывание int 15h. Две из них имеют отношение к работе с Extended memory, а еще одна используется для перехода в защищенный режим. Следует отметить, что изначально они создавались для специальных целей и не рассчитаны на использование в прикладных задачах. Полное описание всех функций группы 15h можно найти в Tech Help или в одном из руководств по BIOS.

Функция 87h Move Extended Memory Block перемещает блок данных из расширенной памяти в обычную, или в обратном направлении. Размер блока, выраженный в словах, указывается в регистре cx, он не может превышать 8000h, т. е. 32K слов, или 64 Кбайт. В регистры es:si помещается адрес Global Descriptor Table (GDT), содержащий описание источника и приемника.

Признаком успешного выполнения пересылки является очищенный C-разряд при возврате из BIOS. При возникновении аварийной ситуации BIOS прекращает пересылку, устанавливает C-разряд при возврате в задачу, а в регистре ah указывает код ошибки (1, 2, 3).

Для GDT надо зарезервировать 48 байтов памяти, 38 из которых имеют постоянное значение, а 10 заполняются задачей перед обращением к BIOS, они содержат адреса и размеры источника и приемника. Напомним, что данные читаются из источника и записываются в приемник. Таблицу можно зарезервировать, например, с помощью директив, приведенных в примере Б.9.

Пример Б.9. Структура таблицы GDT

```
GDtab: db   16 dup (0) ; 16 пустых байтов
       dw   ?          ; размер источника в байтах (2*[cx]+1)
       dw   ?          ; младшая часть адреса источника
       db   ?          ; старшая часть адреса источника
       db   93h        ; разрешены чтение и запись
       dw   0          ; пустое (резервное) слово
       dw   ?          ; размер приемника в байтах (2*[cx]+1)
       dw   ?          ; младшая часть адреса приемника
       db   ?          ; старшая часть адреса приемника
       db   93h        ; разрешены чтение и запись
       dw   0          ; пустое (резервное) слово
       db   16 dup (0) ; 16 пустых байтов
```

Формат GDT должен строго соблюдаться, поэтому обратите внимание на то, в каких случаях в примере Б.9 употребляются директивы db, а в каких dw. Важно также правильно указывать коды доступа к источнику и приемнику, в частности, 93h разрешает чтение и запись.

Коды адресов источника и приемника 24-разрядные. Адрес обычной памяти вычисляется по схеме, показанной в табл. Б.1. Адрес Extended memory может изменяться в пределах от 10:0000h до 0FF:0FFFFh.

Замечание

Именно эту функцию используют: DOS для загрузки своей резидентной части в старшую память, задача memmaker.exe для перемещения драйверов в старшую память, а также драйверы vdisk и ramdrive. Для ее использования прикладными задачами в Extended memory надо выделить блок нужного размера. Для этого в файле config.sys после имени драйвера himem.sys укажите ключ /int15 = xxxx, где xxxx соответствует размеру (в килобайтах) пространства ОЗУ, которое будет доступно при работе с функцией 87h прерывания int 15h.

Функция 88h Get Extended Memory Size возвращает в регистре ax размер доступного пространства расширенной памяти, выраженный в килобайтах. Это то значение, которое указано при установке драйвера himem.sys, адрес его первого байта 100000h (1 Мбайт).

При работе с Extended memory задача, прежде всего, должна издать эту функцию для проверки наличия требуемого пространства ОЗУ. Если его недостаточно, то выполнение задачи надо прервать, поскольку она не может затребовать дополнительный объем расширенной памяти. Доступным пространством памяти задача распоряжается самостоятельно. Ни DOS, ни драйвер himem.sys не выполняют никаких функций контроля и распределения пространства Extended memory.

Функция 89h Enter Protected Mode выполняет действия, необходимые для перехода в защищенный режим и переводит микропроцессор в этот режим, т. е. после возврата из BIOS задача уже будет выполняться в защищенном режиме. Напомним, что временный переход в защищенный режим производится при выполнении функции 87h перед пересылкой блока, но он не заметен для задачи. В данном случае речь идет о полном переходе на выполнение задачи в защищенном режиме.

Перед обращением к BIOS надо сформировать специальную структуру данных (Global Descriptor Table и Interrupt Descriptor Table). Для заполнения этой структуры вы должны иметь представление о том, что такое дескриптор сегментного регистра, зачем и как надо изменять содержимое векторов прерываний и другие особенности перехода из реального режима в защищенный и обратно. Поэтому советуем отложить эксперименты с данной функцией до тех пор, пока вы не начнете изучать программирование для защищенного режима.

Виртуальный диск. Если на компьютере установлен достаточно большой объем оперативной памяти, то часть его можно использовать для размещения виртуального (или электронного) диска. Большой объем понятие условное, но при наличии 64 Мбайт памяти 32 Мбайт можно отдать под виртуальный диск, причем с явной пользой для дела. Если, например, вы измените настройки Windows так, чтобы область свопинга находилась на виртуальном диске, то система будет работать гораздо быстрее.

В состав устаревших версий DOS входил специальный драйвер vdisk.sys, предназначенный для создания и поддержки работы диска в дополнительной памяти ПК. Для чтения с диска и записи на него драйвер использовал функцию 87h прерывания int 15h, т. е. при каждом обращении к виртуальному диску происходил временный переход в защищенный режим.

В состав современных версий DOS входит улучшенная версия драйвера ramdrive.sys. При его установке можно выбирать способ доступа к дополнительной памяти. При указании ключа /E диск будет расположен в области Extended memory, а при указании ключа /A — в Expanded memory. При каждом обращении к диску в первом случае будет происходить временный переход в защищенный режим, а во втором случае для пересылки данных будут использоваться функции EMS, описанные в предыдущем разделе.

Для установки виртуального диска в конец файла config.sys надо записать следующую команду:

```
device = c:\dos\ramdrive.sys 8192 /a.
```

В данном случае предполагается, что файл ramdrive.sys хранится на диске C: в каталоге DOS, а для размещения виртуального диска выделяется 8192 байта в Expanded memory.

Если ваша графическая задача будет использовать виртуальный диск для выборки и временного хранения данных, то это ускорит ее работу. Только

не забывайте, что содержимое виртуального диска теряется при выключении или перезагрузке ПК.

Заключение. При создании задач, предназначенных для выполнения в реальном режиме работы микропроцессора, дополнительную память ПК имеет смысл использовать как `Expanded memory`. Функции `EMS` позволяют прикладным задачам распоряжаться пространством расширенной памяти без существенных ограничений. Работа с `Extended memory` применяется, если задача выполняется в защищенном режиме. В таком случае использовать функции `EMS` не целесообразно, поскольку задаче доступно все свободное пространство оперативной памяти без каких-либо ограничений.

ПРИЛОЖЕНИЕ В

Оформление подпрограмм

Использование подпрограмм (`subroutine`) или процедур (`procedure`) является одним из универсальных приемов программирования. Возможность работы с ними предусмотрена во всех языках программирования. Изначально идея заключалась в следующем: неоднократно выполняемые действия оформляются в виде самостоятельного фрагмента программы так, чтобы к нему можно было обратиться из любой ее точки и затем вернуться назад. Со временем эта идея развилась, появилась категория процедур, текст которых не описывается в программе, а готовится заранее, хранится в специальных библиотеках и доступен для любых программ. В комплект компиляторов с алгоритмических языков обычно включены библиотеки, содержащие процедуры различного назначения, в том числе и для работы с новым периферийным оборудованием.

В настоящем приложении описана техника составления процедур при программировании на Макроассемблере. Особое внимание уделено рассмотрению условий, при которых такие процедуры могут использоваться в программах, составленных на языках высокого уровня Си, Паскаль, Фортран и др. Именно ради этого данное приложение включено в текст книги.

В.1. Классификация подпрограмм

При работе с Макроассемблером подпрограммы (процедуры) делятся на ближние и дальние, внутренние и внешние. Два первых термина характеризуют способ вызова подпрограммы и возврата из нее, а два вторых — локализацию подпрограмм по отношению к тексту задачи.

Ближние подпрограммы. При входе в ближнюю (`near`) подпрограмму и при возврате из нее текущее содержимое сегментного регистра `CS` не изменяется. Это означает, что вызов ближней подпрограммы возможен только из того сегмента, в котором она описана. Все подпрограммы, приведенные в при-

мерах основной части книги и ее приложений, являются ближними, поэтому они могут располагаться только в разделе кодов задачи.

От простой группы команд ближняя подпрограмма отличается только тем, что первая команда обязательно имеет метку (имя), а последней *выполняемой* командой является `ret` или `retn` (это два разных имени одной инструкции). Она (`ret`) просто выталкивает содержимое верхушки стека в счетчик команд (регистр `IP`) и увеличивает адрес указателя стека на 2, т. е. равноценна команде `pop ip`. В результате происходит возврат на вызывающий модуль.

Выражение "последняя выполняемая команда" надо понимать буквально, `ret` завершает не текст подпрограммы, а ее выполнение. В простых случаях она может завершать текст подпрограммы, но если последняя имеет разветвления, то каждая ветвь может заканчиваться командой `ret`.

Обращение к подпрограмме (ее вызов) выполняет специальная команда `call`, содержащая адрес точки входа. Для его указания можно использовать все стандартные способы адресации, например, имя точки входа, явное задание адреса, выбор адреса из регистра и т. д. Команда `call` помещает в стек адрес возврата и выполняет безусловный переход на указанную точку входа. Адресом возврата является текущее содержимое счетчика команд (`IP`). После выборки кода инструкции и операндов `IP` всегда содержит адрес начала следующей команды. Таким образом, специальная команда `call` нужна для того, чтобы сформировать в стеке адрес возврата для команды `ret`, завершающей выполнение подпрограммы.

Подпрограмма может работать, и обычно работает, со стеком. Причем к моменту выполнения команды `ret` в верхушке стека должен находиться адрес возврата. Сказанное не означает, что его нельзя изменять. Это делается в особых случаях, когда по каким-то причинам надо вернуться не на вызывающий модуль, а в любое другое место задачи.

