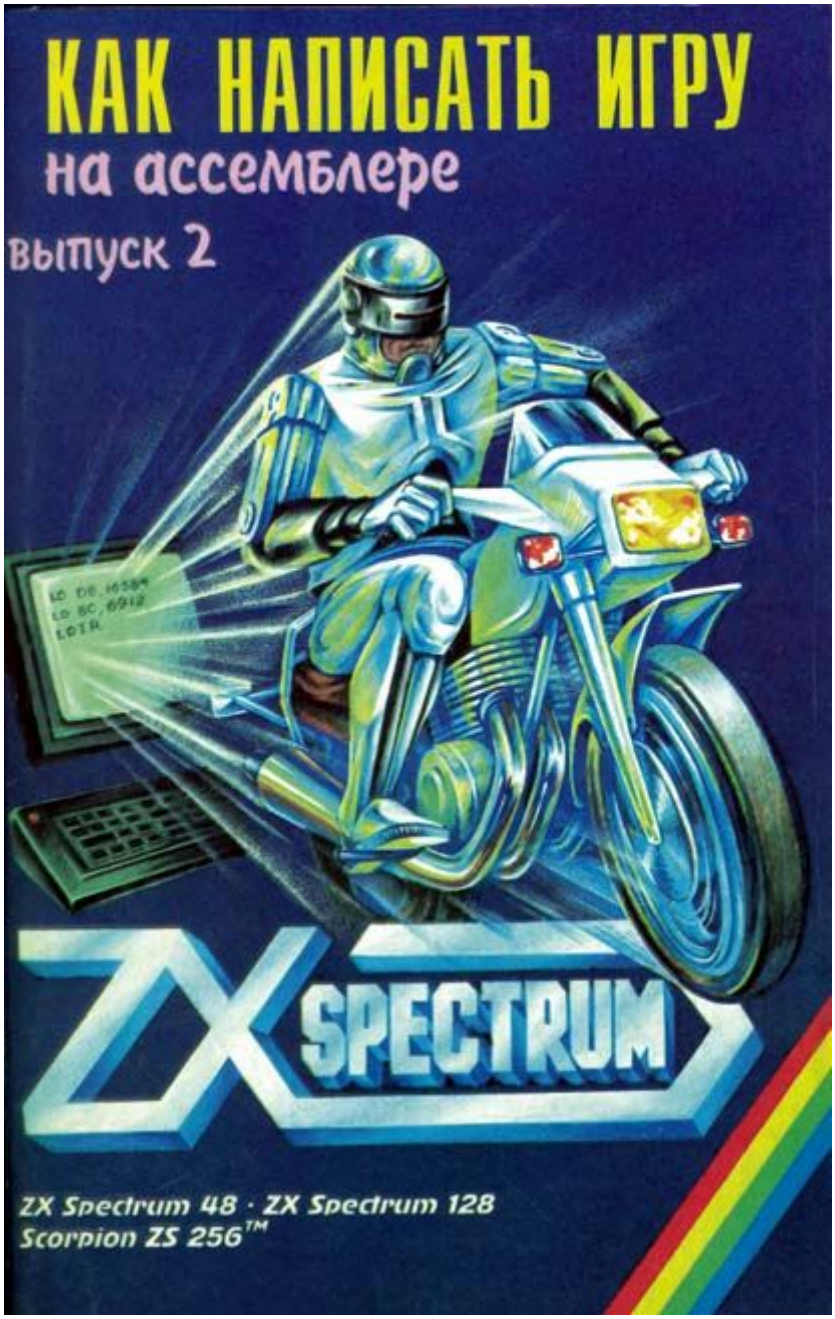

КАК НАПИСАТЬ ИГРУ
на ассемблере
выпуск 2
ZX SPECTRUM
ZX Spectrum 48 · ZX Spectrum 128
Scorpion ZS 256™

Александр Евдокимов, Александр Капульцевич, Игорь Капульцевич

Как написать игру на ассемблере для ZX Spectrum

Серия «Все о ZX Spectrum»
Книга девятая

Главный редактор	*В. Усманов*
Научный редактор	*А. Ларченко*
Художественный редактор	*Н. Вавулин*
Художник	*А. Андрейчук*
Корректор	*Г. Седова*
Оригинал-макет подготовил	*А. Евдокимов*
HTML-версия	*А. Евдокимов*

Эта книга - вторая из цикла «Как написать игру для ZX Spectrum». Речь пойдет о применении ассемблера для написания игровых программ, что позволяет создавать более яркие и динамичные игры, применять разнообразные спецэффекты.