Дальние подпрограммы. Дальняя (`far`) подпрограмма отличается от ближней тем, что она расположена не в том сегменте, в котором находится вызывающий модуль. Поэтому при обращении к удаленной подпрограмме изменяется содержимое не только счетчика команд (`IP`), но и кодового сегментного регистра (`CS`).

Мнемоническим именем команды вызова в любом случае является `call`, но ему могут соответствовать разные коды операций (машинных инструкций). Обнаружив в тексте команду `call`, Макроассемблер анализирует описание указанного в ней имени, и в зависимости от его типа (`far` или `near`) выбирает нужный код операции вызова подпрограммы. В частности, если имя соответствует удаленной процедуре, то будет выбран код операции, при выполнении которого в стек записывается сначала содержимое сегментного регистра `CS`, а затем счетчика команд `IP`. Таким образом, при входе в дальнюю подпрограмму в верхушке стека находится исходное значение `IP`, а перед ним — значение `CS`.

Последней выполняемой командой дальней подпрограммы является `retf`, она выталкивает из верхушки стека не одно, а два слова. Первое слово выталкивается в счетчик команд `IP`, а второе — в сегментный регистр `CS`. В результате в регистрах `CS:IP` оказывается полный адрес точки возврата.

Сегмент, содержащий дальнюю подпрограмму, может входить в тело задачи или располагаться во внешнем модуле. Во втором случае подпрограмма оказывается доступной не для одной, а для разных задач.

Описание подпрограмм. Для оформления подпрограмм предназначены две директивы `PROC` и `ENDP`. Первая объявляет начало блока подпрограммы, а вторая — его конец. Перед обеими директивами указывается одно и то же имя, которое является именем точки входа в подпрограмму.

Упрощенная форма директивы `PROC` имеет следующий вид:

```
имя_подпрограммы PROC far или near
```

Обратите внимание на отсутствие символа "двоеточие" после имени подпрограммы. Слова `far` или `near` задают тип процедуры, т. е. характеризуют ее удаленность от точки вызова.

Явное описание процедуры с помощью указанных директив упрощает работу программиста. При обработке директивы `PROC` Макроассемблер помещает в свои рабочие таблицы имя и тип подпрограммы. Теперь, обнаружив в вызывающем модуле команду `call`, он по имени процедуры сам определит соответствующий ей код операции.

Кроме того, при компиляции блока подпрограммы, обнаружив в тексте команду `ret`, Макроассемблер выберет ее код (`retn` или `retf`) для корректного возврата в вызывающий модуль.

Упрощенная форма директивы `PROC` применима при работе с любой версией Макроассемблера, начиная с 5.1. В последующих версиях MASM появилась расширенная форма директивы `PROC` (см. раздел В.5).

Дополнительные точки входа. В зависимости от конкретного назначения подпрограмма может иметь не одну, а несколько точек входа. Для описания дополнительных точек входа в процедуры применяется специальная директива:

```
name LABEL far или near
```

Здесь `name` соответствует имени точки входа, а `far` или `near` указывает ее удаленность от точки вызова. Данная директива просто описывает удаленную метку, независимо от ее конкретного назначения. Если она является точкой входа в подпрограмму, то для вызова используется команда `call name`. А если это продолжение программы, расположенное в другом сегменте, то переход на него выполняет команда `jmp name`.

Пример описания подпрограмм. Для работы с окнами видеопамяти в основной части книги неоднократно использовались процедуры `NxtWin`, `SetWin`

и PrevWin, их исходный текст приведен в примере 2.8. Покажем (см. пример В.1), что изменится в этом тексте, если процедуры явно описать как удаленные.

Пример В.1. Три подпрограммы для работы с видеоокнами

```
NxtWin   PROC  far            ; описание процедуры NxtWin
         push  ax             ; сохраняем содержимое ax
         mov   ax, GrUnit     ; читаем единицу приращения окна
         add   Cur_win, ax    ; увеличиваем номер окна
         pop   ax             ; восстанавливаем содержимое ax
SetWin   LABEL far            ; точка входа в процедуру SetWin
@@:      PushReg <ax,bx,dx>   ; сохранение содержимого регистров
         xor   bx, bx         ; признак установки окна
         mov   dx, Cur_win    ; номер устанавливаемого окна
         call  [VMC]          ; обращение к подпрограмме BIOS
         PopReg <dx,bx,ax>    ; восстановление содержимого регистров
         ret                  ; возврат в вызывающий модуль
PrevWin  LABEL far            ; точка входа в процедуру PrevWin
         push  ax             ; сохранение содержимого ax
         mov   ax, GrUnit     ; читаем единицу приращения окна
         sub   Cur_win, ax    ; уменьшаем номер окна
         pop   ax             ; восстанавливаем содержимое ax
         jmp   SHORT @B       ; переход на установку окна
NxtWin   ENDP                 ; конец процедуры NxtWin
```

По сравнению с исходным текстом в примере В.1 добавились директивы, описывающие блок процедуры NxtWin, и две дополнительные точки входа SetWin и PrevWin. Кроме того, введена локальная метка @@, переход на нее выполняет команда jmp SHORT @B. В оригинале ей соответствовала команда jmp SHORT SetWin. В данном случае метка SetWin описана как удаленная, короткий переход на нее не возможен, поэтому введена локальная метка.

Замечание

Подпрограммы примера В.1 еще нельзя использовать для работы. Предварительно их надо оформить в виде программного модуля и объявить общедоступными. Как это делается, описано в следующем разделе.

Внешние и внутренние переменные. Все переменные, описанные в тексте конкретной программы (далее — в модуле), являются внутренними или локальными. Для того чтобы некоторые из них или все стали общедоступными, их имена надо перечислить в списке следующей директивы:

```
PUBLIC name[[, name]]
```

Такая форма записи означает, что в директиве может быть указано столько имен, сколько поместится в строке, и что они отделяются друг от друга за-

пятыми. После последнего символа запятая недопустима, для удобства чтения после запятых лучше делать пробел. Директиву можно повторять столько раз, сколько требуется для перечисления всех имен. В теле программы она обычно располагается перед описанием первого сегмента.

При обработке директивы Макроассемблер определяет типы имен по их описаниям в тексте программы и помещает список имен с указанием типов в объектный модуль. Эти данные нужны компоновщику (`link.exe`), они имеют специальное назначение и не влияют на размер будущей задачи. Если вы посмотрите листинг файла, который может формировать Макроассемблер, то увидите, что таким именам присвоена характеристика `Global`.

Для того чтобы имена, объявленные общедоступными или глобальными, можно было использовать в другом модуле, их надо описать в нем как внешние с помощью следующей директивы:

```
EXTERN name:type[[, name:type]]
```

В программном модуле данная директива обязательно располагается перед описанием первого сегмента. Если список внешних имен большой, то директива повторяется нужное число раз. Тип зависит от назначения имени. Переменные могут иметь типы `byte`, `word`, `dword` и т. д. Метки и имена подпрограмм имеют тип `far`.

Макроассемблер заносит перечисленные имена и их типы в таблицы, которые он создает в процессе компиляции, и при каждом обнаружении любого из имен проверяет соответствие применения имени его типу. Как обычно, если использование имени в тексте модуля не соответствует его типу, выводится аварийное сообщение. Правильность указания самих имен Макроассемблер проверить не может.

Внешние имена и их типы нужны компоновщику (`link.exe`), поэтому Макроассемблер помещает их в объектный модуль. Это специальная информация, она не влияет на размер строящейся задачи.

Таким образом, общедоступные имена должны быть описаны в двух модулях — источнике и приемнике. В источнике они перечисляются в директиве `PUBLIC`, а в приемнике — в директиве `EXTERN`. С их описаниями работает компоновщик. Если в одном из модулей встречается внешнее имя, то он ищет его в списках глобальных имен других модулей. При отсутствии соответствующего описания будет выдано сообщение об ошибке.

В.2. Оформление программных модулей

Общедоступные подпрограммы могут располагаться в исходных или в объектных модулях. Исходный модуль состоит из программного сегмента, содержащего описание одной или нескольких подпрограмм на языке Макроассемблера. Объектный модуль получается в результате обработки исходного

модуля компоновщиком. Исходные модули могут располагаться непосредственно в тексте основной программы или храниться в отдельных файлах, в последнем случае они являются общедоступными. Объектные модули могут храниться в виде отдельных файлов, имеющих тип `obj`, или в библиотечных файлах, имеющих тип `lib`. Они подключаются к задаче только в процессе ее компоновки (сборки). В данном разделе описаны рекомендации, которых следует придерживаться при оформлении обоих типов модулей.

Пример модуля в теле задачи. На первой стадии подготовки исходного модуля производится составление, набор текста и отладка подпрограмм, поэтому модуль удобнее включить в текст основной программы в виде отдельного сегмента, а не хранить в отдельном файле. В примере В.2 показан вариант оформления дополнительного сегмента в основной программе.

Пример В.2. Сегмент с описанием подпрограмм NxtWin, SetWin, PrevWin

```
subr    SEGMENT word public 'subr'  ; начало сегмента
        ASSUME  cs:subr, ds:@data   ; установка соответствия
;       .386                        ; тип микропроцессора
;       Далее располагается текст примера В.1, содержащий
;       описание подпрограмм NxtWin, SetWin и PrevWin
subr    ENDS                        ; конец сегмента
```

Первая директива примера В.2 открывает описание сегмента. В данном случае его параметры можно было не указывать, они приведены просто для иллюстрации. Параметр `word` обозначает, что сегмент располагается в памяти, начиная с четного адреса. Параметр `public` является признаком общедоступного сегмента. Заключенное в кавычки название сегмента передается компоновщику и становится общедоступным.

Замечание

Обратите внимание на то, что между параметрами директивы `SEGMENT` отсутствуют запятые!

Директива `ASSUME` нужна для того, чтобы Макроассемблер мог определить, что будет находиться в сегментных регистрах `cs` и `ds` при выполнении подпрограмм. Без этого невозможна компиляция команд. С регистром `cs` всегда связывается имя сегмента, в котором расположена директива `ASSUME`. С регистром `ds` связывается имя сегмента данных, который описан вне данного модуля. Имя, с которым ассоциируется `ds`, зависит от способа описания сегмента данных в основном тексте программы. В примере В.2 предполагается, что сегмент данных был описан с помощью специальной директивы `.data` (см. раздел Б.1.2).