Из книги вы узнаете: что такое ассемблер; как вводить и редактировать программы на этом языке; как работает микропроцессор Z80; как использовать подпрограммы ПЗУ; как создаются различные видео- и аудиоэффекты.

Книга написана очень доступным языком и будет полезна как начинающим, так и более «продвинутым» программистам.

ISBN 5-7190-0038-0

ATTENTION: This PDF is made on authors' ZIP available at:
http://www.nevsky.net/~aevdo/

I'm too lazy to read from a screen all the time so I copied all authors' HTMLs and pasted into one DOC ready-to-print.

ATTENTION! Links DON"T WORK (although they are blue & underlined) so even don't try to click them.

Сделано в 2001 году — חינוך

Оглавление

ВВЕДЕНИЕ

Эта книга адресована в первую очередь тем, кого уже перестал удовлетворять несколько ограниченный и неповоротливый Бейсик и кто мечтает наконец научиться писать программы на ассемблере. Книга рассчитана на достаточно подготовленного читателя, прошедшего «боевое крещение» Бейсиком, а новичкам в программировании мы можем порекомендовать первую книгу из серии «Как написать игру для ZX Spectrum». Надеемся также, что и профессионалы смогут найти здесь для себя некоторые зерна истины.

Как и в предшествующей книге, речь здесь пойдет преимущественно об игровых программах, однако хотим вас предупредить заранее, что ассемблер - штука серьезная и нам не раз придется погружаться в пучины мудреных понятий и терминов. Но со своей стороны мы обещаем сделать эти погружения не слишком головокружительными, смягчив суровую необходимость занимательными примерами.

Возможно, вас несколько смутили только что прочитанные строки, да и раньше вам, быть может, не раз приходилось слышать, мол, писать программы на ассемблере невероятно сложно. Но, право, не так страшен ассемблер, как его малюют, а что касается сложностей, так вспомните свои первые шаги в том же Бейсике.

Конечно, между этими двумя языками существуют различия, причем принципиальные. Так, действие почти каждого оператора Бейсика можно сразу увидеть на экране [1], чего не скажешь об ассемблере. Здесь, если вы хотите получить какой-то видимый невооруженным глазом результат, нужно, засучив рукава, создавать целую программу. И если в Бейсике достаточно только набрать текст программы, чтобы она заработала, то ассемблерный текст нужно еще специальным образом обработать - оттранслировать (трансляция исходных текстов на ассемблере в принципе очень похожа на обработку бейсик-программ с помощью компиляторов.) Однако надеемся, что после внимательного изучения данного пособия обозначенные выше сложности не покажутся вам серьезным препятствием.

Вообще же, основная сложность ассемблера заключается, пожалуй, в одном: текст программы получается очень длинным из-за того, что каждая инструкция выполняет какое-то одно элементарное действие типа сложения или вычитания, поэтому для получения ощутимого результата приходится писать длинный ряд инструкций. Зато, в отличие от Бейсика, понять смысл каждой команды, как нам кажется, значительно проще. Во всяком случае, мы попытаемся и вам передать нашу уверенность в этом.

Поскольку исходные тексты программ на ассемблере получаются весьма громоздкими, мы не смогли поместить в книге ни одной полноценной игры, компенсировав этот недостаток массой небольших, но очень полезных примеров и фрагментов игровых программ. И если вы уже имеете некоторый опыт в создании компьютерных игрушек, то вам не составит большого труда собрать свою собственную программу из приведенных в книге «кирпичиков».