Если вы забудете указать директиву `ASSUME`, то при компиляции Макроассемблер может выдавать аварийные сообщения, смысл которых заключается

в том, что не определен один из сегментных регистров. Чтобы лучше понять назначение директивы, уберите ее из текста и посмотрите, что из этого получится. В частности, результат зависит от версии MASM.

В примере В.2 строка, содержащая третью директиву, начинается с символа "точка с запятой". Подпрограммы `NxtWin`, `SetWin` и `PrevWin` составлены с использованием набора команд микропроцессора Intel 8086, поэтому в данном случае директива `.386` не нужна. Однако большинство описанных в книге примеров рассчитано на возможности микропроцессора Intel 80386, и для их компиляции данная директива необходима.

Разрядность сегмента. При обработке директив описания сегментов Макроассемблер проверяет установленный тип микропроцессора и выбирает соответствующий режим выполнения команд, расположенных в сегменте (реальный или защищенный). По умолчанию установлен 16-разрядный (реальный) режим выполнения команд и набор инструкций для микропроцессора Intel 8086.

Если директива `.386` предшествует описанию сегмента, то он будет объявлен как 32-разрядный, расположенные в нем команды будут рассчитаны на работу с 32-разрядными адресами. В реальном режиме результаты выполнения таких команд непредсказуемы.

Поэтому при создании программ или подпрограмм, предназначенных для выполнения в реальном режиме работы микропроцессора, директиву `.386` надо располагать после описания сегмента. В таком случае она оказывает влияние только на набор инструкций микропроцессора.

Начиная с версии 6.0, Макроассемблер поддерживает директивы `.486` и `.586`, разрешающие использование новых инструкций микропроцессоров Intel 486 и Pentium. Кроме того, появилась возможность выбора разрядности сегментов по умолчанию или ее явного описания с помощью ключевых слов `USE16` и `USE32`.

В конце файла, формируемого Макроассемблером, приводится описание всех сегментов программного модуля. При компиляции обязательно укажите имя файла и затем проверьте, соответствуют ли указанная в нем разрядность сегментов той, которую вы предполагаете.

Расположение сегмента в тексте программы зависит от версии MASM, который вы используете. Если это MASM 6.0 и выше, то дополнительный сегмент может располагаться как перед основным текстом программы (перед сегментом кодов), так и после него. Но если вы работаете с MASM 5.1, то дополнительный сегмент может располагаться *только перед* сегментом кодов. В противном случае при каждом вызове подпрограмм, расположенных в дополнительном сегменте, MASM 5.1 выводит аварийное сообщение о необходимости предварительного описания подпрограммы.

Подключение исходного модуля. Расположенные в теле задачи модули не являются общедоступными. Поэтому после отладки дополнительный сегмент с описанием подпрограмм удаляется из текста программы и помещается в отдельный файл. Имя, тип и расположение файла на жестком или гибком диске вы можете выбирать по своему усмотрению.

Для включения содержимого файла в нужном месте текста программы указывается специальная директива:

```
INCLUDE спецификация_файла
```

Спецификация должна быть настолько подробной, чтобы Макроассемблер мог найти и прочитать файл. Очень часто эта директива применяется для подключения файлов, содержащих тексты макроопределений. Обычно в установочный комплект MASM включено несколько таких файлов, они могут располагаться в специальном каталоге `INCLUDE`.

В данном случае нас интересуют файлы, содержащие исходные тексты общедоступных подпрограмм. На состав и назначение подпрограмм не налагается никаких специальных ограничений, должны лишь соблюдаться общие правила оформления программных сегментов, а именно:

- каждый сегмент имеет уникальное имя;
- размер кода после компиляции не превышает 65 536 байтов;
- в тексте отсутствуют ошибки, т. е. он должен быть предварительно отлажен.

Таким образом, содержимое включаемого файла становится частью текста программы и компилируется Макроассемблером. Если в задании на компиляцию указан файл листинга, то, просмотрев его, вы увидите полный результат компиляции, в том числе и включенного файла.

Пример объектного модуля. Для получения объектного модуля надо внести изменения в подключаемый модуль и откомпилировать его отдельно от основной задачи.

Изменения заключаются в том, что в начале текста модуля добавлены две директивы:

- `EXTERN` — для описания используемых внешних имен;
- `PUBLIC` — для объявления подпрограмм модуля общедоступными.

Кроме этого, нужен признак конца текста модуля, которым является директива `END`. В примере В.3 показано, что изменится в модуле примера В.2.

Пример В.3. Исходный текст для получения объектного модуля

```
;       Сюда надо вставить макроопределения из примера 2.12
subr    SEGMENT word public 'subr'  ; начало сегмента
        EXTERN                GrUnit:word, Cur_win:word, VMC:dword
```

```
        PUBLIC                  NxtWin, SetWin, PrevWin
        ASSUME  cs:subr         ; установка соответствия
;       .386                    ; тип микропроцессора
;       Далее располагается текст примера B.1, содержащий
;       описание подпрограмм NxtWin, SetWin и PrevWin
subr    ENDS                    ; конец сегмента
        END                     ; конец текста модуля
```

Текст примера В.3 начинается с комментария, напоминающего о том, что перед описанием сегмента надо расположить макроопределения `PushReg` и `PopReg`. Они используются в подпрограмме `SetWin` для сохранения в стеке и последующего восстановления содержимого регистров `ax`, `bx` и `dx` (см. пример В.1). Можно отказаться от их включения, заменив первый макровызов тремя командами `push`, а второй тремя командами `pop`. Раньше мы об этом не говорили, поскольку модуль предназначался для совместной компиляции с текстом программы, в котором описаны указанные макроопределения.

В директиве `EXTERN` перечислены имена переменных `GrUnit`, `Cur_win` и `VMC`, которые описаны в сегменте данных основной программы. Назначение и способ определения значений этих переменных подробно обсуждались в главе 2, а их описание показано в примере 2.11 той же главы.

Следующая директива `PUBLIC` объявляет имена подпрограмм `NxtWin`, `SetWin` и `PrevWin` общедоступными.

Обратите внимание на то, что в директиве `ASSUME` описан только кодовый сегмент, а если вы работаете с MASM 6.0 или более поздней версией, то эту директиву можно вообще исключить.

Далее в модуле должно располагаться описание подпрограмм, текст которого приведен в примере В.1, а после него директива `END` без указания метки, поскольку модуль не является выполняемой задачей.

Текст примера В.3 не может быть использован для включения в основную программу с помощью директивы `INCLUDE`. Он компилируется отдельно. При этом Макроассемблер формирует объектный модуль, который понадобится компоновщику при построении задачи.

Для компоновки нужен еще один объектный модуль. Он получается при компиляции основного текста задачи. Если в основном тексте описаны подпрограммы для работы с окнами видеопамяти, то их оттуда надо удалить. Кроме того, в основной текст надо включить следующие две директивы:

```
PUBLIC GrUnit, Cur_win, VMC
EXTERN NxtWin:far, SetWin:far, PrevWin:far
```

Первая из них объявляет переменные `GrUnit`, `Cur_win` и `VMC` общедоступными, а вторая описывает имена и типы внешних подпрограмм. После вклю-

чения указанных директив основной текст задачи компилируется для получения объектного модуля.

Построение задачи. Условимся считать, что файл, содержащий объектный модуль основного текста будущей задачи, имеет имя `bmpsuper.obj`, а файл, содержащий объектные модули подпрограмм, имя `bmpsub.obj`. Для их объединения в одну задачу выполняется следующая команда:

```
link bmpsuper bmpsub  или  link bmpsuper+bmpsub
```

В данном случае предполагается, что файлы `bmpsuper` и `bmpsub` имеют тип `obj` и расположены в том же каталоге, в котором находится задача `link.exe` (компоновщик). Если это не так, то указывается спецификация, позволяющая найти файлы в других каталогах.

Важно

Имена объединяемых файлов может разделять либо пробел, либо знак "плюс".

Если между именами поставить запятую, то компоновщик будет обрабатывать два файла независимо друг от друга, т. е. он попытается построить две разные задачи. Разумеется, это приведет к ошибке, поскольку в каждом из файлов будут обнаружены неопределенные внешние имена.

Первым в списке должен располагаться файл, содержащий основной текст задачи, из которого вызываются подпрограммы, описанные в последующих файлах. Если имя строящейся задачи явно не указано, то ей будет присвоено имя первого файла и тип `exe`. В нашем случае имя задачи `bmpsuper.exe`. Если компоновщик не обнаружил ошибок, то задачу можно выполнять.

Заключение. При программировании на ассемблере можно использовать подпрограммы, хранящиеся либо в исходных модулях, подключаемых во время компиляции основного текста, либо в виде объектных модулей, объединяемых с главным модулем при построении задачи. Чему отдать предпочтение, решать вам.

Однако если основной модуль создается на одном из алгоритмических языков (Фортран, Паскаль, Си и пр.), то вспомогательные подпрограммы, составленные на ассемблере, должны оформляться в виде объектных модулей.

И последний совет. Постепенно у вас накопится достаточно много объектных модулей. Для упрощения собственного труда их лучше объединить в одну или несколько библиотек. В комплект поставки Макроассемблера обязательно входит библиотекарь, хранящийся в файле `lib.exe`. Он выполняет много полезных функций, связанных с созданием, просмотром, пополнением и изменением библиотек объектных модулей. Библиотекарь поддерживает активный диалог с оператором, поэтому научиться работать с ним несложно.

В.3. Параметры в стеке

Характерной особенностью подпрограмм является то, что используемые при вычислениях величины передаются им в виде *входных параметров*. В свою очередь, подпрограммы могут возвращать результаты вычислений в виде *выходных параметров*. Способ доступа к параметрам зависит от того, где они расположены.

Наиболее простой формой является передача параметров в регистрах общего назначения. Она используется процедурами BIOS и DOS. В этом случае доступ к параметрам осуществляется наиболее просто и быстро, но существует ряд ограничений, сводящих на нет это преимущество. Здесь для нас существенно одно из таких ограничений, а именно то, что компиляторы с алгоритмических языков обычно не используют регистры для передачи параметров. В некоторых из них, например в Си++, предусмотрена возможность размещения параметров в регистрах, но это уже относится к категории трюков или уловок, но не к стандартам программирования.

Если создаваемая вами подпрограмма предназначена для использования в программах, составленных на одном из алгоритмических языков, то она должна быть рассчитана на расположение параметров в стеке, поскольку в большинстве случаев компиляторы размещают их именно в нем. К описанию особенностей компиляторов мы вернемся после обсуждения техники передачи параметров при программировании на ассемблере.

Общие сведения. Стеком называется любой произвольно выбранный блок оперативной памяти, работа с которым производится по принципу "последнее записанное — первое считанное" (LIFO – last in first out). Иначе говоря, выражение "стек" характеризует не тип памяти, а способ работы с ней.

В процессе выполнения задачи в регистре SS хранится код сегмента оперативной памяти, в котором расположена область стека, а текущий адрес (смещение) верхушки стека в этом сегменте хранится в регистре SP, который называется указателем стека. Разрядность регистров зависит от режима работы микропроцессора: в реальном режиме они содержат по 16 разрядов, а в защищенном режиме по 32 разряда.

Стек нарастает в сторону уменьшения адресов, поэтому при входе в задачу регистр sp содержит наибольший доступный адрес в области стека. При записи данных в стек адрес, хранящийся в sp, уменьшается, а при их выборке из стека — увеличивается.

Для записи и чтения данных в режиме LIFO предназначены команды PUSH и POP, которые неоднократно использовались в примерах. Команда push предварительно уменьшает содержимое регистра sp, а затем записывает операнд в вычисленный адрес. Команда pop, наоборот, сначала считывает операнд, а затем увеличивает содержимое sp. В обоих случаях адрес, хранящийся в sp,

изменяется на размер операнда, который может составлять 2 или 4 байта. Один байт записать в стек нельзя, команда push преобразует его в слово.

Кроме команд push и pop в режиме LIFO со стеком работают команды вызова обычных (call) и прерывающих (int) подпрограмм и команды возврата из них (ret и iret).

Для непосредственного доступа к области стека выделен специальный регистр BP. Он может использоваться в обычных командах (пересылки, сложения и пр.) в тех случаях, когда один их операндов расположен в области стека. Если операнд находится в регистре bp, то при его обработке микропроцессор, по умолчанию, выбирает в качестве сегментного регистра SS, а не DS, как обычно. В случае необходимости можно явно указать любой другой сегментный регистр.

Новое макроопределение. Перед обращением к подпрограмме в стек записываются ее параметры. Запись в стек обычно выполняет команда push. Для сокращения текста программы и придания ему большей наглядности можно использовать специальный макровызов. Текст соответствующего макроопределения приведен в примере В.4.

Пример В.4. Макроопределение для вызова подпрограмм

```
@Invoke  macro name, par  ; заголовок макроопределения
         irp   r, <par>   ; начало оператора повторения
         push  r          ; заготовка повторяемой команды
         endm              ; конец оператора повторения
         call  name       ; заготовка команды вызова подпрограммы
endm                       ; конец текста макроопределения
```

В приведенных ранее примерах неоднократно использовался макровызов PushReg, текст его макроопределения приведен в примере 2.12. В данном случае к этому тексту добавилась только одна строка, содержащая заготовку команды call. Поэтому в результате макроподстановки в текст программы сначала будет включена группа команд push, а затем команда call.

Если макроопределение примера В.4 включено в текст основной задачи, то для его использования в нужном месте указывается следующий макровызов:

@Invoke имя_процедуры <список_параметров>

Имя процедуры может быть как внешним, так и внутренним. Список параметров *обязательно* заключается в угловые скобки, а параметры отделяются друг от друга запятыми, после которых допустим пробел. Форма записи параметров стандартная для команды push. Пример использования макровызова приведен в конце данного раздела.

Доступ процедур к параметрам. При входе в процедуру в верхушке стека расположен адрес возврата, а перед ним параметры. Для работы с парамет-

рами входящие в тело процедуры команды должны иметь прямой доступ к области стека. Как было сказано выше, для этой цели удобно использовать регистр `BP`, но при входе в подпрограмму его содержимое не определено. Поэтому в начале подпрограммы надо сохранить исходное содержимое `bp` и записать в него опорный (базовый) адрес, которым является текущее значение указателя стека. Сохранение содержимого `bp` нужно для того, чтобы выполнение данной подпрограммы не влияло на выполнение вызывающего модуля. При выходе из подпрограммы перед выполнением команды `ret` сохраненное значение надо вытолкнуть в `bp`.

Таким образом, большинство подпрограмм, ориентированных на работу со стеком, начинается с двух следующих команд:

```
push    bp        ; сохранение исходного содержимого bp
mov     bp, sp    ; запись в bp адреса верхушки стека
```

Давайте уточним, что находится в стеке после выполнения этих команд. Для определенности будем считать, что вызвана внешняя подпрограмма, имеющая два параметра, каждый из которых занимает одно слово. Микропроцессор работает в реальном режиме, поэтому адрес возврата занимает два слова. В таком случае стек содержит величины, приведенные в табл. В.1.

Таблица В.1. *Вариант размещения данных в стеке*

Смещение	Что находится в слове
`bp + 0`	Исходное содержимое регистра `bp`
`bp + 2`	Младшая часть адреса возврата (`IP`)
`bp + 4`	Старшая часть адреса возврата (`CS`)
`bp + 6`	Второй параметр подпрограммы
`bp + 8`	Первый параметр подпрограммы

В соответствии с табл. В.1, при сделанных выше допущениях, полный адрес первого параметра равен `ss:[bp+8]`, а второго — `ss:[bp+6]`. Как уже говорилось, сегментный регистр `ss` в записи операндов не указывается, поскольку в данном случае он используется по умолчанию. Например, произведение параметров можно вычислить с помощью двух команд:

```
mov     ax, [bp+6]   ; ax = значение первого параметра
mul     [bp+8]       ; dx:ax = ax * значение второго параметра
```

В общем случае адрес параметра, т. е. значение, прибавляемое к `bp`, зависит от размеров предшествующих величин и его собственного размера. Поэтому при разработке подпрограммы надо предварительно определить, что конкретно будет находиться в стеке при ее вызове.

Внешняя процедура cnvindec. Рассмотрим простую внешнюю процедуру, которая преобразует последовательность десятичных цифр, представленных в коде ASCII, в код десятичного числа. Цифры в коде ASCII получаются, например, при вводе чисел с клавиатуры. Для того чтобы результат ввода можно было использовать при вычислениях, последовательность цифр надо преобразовать в шестнадцатеричный код числа.

Алгоритм формирования десятичного числа следующий. Обозначим формируемое число как result и предположим, что в исходном состоянии result = 0. В таком случае на шаге номер I значение result умножается на 10 и к произведению прибавляется код очередной цифры:

```
result = result * 10 + digit [I]
```

Перед прибавлением кода очередной цифры его надо преобразовать в двоичный код. В формате ASCII коды цифр изменяются от 30h до 39h, поэтому для преобразования из кода цифры вычитается код нуля (30h). Кроме того, надо проверить, действительно ли очередной символ строки является цифрой, и если это не так, то процесс формирования числа прекращается.

Завершенный текст подпрограммы приведен в примере В.5. Перед ее вызовом в стеке указывается полный адрес преобразуемой строки (сегмент и смещение). Сформированное число помещается в стек на место адреса строки. Кроме того, при возврате из подпрограммы в регистре al находится код символа, при обнаружении которого было прекращено формирование результата. Им может быть любой символ, кроме цифры. Вариант обращения к подпрограмме описан ниже.

Пример В.5. Исходный текст процедуры cnvindec

```
        PUBLIC cnvindec     ; объявляем процедуру общедоступной
subr    SEGMENT word public 'subr'; начало сегмента subr
        ASSUME cs:subr      ; cs ассоциируется с subr
        .386                ; задаем тип микропроцессора
dten    dd   10             ; константа для умножения на 10
cnvindecPROC far            ; начало блока процедуры
        push bp             ; сохранение содержимого bp
        mov  bp, sp         ; bp = sp базовый адрес в стеке
        push edx            ; сохраняем содержимое edx
        push fs             ; сохраняем содержимое fs
        push si             ; сохраняем содержимое si
        lfs  si, [bp+6]     ; fs:si = адрес начала строки текста
        mov  dword ptr [bp+6], 0; result = 0 очистка результата
cnvloop:xor  eax, eax       ; очистка eax
        lods byte ptr fs:[si]; al = очередной символ строки
        cmp  al, '0'        ; код символа меньше кода цифры 0 ?
        jb   endcnv         ; -> да, конец формирования числа
```

```
        cmp   al, '9'        ; код символа больше кода цифры 9 ?
        ja    endcnv         ; -> да, конец формирования числа
        sub   al, 30h        ; вычитаем код цифры 0
        xchg  eax, [bp+6]    ; переставляем eax и result
        mul   cs:dten        ; edx:eax = result * 10
        add   [bp+6], eax    ; result = result + eax
        jmp   short cnvloop  ; -> на начало цикла преобразования
endcnv: pop   si             ; восстанавливаем содержимое si
        pop   fs             ; восстанавливаем содержимое fs
        pop   edx            ; восстанавливаем содержимое edx
        pop   bp             ; восстанавливаем содержимое bp
        ret                  ; возврат из подпрограммы
cnvindec ENDP                ; конец блока процедуры
subr    ENDS                 ; конец сегмента subr
        END                  ; конец текста модуля
```

Подпрограмма примера В.5 оформлена в виде готового для компиляции модуля. Способ оформления такого модуля описан в предыдущем разделе и показан в примере В.3. Поэтому мы начнем с основного текста.

В сегменте `subr` перед текстом процедуры описано двойное слово `dten` и ему присвоено значение 10. Эта переменная используется в процедуре при умножении, она нужна потому, что операндом команды `mul` не может быть константа 10.

Процедура преобразования имеет имя `cnvindec`. Ее текст начинается с подготовки регистра `bp` и сохранения в стеке используемых регистров. После этого в регистры `fs:si` загружается адрес преобразуемой строки.