Еще одна сложность ассемблера состоит в том, что в нем отсутствуют привычные сообщения об ошибках. Ассемблер будет «ругаться» только тогда, когда вы попытаетесь подсунуть ему несуществующую инструкцию или еще что-нибудь в этом роде. А так можно шутки ради написать заведомо неработоспособную программу и ассемблер ничтоже сумняшеся оттранслирует ее, да еще подбодрит вас сообщением, мол, ошибок нет, программа получилась - просто блеск. Но вот что любопытно: этот, казалось бы, неприятный недостаток обращается в несравненное достоинство, совершенно недоступное никакому другому языку! Освоив ассемблер, вы наверняка в один прекрасный момент почувствуете, что значит истинная свобода. Ведь теперь вы не будете скованы никакими условностями, и никто не рявкнет вдруг из-за спины: «это нельзя, то нельзя!» Отныне все условности для себя будете создавать вы сами и никто не помешает в любой момент отбросить их в сторону.

Теперь по сложившейся традиции необходимо сказать несколько слов о структуре книги. Будем исходить из предположения, что вам уже известны составные части игровой программы и способы их создания, и мы лишь мельком «пробежимся» по этому материалу, сосредоточив основное внимание на средствах достижения той или иной цели. Вначале мы продемонстрируем, как можно получить те же результаты, что и при использовании операторов Бейсика, а затем все чаще станем применять методы, недоступные языкам высокого уровня и дающие совершенно уникальные эффекты.

Первая глава вводит вас в мир игровых программ, показывает, из каких основных частей они состоят: заставки, игрового пространства, блока взаимодействия с играющим, блока оценки игровой ситуации и блока музыкального сопровождения игры. В ней дается общее представление о каждой из этих частей и их функциях, приводятся примеры и рисунки.

Как вы понимаете, прежде чем приступать к изучению нового языка, необходимо четко уяснить, что он собой представляет, усвоить несколько новых понятий и на самых элементарных примерах научиться писать исходные тексты. Поэтому во второй главе рассказывается о том, что такое машинные коды и ассемблер, чем они отличаются друг от друга и каковы преимущества и недостатки ассемблера. Вы познакомитесь с организацией памяти ZX Spectrum, с регистрами и регистровыми парами, то есть со всем тем, без чего невозможно написать даже самую маленькую ассемблерную программку.

В третьей главе, шаг за шагом, мы научим вас вводить и редактировать программы в ассемблере GENS4, расскажем о структуре строки исходного текста, трансляции программ, о сохранении и удалении текстов, и о многом другом, с чем вы столкнетесь буквально в первые минуты знакомства с ассемблером.

Четвертая глава показывает, как подготовить экран к работе, то есть установить его атрибуты, бордюр, а при необходимости очистить от ненужных символов; как сделать на нем разные надписи, вывести числа и создать простейшие изображения. При этом каждый, даже самый маленький шаг, сопровождается программой и подробными комментариями к ней. После нескольких уроков приводится вполне самостоятельная программа статической заставки, на примере которой вы узнаете, как создаются блоки данных для вывода на экран простейших спрайтов, всевозможных рамок и различных надписей не только латинскими, но и русскими буквами.

В пятой главе мы обсудим вопрос о том, как заставить двигаться по экрану отдельные символы, целые строки и небольшие спрайты. Особое внимание уделено принципам организации циклов, составлению и использованию подпрограмм, в соответствии с которыми разработано несколько полезных для вашего будущего творчества процедур. Показано, как сформировать и заставить двигаться окна, «растворить» изображение на экране, сделать настоящую мультипликационную картинку.

Шестая глава посвящена многокадровой заставке и проблемам, которые возникают при ее создании. В частности, приводятся программки, формирующие простые звуки, а также делающие из стандартного набора символов высокие и широкие буквы. Если же вы не чувствуете в себе особых способностей к рисованию, мы предложим простой способ создания изображения названия игры. Значительное внимание уделено скроллингу окон во всех направлениях даются примеры и пояснения к их использованию. В конце главы приводится полноценная многокадровая заставка, которую с небольшими изменениями или без таковых, вы сможете использовать в своих игровых программах.

Из сказанного может создаться впечатление, что мы слишком уж много внимания уделяем заставкам, но на это есть по меньшей мере две причины: во-первых, заставка, как-никак - это лицо любой игровой программы, а во-вторых, на этих примерах проще всего изучать команды машинного языка. Поэтому к концу шестой главы вы усвоите большую их часть и когда в следующей, седьмой главе, перейдете к изучению игрового пространства, можно будет заняться чистым творчеством, не отвлекаясь на техническую сторону программирования.