Важно

Перед вызовом процедуры в стек *сначала* записывается *сегмент*, а затем смещение строки. Только при выполнении этого условия команда `lfs` поместит в регистр `fs` код сегмента, а в регистр `si` — смещение.

После загрузки адреса строки в регистры параметры не нужны и отведенное для них место используется для размещения формируемого числа. Предварительно команда `mov dword ptr [bp+6], 0` очищает два слова стека с адресами `[bp+6]` и `[bp+8]`.

Цикл формирования числа начинается с команды, имеющей метку `cnvloop`, и заканчивается командой `jmp short cnvloop`. Код формируемого числа может содержать до 32-х разрядов, поэтому вычислительные операции выполняются с операндами, имеющими размер двойного слова.

Цикл начинается с очистки регистра `eax` и записи в его младший байт (`al`) кода очередного символа. Затем проверяется, чему соответствует этот код. Если он соответствует цифре, то из содержимого `al` вычитается код цифры 0, производится перестановка содержимого `eax` и `result` и выполняется

умножение `result * 10`. В связи с тем, что константа `dten` расположена в кодовом сегменте, в команде `mul` перед ней явно указано имя регистра `cs`. Младшая часть результата умножения (содержимое `eax`) прибавляется к `result` и происходит возврат на начало цикла формирования числа.

Если очередной символ не является цифрой, то выполнение цикла прекращается и происходит переход на метку `endcnv`. Начиная с этой метки расположены команды, восстанавливающие содержимое сохраненных в стеке регистров, и команда `retf`, завершающая выполнение подпрограммы.

Замечание

Значение формируемого подпрограммой примера В.5 числа может изменяться в пределах от 0 до 4294967295 (2^{32} — 1). Контроль переполнения результата отсутствует, это сделано для упрощения текста подпрограммы. При необходимости вы можете ввести такой контроль или увеличить диапазон допустимых значений числа. Напомним, что при умножении на 10 старшая часть произведения находится в регистре `edx`, но подпрограмма не работает с этим регистром.

Использование процедуры cnvindec. Для использования в задачах модуль примера В.5 компилируется, и полученный объектный модуль объединяется с объектным модулем основной задачи. Как это делается, описано в разделе В.2. Здесь нас будет интересовать вызов подпрограммы задачей.

Предположим, что в разделе данных задачи зарезервирован буфер для размещения вводимой строки текста и ему присвоено имя `linbuf`. Для хранения сформированного числа в разделе данных надо зарезервировать двойное слово, присвоив ему подходящее имя, например, `argument`.

В задаче можно выбрать любой удобный для вас способ ввода текста строки с клавиатуры. Только не забывайте, что вводимые символы нужно отображать на экране монитора, а способ отображения зависит от установленного задачей видеорежима. В текстовых видеорежимах обычно применяется стандартная функция DOS, ее код `0Ah`. Подпрограммы для ввода текста в графических видеорежимах описаны в главе 5, раздел 5.2.5, а их использование показано в приложении А, пример А.1 .

После ввода строки, для формирования числа в основном тексте задачи выполняются следующие команды:

```
@Invoke       cnvindec <ds, offset linbuf>; вызов подпрограммы
pop           argument      ; запись сформированного числа
```

При компиляции, обнаружив имя `@Invoke`, Макроассемблер ищет его в своих таблицах, и если описанное в примере В.4 макроопределение включено в текст задачи, то макровызов преобразуется в следующие три команды:

```
push      ds              ; запись в стек содержимого ds
push      offset linbuf   ; запись в стек смещения linbuf
call      cnvindec        ; обращение к подпрограмме
```

После возврата из подпрограммы в стеке находится результат в виде двойного слова, которое задача должна вытолкнуть в специально выделенную переменную `argument`. Кроме того, в регистре `al` находится код символа, обнаружив который подпрограмма завершила свою работу. В случае необходимости в задаче можно предусмотреть проверку кода этого символа и выполнение тех или иных действий в зависимости от результата проверки.

Таким образом, мы описали простой пример подпрограммы, параметры которой находятся в стеке, и теперь можно перейти к обсуждению общих вопросов, связанных с использованием стека в подпрограммах.

В.4. Работа процедур со стеком

В данном разделе описаны правила, которых следует придерживаться при составлении внешних подпрограмм, ориентированных на работу со стеком.

Распределение пространства стека. Общий случай распределения пространства стека при выполнении процедуры показан в табл. В.2.

Таблица В.2. *Распределение пространства стека в порядке увеличения адресов*

Общедоступная область стека
Промежуточные переменные подпрограммы
Исходное содержимое регистра `bp` или `ebp`
Адрес возврата из подпрограммы
Параметры подпрограммы
Недоступная часть стека

Общедоступная область расположена в начале стекового сегмента, ее минимальный адрес (смещение) равен нулю, а максимальный хранится в указателе стека (в регистре `sp`). Обычно она используется для хранения содержимого регистров и передачи параметров вызываемым подпрограммам.

Место для промежуточных переменных резервирует подпрограмма, если в этом есть необходимость. Она же сохраняет в стеке исходное содержимое регистра `bp` или `ebp` при работе в 32-разрядном режиме. Во время выполнения подпрограммы адрес, в котором сохранено исходное значение регистра `bp` (или `ebp`), используется в качестве базы для доступа команд к параметрам или промежуточным переменным.

Адрес возврата и параметры размещает в стеке основная задача, вызывающая данную подпрограмму. При входе в подпрограмму указатель стека содержит адрес первого свободного слова, в которое обычно помещается исходное значение регистра `bp` или `ebp`.

Недоступная часть стека названа так не потому, что она физически недоступна, а потому, что подпрограмма не должна изменять хранящиеся там данные. Если перед размещением параметров стек был полностью очищен, то недоступной области просто не существует.

Промежуточные переменные. В зависимости от конкретных особенностей подпрограммы при ее выполнении может возникнуть необходимость в использовании переменных для хранения промежуточных результатов вычислений в оперативной памяти. Назовем такие переменные "промежуточными".

До сих пор, в приводимых примерах переменные располагались в разделе данных задачи. Только в примере В.5 переменная `dten` хранится в сегменте кодов. Как правило, у внешних подпрограмм нет собственного сегмента данных, исключения возможны, но они встречаются редко. Размещать же промежуточные результаты в сегменте данных основной задачи не целесообразно, поскольку они используются только во время выполнения подпрограммы и не нужны в других случаях.

Пространство для размещения промежуточных переменных лучше всего выделять в стеке при входе в подпрограмму и освобождать его перед выходом из нее. Для резервирования требуемого пространства после сохранения содержимого регистра `bp` надо просто уменьшить текущее значение указателя стека на суммарный размер промежуточных переменных, выраженный в байтах. В примере В.6 показано, как это обычно делается.

Пример В.6. Варианты оформления начала подпрограммы

```
;     Вариант 1 — использование трех команд
      push   bp        ; сохранение содержимого bp
      mov    bp, sp    ; запись в bp адреса верхушки стека
      sub    sp, N     ; резервирование N байтов в стеке
;     Вариант 2 — специальная команда enter
      enter  N, 0      ; заменяет три команды варианта 1
```

В первом варианте примера В.6 показано, как резервируется пространство размером `N` байтов с помощью обычных команд. Начиная с модели Intel 80286, у микропроцессоров появилась специальная команда `enter`. Она сохраняет в стеке содержимое регистра `bp`, копирует в `bp` адрес верхушки стека и уменьшает на `N` содержимое `sp`, т. е. по результату эквивалентна трем командам варианта 1. При использовании во внешних процедурах второй параметр команды `enter` равен нулю.

Важно

Ни при каких обстоятельствах значение указателя стека не может быть нечетным числом. Поэтому количество байтов, отводимых для размещения промежуточных переменных, обязательно должно быть четным. Однако это не означает, что в пространстве стека нельзя размещать байты и работать с ними.

После выполнения любого из вариантов примера В.6 регистр `bp` используется для прямого доступа к параметрам и промежуточным переменным. Параметры расположены выше, а промежуточные переменные — ниже находящейся в `bp` точки отсчета. Обозначим смещение параметра или переменной как `XX`. В таком случае, при обращении к параметрам содержимое в `bp` увеличивается на величину смещения (`[bp+XX]`), а при обращении к промежуточным переменным оно уменьшается на величину смещения (`[bp-XX]`).

Прежде чем использовать переменные в командах, надо вычислить смещение каждой из них относительно регистра `bp`. Оно зависит от размеров предыдущих и данной переменной и не может быть равно нулю. Например, первая по порядку переменная может быть байтом, словом или двойным словом, ее адрес будет соответственно равен `[bp-1]`, `[bp-2]` или `[bp-4]`.

Имена параметров и переменных. Запись адреса в явном виде вполне корректна, но не наглядна. Для того чтобы текст подпрограммы был более понятен, при визуальном анализе лучше использовать имена.

Особенность данного случая в том, что переменные и параметры распределяются не статическим, а динамическим способом. Это не позволяет использовать для их описания обычные директивы `db`, `dw` и пр. Вместо этого используется директива `EQU` (эквивалентно). Перед ней указывается имя параметра или переменной, а после нее описание адреса и размера.

Предположим, что внешней подпрограмме передаются два параметра, каждый из которых имеет размер слова. Один их них задает ширину строки на экране в точках, а второй — количество строк на экране. В главе 2 для обозначения этих величин были введены имена `Horsize` и `Versize`. Для присвоения этих имен параметрам в текст модуля подпрограммы надо включить две следующие директивы:

```
Horsize  EQU   word ptr [bp + 6]   ; количество точек в строке
Versize  EQU   word ptr [bp + 8]   ; количество строк на экране
```

В этом примере предполагается, что перед вызовом внешней процедуры в стек сначала было записано значение параметра `Versize`, а затем `Horsize`, например, так:

```
@Invoke имя_подпрограммы <Versize, Horsize>
```

Только в таком случае описание параметров соответствует их реальному расположению в стеке.

Обычно директивы `EQU` размещаются в начале тела модуля, вне сегмента (или вне сегментов). Имена, присвоенные в подпрограмме, являются локальными, поэтому вполне допустимо их совпадение с именами, описанными в основной программе или в других подпрограммах.

При указанном описании параметров для вычисления количества точек на экране в тексте подпрограммы выполняются следующие две команды:

```
mov     ax, Horsize      ; ax = количество точек в строке
mul     Versize          ; dx:ax = Horsize * Versize
```

Описание промежуточных переменных отличается от описания параметров только указанием отрицательного смещения относительно bp, пример:

```
Address  EQU   word ptr [bp – 2]   ; описание переменной Address
```

Следует отметить, что последовательность директив EQU является своеобразным описанием формальных параметров. Если такие директивы включены в текст модуля, то по его распечатке можно определить тип и последовательность указания параметров при вызове процедуры.

Контроль пространства стека. Контроль состояния стека нужен в тех случаях, когда задача использует вложенные процедуры и уровень вложенности достаточно велик. Вложенными называются процедуры последовательно вызывающие друг друга. При этом пространство стека, доступное каждой последующей процедуре, сокращается. При неудачном стечении обстоятельств оно может быть просто исчерпано. Особенно активно используют стек рекурсивные процедуры, способные многократно вызывать самих себя, правда, в графических задачах они обычно не используются.

Если проверка свободного пространства стека предусмотрена, то она выполняется в начале процедуры. При корректной работе со стеком размер свободного пространства в байтах равен значению указателя стека (содержимому регистра sp). Для выполнения контроля надо сравнить текущее содержимое sp с той величиной, которая нужна для выполнения процедуры. Если содержимое sp больше требуемого значения, то процедура может быть выполнена, в противном случае обычно выводится аварийное сообщение и выполнение задачи прекращается.

Процедура может использовать стек для размещения промежуточных переменных, хранения содержимого регистров и для вызова вложенных процедур. Определить требуемый размер пространства в стеке можно только на основании анализа исходного текста конкретной процедуры. Это должен делать ее разработчик.

Следует отметить, что проверка доступного пространства в стеке не является обязательной. Ее можно использовать на стадии отладки задачи, а затем исключить из подпрограмм. Напоминаем, что размер стекового сегмента указывается при его описании в тексте основной программы, поэтому всегда можно выбрать оптимальное значение.

Очистка стека является обязательным действием, выполняемым перед возвратом из процедуры. На стадии отладки задачи многие ошибки могут быть связаны с некорректными действиями при очистке стека.

В общем случае подпрограмма использует три разные области стека (см. табл. В.2) и их очистка производится разными способами.

Прежде всего, очищается общедоступная часть стека. Для этого количество использованных в подпрограмме команд `push` и `pop` должно совпадать.

Следующим шагом является освобождение пространства, выделенного для хранения промежуточных переменных, разумеется, если оно выделялось. В примере В.6 показаны два варианта резервирования пространства в стеке. В зависимости от используемого варианта выбирается способ освобождения стека.

Если применялся первый вариант примера В.6, то возможны два способа освобождения зарезервированного пространства. Первый способ заключается в увеличении содержимого регистра `sp` на величину `N`, соответствующую размеру выделенного пространства в байтах (`add sp, N`). Второй способ состоит в том, что в регистр `sp` копируется содержимое `bp`, при условии, что оно не изменялось в процессе выполнения подпрограммы (`mov sp, bp`).

Если для резервирования пространства в стеке использовался второй вариант примера В.6, т. е. команда `enter`, то для его освобождения применяется специальная команда `leave`, не имеющая параметров. При ее выполнении содержимое регистра `bp` копируется в `sp`, поэтому оно не должно изменяться при выполнении подпрограммы.

После выполнения описанных действий в верхушке стека должно находиться исходное содержимое регистра `bp`. Оно выталкивается командой `pop bp`, после которой можно выполнить `ret` для завершения подпрограммы.

Таким образом, если процедура ориентирована на передачу параметров в стеке, то ее выполнение завершают следующие две команды:

```
pop   bp    ; восстановление содержимого bp
ret         ; возврат из подпрограммы
```

Удаление параметров. При возврате таким способом в основную задачу или в вызывавшую процедуру в стеке остаются параметры, которые надо удалить. Это можно сделать либо при выполнении команды `ret`, либо в вызывающем модуле сразу после возврата из подпрограммы.

Если в качестве параметра команды `ret` указать число `K`, то после выборки адреса возврата оно будет прибавлено к указателю стека. Такая модификация команды `ret` введена специально для удаления параметров при возврате из подпрограммы. Число `K` задает размер освобождаемой области стека в байтах, оно всегда четное. Например, если при вызове процедуры в стек было записано `M` параметров, каждый из которых имел размер слова, то `K = 2M`.

Для удаления параметров в вызывавшей процедуре сразу после возврата из подпрограммы выполняется команда `add sp, K`, где `K` задает размер освобождаемой области стека в байтах, о чем мы только что говорили.

Какой из двух способов лучше? Однозначного ответа на этот вопрос не существует, на практике применяются оба варианта. Удаление параметров при выполнении команды `ret` проще, но оно не допустимо, если в стеке находятся *выходные параметры* подпрограммы, предназначенные для вызывающего модуля. Во избежание подобных случаев при вызове подпрограммы в стеке можно указывать *адреса выходных параметров*, а в подпрограмме записывать результаты вычислений не в стек, а по указанным адресам. По окончании выполнения подпрограммы адреса параметров не нужны, и их можно удалять командой `ret`.

Таким образом, вы можете выбрать любой из двух вариантов удаления параметров, но желательно остановиться на одном варианте и использовать его во всех случаях.

B.5. Учет особенностей компилятора

При разработке ассемблерных процедур для программных модулей, составленных на алгоритмических языках, должны быть выполнены два условия. Во-первых, процедура должна поддерживать ту форму интерфейса с вызывающим модулем, которая принята в конкретном алгоритмическом языке. Во-вторых, она должна быть описана так, чтобы компилятор с выбранного вами языка мог составить нужную последовательность команд для ее вызова. Другими словами, вызывающий модуль и процедура должны соответствовать друг другу. Кроме того, обязательно должна оставаться возможность вызова из модуля, составленного на языке Макроассемблера. Она нужна, по крайней мере, для предварительной отладки и желательно, чтобы после отладки основной текст процедуры не изменялся.

Учитывая практическую важность сказанного в состав Макроассемблера, начиная с MASM 6.0, включены специальные средства для описания подпрограмм и оформления их вызова. В данном разделе приведен краткий обзор этих средств и пример их использования. При этом автор стремился выделить наиболее важные вопросы, ответ на которые не всегда очевиден.

В полный комплект поставки MASM 6.0 и последующих версий входит подробный `HELP`, содержащий описание директив, операторов и прочих атрибутов языка. Работая с Макроассемблером, вы всегда можете получить дополнительные сведения по интересующему вас вопросу.

Различие алгоритмических языков. Алгоритмические языки созданы и создаются для решения различных классов или категорий задач, поэтому они принципиально отличаются друг от друга. Здесь нас интересуют только те различия, которые относятся к действиям, связанным с вызовом процедур. Основная часть вызова процедуры в большинстве алгоритмических языков имеет следующий вид:

```
имя_процедуры (список_параметров)
```

Однако соответствующая ей последовательность команд зависит от конкретных особенностей компилятора. Прежде всего, в именах процедуры и параметров могут различаться или не различаться заглавные и строчные буквы. Далее, параметры могут записываться в стек в порядке их перечисления в списке или в обратном порядке. Количество параметров в списке может быть фиксированным или переменным. Наконец, параметры могут удаляться из стека при возврате из процедуры или в вызывающем модуле.

В табл. В.3 перечислены особенности компиляторов, свойства которых может учитывать Макроассемблер при компиляции программных модулей.

Таблица В.3. *Допустимые языки и их особенности*

	C	Sy	St	B	F	P
Различаются строчные и заглавные буквы в именах	Да	Да	Да	Нет	Нет	Нет
Строчные буквы в именах преобразуются в заглавные	Нет	Нет	Нет	Да	Да	Да
Параметры записываются в стек в порядке их перечисления в списке	Нет	Нет	Нет	Да	Да	Да
Параметры записываются в стек в обратном порядке (от конца списка)	Да	Да	Да	Нет	Нет	Нет
Параметры удаляются в процедуре при выполнении команды `ret`	Нет	Нет	*	Да	Да	Да
Параметры удаляются в вызывающем модуле (команда `add sp, N`)	Да	Да	*	Нет	Нет	Нет
Допустим список параметров переменной длины (тип `varagr`)	Да	Да	Да	Нет	Нет	Нет

В табл. В.3 использованы следующие сокращения имен языков: C — Си, Sy — Syscall, St — Stdcall, B — Basic (Бейсик), F — Fortran (Фортран), P — Pascal (Паскаль). Пояснять смысл имен Си, Бейсик, Фортран и Паскаль нет необходимости. Слова Syscall и Stdcall не соответствуют ни одному из конкретных языков, их точное назначение в `HELP` не описано, об этом можно только догадываться. Звездочки в столбце St обозначают, что место удаления параметров зависит от формы задания их списка. При обычной форме вызова их удаляет процедура, а если количество параметров переменное, то они удаляются в вызывающем модуле.

Для того чтобы Макроассемблер учитывал при компиляции программного модуля перечисленные в табл. В.3 особенности, у директив `MODEL`, `PROC` и `PROTO` (см. ниже) появился новый спецификатор `Langtype`. Его допустимыми

значениями являются имена C, Syscall, Stdcall, Basic, Fortran и Pascal. Имя должно быть указано явно, умолчаний не существует.

Замечание

При описании названий языков в тексте книги мы также указываем русскую транскрипцию их названий, но в записи параметров на языке Макроассемблера используются *только латинские* имена.

Полное описание процедуры. По-прежнему допустима сокращенная форма описания процедуры, при которой указывается только ее тип `far` или `near`. Однако для того чтобы Макроассемблер взял на себя оформление текста процедуры, необходимо ее полное описание, которое имеет следующий вид:

```
метка PROC тип [спецификаторы] [, список_параметров]
```

Тип процедуры желательно указывать всегда, допустимые типы `far`, `far16`, `far32`, `near`, `near16` и `near32`. Имена `far` и `near` (без цифр) используются в тех случаях, когда тип процедуры соответствует типу сегмента, в котором она расположена. Тип сегмента, в свою очередь, зависит от модели памяти и директив `.386`, `.486`, `.586`.