В седьмой главе мы познакомим вас с основой основ любой игровой программы - игровым пространством и детально рассмотрим круг вопросов, связанных с формированием и быстрым выводом на экран пейзажей и спрайтов. Чтобы облегчить их создание, предлагается простая программа, которая позволяет легко получить готовые к использованию спрайты и даже целые спрайт-файлы. Подробно изложена проблема восстановления фона при движении спрайтов в игровом пространстве и описаны пять наиболее употребительных способов, каждый из которых проиллюстрирован примером. В заключительном разделе показано, что элементы игрового пространства можно формировать не только непосредственно на экране, но и в памяти компьютера, и после окончания построения изображения быстро выводить его на экран, создавая многоплановые мультипликационные картинки.

Восьмая глава показывает, как управлять спрайтами во время игры, перемещая их во всех направлениях, в том числе по диагонали. В зависимости от сюжета и цели игры вы получаете возможность остановить свой выбор на управлении от клавиатуры или одном из типов джойстиков. Но можете воспользоваться и универсальной процедурой, благодаря которой спрайты одинаково реагируют как на клавиши, так и на повороты ручки джойстика.

В следующей, девятой главе, речь идет о том, как оценивать действия играющего и его противников на протяжении всей игры: начислять очки, следить за количеством «жизней» и оставшимися ресурсами, и вообще за всем тем, что поддается подсчету. А для того, чтобы вычисления не вызывали больших проблем, показано, как воспользоваться услугами «зашитой» в ПЗУ компьютера программой, называемой калькулятором. С ее помощью можно выполнять математические действия не только над целыми, но и над дробными числами. Наконец, детально рассмотрена такая малопонятная для многих начинающих программистов вещь, как прерывания. Показано, как написать свою собственную процедуру обработки прерываний на примере контроля оставшегося до конца игры времени.

Из десятой главы вы узнаете о нескольких различных способах извлечения звуков. В ней мы покажем, как получить отдельные акустические эффекты для звукового оформления игры, а также, как заставить компьютер проигрывать целые музыкальные фразы. Обладатели компьютеров ZX Spectrum 128 и Scorpion ZS 256 получат представление о работе с трехканальным музыкальным сопроцессором. В последнем разделе главы приводится краткое описание второй версии музыкального редактора Wham - Wham FX, предназначенного для программирования мелодий, исполняемых музыкальным сопроцессором.

Завершает книгу глава одиннадцатая, в которой будут обсуждены некоторые «профессиональные» возможности ассемблера GENS4. К ним относится работа с макросами, условная трансляция, методика ассемблирования программ больших размеров, состоящих, возможно, из нескольких исходных файлов.

Изначально мы планировали включить в книгу подробное описание популярной игры Tetris и на ее примере показать, как создается полноценная программа на ассемблере, с чего нужно начать и чем закончить. Но увы, ограничение объема не позволило сделать этого. Тем не менее, мы все же надеемся, что данная книга не последняя в серии «Как написать игру» и позже удастся опубликовать не только упомянутую программу, но и многие другие, не вошедшие в настоящее издание.

ГЛАВА ПЕРВАЯ,

в которой обсуждается структура игровой программы

Если вы удостоили вниманием первую книгу из серии «Как написать игру», то о структуре игровой программы знаете уже более чем достаточно и не много потеряете, пропустив эту главу. Но учитывая интересы тех, кто упомянутую книгу даже в руках не держал, мы все же повторимся немного и поясним некоторые термины, которыми будем в дальнейшем пользоваться.

Несмотря на несметное количество существующих компьютерных игр различного уровня сложности и жанра, все же есть в них что-то общее, поддающееся классификации. Если проанализировать любую из них от начала загрузки до появления на экране сообщения GAME OVER (игра окончена), то можно выделить несколько частей, которые, хотя и связаны между собой в единое целое какими-то общими переменными и данными, но тем не менее, выполняют в игре вполне самостоятельные функции. Это позволяет разрабатывать каждую из частей независимо от других (однако, не забывая о связях с остальными частями), что в значительной степени упрощает создание игровой программы в целом. О каких же частях идет речь? Это:

- заставка;
- игровое пространство;
- блок взаимодействия с играющим;
- блок оценки игровой ситуации;
- блок звукового сопровождения игры.