Спецификаторы указывают тип языка (описан выше), доступность для других модулей (`Private`, `Public`, `Export`) и аргументы пролога и эпилога. По умолчанию выбирается тип языка, описанный в директиве `MODEL`, общедоступная процедура `Public` и стандартная форма пролога и эпилога, которая описана ниже.

Важно

Разделителями между именами типа и спецификаторов могут быть только пробелы. Первая запятая является признаком конца спецификаторов и начала списка параметров. При несоблюдении этого правила Макроассемблер выдает различные сообщения об ошибках и прерывает компиляцию модуля.

Список параметров по своему назначению аналогичен последовательности директив `EQU`, описанных в предыдущем разделе и позволяет использовать в основном тексте процедуры имена вместо ссылок на регистр `bp`. При его обработке Макроассемблер вычисляет смещение каждого параметра относительно регистра `bp` и подставляет нужные величины в команды подпрограммы. Если вы посмотрите листинг модуля, то увидите значения смещений для всех перечисленных в списке имен.

Параметры должны иметь уникальные имена, отличающиеся от других имен, описанных в данном модуле. После каждого имени указывается символ "двоеточие" и размер параметра (`word`, `dword` и т. п.). Если это не сделано, то по умолчанию размер параметра зависит от разрядности сегмента, в котором расположена процедура, — `word` для 16-разрядных и `dword` для 32-разрядных сегментов.

Таким образом, в полном описании процедуры появились два наиболее важных элемента — тип языка и список фактических параметров. Они содержат исчерпывающую информацию о способе компиляции процедуры.

Пролог и эпилог. В процессе компиляции подпрограммы Макроассемблер включает в объектный модуль две группы команд, одна из которых называется прологом (`prologue`), а другая эпилогом (`epilogue`). Пролог вставляется перед основным текстом подпрограммы, а эпилог перед командой `ret`. Вставляемые команды вам уже известны.

Стандартный вариант пролога содержит следующие две команды:

```
push        bp          ; сохранение исходного содержимого bx
mov         bp, sp      ; bx = адрес верхушки стека
```

Соответственно эпилогом являются команды

```
pop   bp    ; восстановление содержимого регистра bx
ret   N     ; изменение команды ret зависит от langtype
```

Если значением `Langtype` является Basic, Fortran или Pascal, то при формировании эпилога к команде `ret` будет добавлен суммарный размер параметров в байтах для очистки стека при возврате из подпрограммы. Если значением `Langtype` является C, то команда `ret` не изменяется, а в текст вызывающего модуля вставляется команда `add sp, N`.

Обычно команды пролога и эпилога отсутствуют в листинге. Для того чтобы они в нем оказались в начале текста модуля, укажите директиву `.Listall` (точка перед именем обязательна).

Пролог и эпилог можно как исключить, так и расширить. Для исключения пролога в текст модуля перед описанием сегмента включается директива

```
OPTION PROLOGUE:NONE
```

Аналогичная директива существует и для эпилога, но ее использовать не обязательно. Макроассемблер вставляет эпилог *только перед командой* `ret`, поэтому для исключения эпилога достаточно использовать имена `retn`, `retf` или `ret N`, в зависимости от обстоятельств. Обратите внимание, наличие или отсутствие эпилога не влияет на включение команды `add sp, N`, поскольку она расположена в другом программном модуле.

В пролог можно добавить сохранение в стеке содержимого используемых регистров, а в эпилог — их восстановление. Для этого в описание процедуры включается спецификатор `USES`, а после него перечисляются используемые в подпрограмме регистры.

Замечание

Разделителем между именами регистров является пробел, указание запятых не допустимо.

Если процедура содержит свой сегмент данных, то в пролог можно включить команды переопределения содержимого регистра ds, при этом в эпилог добавляется команда, восстанавливающая из стека исходное значение регистра ds. Для выполнения этих действий в описание процедуры включается спецификатор <loadds> (угловые скобки обязательны).

Таким образом, пролог и эпилог позволяют не записывать в основном тексте вспомогательные действия, выполняемые при входе в процедуру и выходе из нее. Целесообразность включения команд пролога и эпилога решается в каждом конкретном случае с учетом назначения процедуры.

Оформление процедуры cnvindec. В комплект поставки Макроассемблера входят исходные тексты программных модулей, иллюстрирующие различные случаи применения директивы PROC. Тем не менее автор счел целесообразным показать, что изменится в тексте процедуры cnvindec (см. пример В.5), если в ее описание включить полную форму директивы PROC. Измененный текст приведен в примере В.7.

Пример В.7. Измененный текст процедуры cnvindec

```
        .LISTALL                ; разрешаем печатать все
subr    SEGMENT word public 'subr'; начало сегмента subr
        .386                    ; задаем тип микропроцессора
dten    dd   10                 ; константа для умножения на 10
cnvindec PROC FAR PASCAL USES edx fs si, address:dword
        lfs  si, address        ; fs:si = адрес начала строки текста
        mov  address, 0         ; result = 0 очистка результата
cnvloop: xor  eax, eax          ; очистка eax
        lods byte ptr fs:[si]   ; al = очередной символ строки
        cmp  al, '0'            ; код символа меньше кода цифры 0 ?
        jb   endcnv             ; -> да, конец формирования числа
        cmp  al, '9'            ; код символа больше кода цифры 9 ?
        ja   endcnv             ; -> да, конец формирования числа
        sub  al, 30h            ; вычитаем код цифры 0
        xchg eax, address       ; переставляем eax и result
        mul  cs:dten            ; edx:eax = result * 10
        add  address, eax       ; result = result + eax
        jmp  short cnvloop      ; -> на начало цикла преобразования
endcnv: pop  si                 ; восстанавливаем содержимое si
        pop  fs                 ; восстанавливаем содержимое fs
        pop  edx                ; восстанавливаем содержимое edx
        pop  bp                 ; восстанавливаем содержимое bp
        retf                    ; возврат из подпрограммы
cnvindec ENDP                   ; конец блока процедуры
subr    ENDS                    ; конец сегмента subr
        END                     ; конец текста модуля
```

В тексте примера В.7 отсутствуют директивы `PUBLIC` и `ASSUME`. Первая из них не нужна потому, что при полном описании процедура, по умолчанию, является общедоступной. Директива `ASSUME` необходима только при использовании компилятора MASM 5.1, а данный пример предназначен для компиляции на более поздних версиях, которые не требуют указания этой директивы. Зато в тексте модуля появилась новая директива `.Listall`, она нужна для того, чтобы Макроассемблер включил в листинг команды пролога.

В основном тексте процедуры отсутствуют ссылки на регистр `bp`, вместо них используется имя параметра `address`. Если вы посмотрите листинг, то увидите, что ему соответствует тип `Dword` и значение `[bp+6]`.

В пролог, кроме двух стандартных команд, включены команды, выполняющие сохранение в стеке регистров `edx`, `fs` и `si`. Эпилог в данном случае исключен, поскольку использована команда `retf`, а не `ret`. Поэтому команды, восстанавливающие содержимое регистров `si`, `fs`, `edx` и `bp`, включены в текст процедуры, первая из них имеет метку `endcnv`.

Замечание

Процедура примера В.7 предназначена для вызова из программных модулей, составленных на языке ассемблера. Это объясняется тем, что сформированное число возвращается в стеке, именно по этой причине в тексте процедуры использована команда `retf`, исключающая вставку эпилога. Если вызывающий модуль составлен на алгоритмическом языке, то взять результат из стека, без специальных ухищрений, невозможно. Проще внести изменения в основной текст процедуры, позволяющие вызывать ее из модуля, составленного на любом языке.

Первый вариант таких изменений заключается в том, что перед выходом из процедуры сформированное число помещается в регистр `eax`. Для этого пять последних команд текста процедуры заменяются следующими:

```
endcnv: mov   eax, address     ; копируем результат в eax
        ret                    ; стандартная форма команды возврата
```

В данном случае выполнение процедуры завершает команда `ret`, поэтому Макроассемблер вставит перед ней эпилог, а к команде `ret` добавит операнд, равный 4, для выталкивания параметров из стека.

Обоснованием этого варианта является то, что в алгоритмических языках функция возвращает результат в регистре `eax` (или `ax`). Например, в модуле, составленном на Фортране, возможна такая форма вызова данной функции:

```
argument = cnvindec (string)
```

Описанный вариант применим, если результатом является только одно число. В общем случае у процедуры появляются дополнительные параметры, содержащие *адреса* для записи результатов вычислений. В конце раздела В.4 было сказано, что если выходные параметры заданы в виде адресов, то при возврате из процедуры допустимо их удаление из стека.

При описании примера В.5 говорилось, что в регистре `al` передается код символа, являющегося ограничителем числа. Если он используется в вызывающем модуле, то к процедуре примера В.7 надо добавить еще один параметр, содержащий адрес для записи кода символа.

Таким образом, при составлении основного текста процедуры важно учитывать, что именно передается в качестве параметра — значение переменной или ее адрес. От этого будут зависеть действия, выполняемые как в самой процедуре, так и в вызывающем модуле.

Директива вызова процедуры. Для вызова процедур, ориентированных на передачу параметров в стеке, предназначена специальная директива:

```
Invoke имя_процедуры [, список_параметров]
```

Список, если он указан, содержит фактические параметры, имена которых должны быть явно описаны в вызывающем модуле или объявлены в директиве `EXTERN`. Параметры отделяются от имени процедуры и друг от друга запятыми. В качестве параметров могут использоваться регистры, пары регистров, имена и адреса переменных. При этом в любом случае размеры *фактических параметров* должны соответствовать размеру *формальных параметров*, перечисленных в описании процедуры.

Если в списке указано имя переменной, то процедуре передается ее значение. Если указано имя регистра, то передается его содержимое. Для передачи содержимого пары регистров между их именами дважды указывается символ "двоеточие". Например, если формальный параметр описан как двойное слово и предназначен для передачи адреса, то в качестве фактического параметра можно указать пару регистров `DS::SI`.

Для передачи адреса переменной перед ее именем указывается ключ `ADDR`. Если формальный параметр описан как двойное слово, то процедуре передается сегмент, в котором описана переменная и ее смещение (адрес) относительно начала сегмента. Если формальный параметр описан как слово, то процедуре передается только сегмент, в котором расположена переменная, в некоторых случаях это может пригодиться на практике.

При передаче адреса вместо имени переменной может быть указано выражение в форме, позволяющей Макроассемблеру однозначно определить адрес (смещение) переменной в сегменте, например, `ADDR linbuf+2`.

При обработке директивы `Invoke` Макроассемблер вставляет в объектный модуль группу команд, выполняющих размещение параметров в стеке, а после них команду вызова процедуры. Обычно результат подстановки в листинге отсутствует. Чтобы увидеть его в начале текста модуля, надо поместить директиву `.Listall` (точка обязательна).

Последовательность записи параметров в стек зависит от имени, указанного в качестве `Langtype` в описании процедуры. Если указаны имена Basic,

Fotran или Pascal, то параметры записываются в стек в порядке их перечисления в списке директивы `Invoke`. Если же указаны имена C, Syscall или Stdcall, то параметры записываются в стек в обратном порядке (см. табл. В.3). Это обеспечивает соответствие способов записи и использования передаваемых в стеке параметров.

Вызов cnvindec. В качестве примера покажем, как может выглядеть вызов процедуры, описанной в примере В.7. Для этого в вызывающем модуле указывается следующая директива:

```
Invoke cnvindec, ADDR linbuf
```

Предполагается, что имя `linbuf` соответствует буферу, содержащему строку цифр в коде `ASCII` (см. раздел В.3). Если `linbuf` расположена в сегменте данных, то директива преобразуется в следующие команды:

```
push      ds              ; запись содержимого ds
push      offset linbuf   ; запись адреса linbuf
call      cnvindec        ; вызов подпрограммы cnvindec
```

Процедура содержит один параметр, поэтому последовательность команд, записывающих величины в стек, не зависит от языка.

Замечание

Описание процедуры *обязательно* должно предшествовать ее вызову директивой `Invoke`. Поэтому если вызывающая часть и процедура находятся в одном программном модуле, то текст процедуры должен быть расположен *перед* текстом вызывающей части. Это требование остается в силе независимо от того, в одном или в разных сегментах программного модуля описаны процедура и вызывающая часть.

Прототип процедуры. Если процедура и вызывающий модуль готовятся независимо друг от друга и объединяются только при компоновке задачи, то выполнить указанное выше условие невозможно. В таком случае в текст вызывающего модуля включается *прототип процедуры*. Для его описания существует специальная директива, имеющая следующий формат:

```
имя_процедуры PROTO тип [langtype] [, список_параметров]
```

Для составления прототипа нужен либо исходный текст процедуры, либо исчерпывающее описание способа вызова соответствующего ей объектного или библиотечного модуля, либо образец прототипа, взятый из другого программного модуля, вызывающего данную процедуру.

Рассмотрим конкретный случай. Предположим, что процедура, описанная в примере В.7, подготовлена в виде объектного модуля. Для создания ее прототипа надо взять из примера В.7 строку, содержащую директиву `PROC`, заменить слово `PROC` на `PROTO` и исключить описание сохраняемых в стеке регистров (`USES edx fs si`).

В результате получится следующий прототип:

```
cnvindec PROTO FAR PASCAL, address:dword
```

Из этого примера видно, что для получения прототипа из описания директивы PROC исключаются только те спецификаторы, которые нужны при компиляции процедуры и не используются при компиляции вызывающего модуля.

При компиляции вызывающего модуля Макроассемблеру недоступно описание процедуры, поэтому имя процедуры, ее тип и значение Langtype он "принимает на веру", но имена параметров проверяет обязательно.

В списке PROTO указываются имена *формальных параметров*. Они не должны совпадать с именами переменных, описанных в вызывающем модуле. В противном случае Макроассемблер выдаст сообщение о повторном определении имени и прервет процесс компиляции. При успешной компиляции имена и размеры параметров включаются в объектный модуль и используются компоновщиком при сборке задачи. На стадии сборки известны описание процедуры и способ ее вызова и компоновщик проверяет их соответствие друг другу.

Заключение. Практическая ценность директив PROC, PROTO и INVOKE состоит в том, что при их использовании Макроассемблер самостоятельно учитывает особенности компиляторов при формировании параметров в стеке и оформлении пролога и эпилога процедуры. Это позволяет разрабатывать процедуры, не зависящие от языка программирования, на котором составлен вызывающий модуль. Кроме того, директива PROTO позволяет использовать в вызывающем модуле, составленном на языке ассемблера, модули из библиотек, входящих в состав компиляторов Си, Фортрана, Паскаля и Бейсика. Однако для этого вам должно быть доступно описание этих библиотек.

В тексте данного раздела мы не упоминали об одной важной детали. Для успешной разработки общедоступных процедур недостаточно знать только язык ассемблера. Вы должны иметь представление не только о языке программирования, на котором будет составлен вызывающий модуль, но и о возможностях конкретного компилятора с этого языка. Все существующие компиляторы поддерживают расширенные версии языков программирования, Макроассемблер не является исключением. А вот в чем заключается это расширение, зависит от конкретной реализации компилятора.

И последнее. Навыки программирования приобретаются не при чтении книг, а в процессе практической деятельности. Успехов вам на этом поприще, уважаемый Читатель и Программист!

Список литературы

1. Андрианов С. А. VESA: стандарт новый, проблемы старые. — Мир ПК, 1998, № 7.
2. Андрианов С. А. VESA 2.0: программируем в защищенном режиме. — Мир ПК, 1998, № 8.
3. Бердышев Е. М. Технология MMX. Новые возможности процессоров P5 и P6. — М.: ДИАЛОГ-МИФИ, 1998. — 234 с.
4. Борн Г. Форматы данных. — Киев: Торгово-издательское бюро BHV, 1995. — 472 с.
5. Григорьев В. Л. Видеосистемы ПК фирмы IBM. — М.: Радио и связь, 1993. — 192 с.
6. Зубков С. В. Ассемблер для DOS, Windows и UNIX. — М.: ДМК, 2000. — 608 с.
7. Пустоваров В. И. Ассемблер. — Киев: Издательская группа BHV, 2000. — 480 с.
8. Уилтон Р. Видеосистемы персональных компьютеров IBM PC и PS/2. — М.: Радио и связь, 1994. — 384 с.
9. Фролов А. И., Фролов Г. В. Графический интерфейс GDI в MS Windows. — М.: ДИАЛОГ-МИФИ, 1994. — 288 с.
10. Huffman D. A. A method for the construction of minimum redundancy codes. — Proceedings IRE, 1952, vol. 40, p. 1098—1101.

Предметный указатель

A

AGP — гнездо на материнской плате 12
Alpha blending — смешение цветов с весовыми коэффициентами 321

B

Background — фон, на котором изображен символ 162
BMP — формат графических файлов для Windows 331

C

CGA-палитра — стандартный набор из 16-ти цветов 135
Chroma keying — фильтрация цвета 320
CMY — дополнительный набор базовых цветов 131
Color registers — другое название DAC-регистров 137
Cur — тип файлов, содержащих рисунки курсоров 208
Cur_win — переменная, содержит текущий номер окна 54

D

DAC — преобразователи код-аналог 130
Direct color — цвет в коде точки:
◊ Hi-Color — 32K или 64K цветов 262
◊ True Color — 16M цветов 263
Direct3D — графический пакет для Windows 20

E

Event — событие, любое изменение состояния мыши 242
Event handler — обработчик событий 250

F

Fogging — затягивание туманом 326
Foreground — точки изображения символа на экране 162
Fwidth — переменная, содержит ширину рисунка в файле 118

G

GrUnit:
◊ единица приращения номера окна видеопамяти 52
◊ переменная, содержит значение GrUnit 53

H

Hidepnt — подпрограмма удаления рисунка курсора 221

Horsize — переменная, содержит количество столбцов на экране 50

I

Ico — тип файлов, содержащих рисунки значков 208
Icon — стандарт хранения файлов типов ico и cur 209
Iheight — переменная, содержит количество строк в рисунке 114
Inline — подпрограмма ввода символов с клавиатуры 203
ISA — системная шина 12
Iwidth — переменная, содержит ширину рисунка в точках 114, 305

J

JPEG — способ сжатия полноцветных рисунков 316
JPG — формат обмена графическими файлами 318

L

LZW — способ сжатия данных 123

M

Motion — подпрограмма перемещения курсора 247
Mouse — макрос, формирует обращение к драйверу мыши 232
Mousm — подпрограмма перемещения курсора по прерываниям 255

N

NxtWin — подпрограмма установки следующего окна 55

O

OEM — изготовитель оборудования 26
OpenGL — графическая библиотека фирмы Silicon Graphics 20
Overscan — регистр, отвечающий за цвет границ экрана 142

P

Palette registers — не используются в режимах VESA 140
PCI — системная шина 12
PCX — стандарт хранения графических данных 118
Pels (Picture Elements) — элементы рисунка, т. е. точки 333
PopReg — макрос, выталкивающий аргументы из стека 67
PPG — графический режим, использующий палитру 22, 77
PrevWin — подпрограмма установки предыдущего окна 55
PushReg — макрос, сохраняющий аргументы в стеке 67

R

RGB — основной набор базовых цветов 131
RLE — способ сжатия графических данных 118

S

SetWin — подпрограмма установки указанного окна 55
Showpnt — подпрограмма построения рисунка курсора 219
Sprite — небольшой перемещаемый рисунок 319
SVGA-монитор 10

T

Tglpntr — подпрограмма, изменяет состояние курсора на противоположное 195, 215
Transparent Surface — прозрачная перегородка 325

V

VBE — стандарт VESA на расширение функций BIOS 21
Vbuff — переменная, содержит код сегмента видеобуфера 66
Versize — переменная, содержит количество строк на экране 50
VESA — ассоциация по стандартизации 21
VGA:
◊ графический стандарт фирмы IBM 13
◊ монитор 10
Video Services — группа функций BIOS, вызываемых через int 10h 14
VMC:
◊ переменная, содержит адрес VMC 52
◊ процедуры BIOS для работы с окнами 54