Одна из основных задач этой книги состоит именно в том, чтобы научить вас реализовывать в ассемблерных программах каждую из перечисленных выше частей, но вначале посмотрим, что же они собой представляют, какие функции в игровых программах выполняют и какими примерно они должны быть.

ЗАСТАВКА

Вы знаете, что заставкой обычно называют картинку, появляющуюся во время загрузки игровой программы. Такие картинки имеют тот же смысл, что и рекламные щиты у дверей кинотеатров. Значительную площадь, как правило, занимает название игры, а все, что изображено на экране, так или иначе связано с ее сюжетом. Здесь же обычно сообщается о фирме-изготовителе, авторах программы и годе ее создания (рис. 1.1).

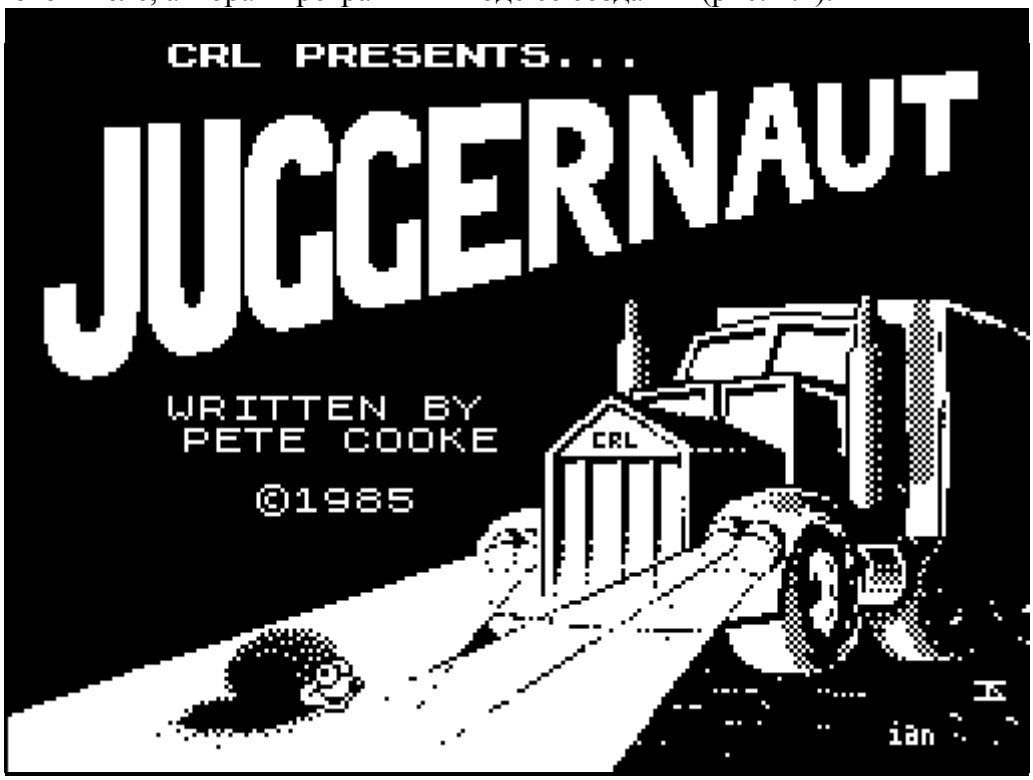


Рис. 1.1. Загрузочная картинка к игре JUGGERNAUT

Часто только это изображение и называют заставкой, хотя на самом деле это лишь первая ее часть, помогающая игроку морально подготовиться к тяготам предстоящего сражения или лишениям, связанным с поисками сокровищ. Однако в солидных игровых программах заставка чаще всего разбивается на несколько частей, именуемых кадрами. Объясняется это очень просто: перед началом игры на экране должно появиться столь большое количество информации, что «впихнуть» ее в один кадр практически невозможно. Поэтому после окончания загрузки программы на экране появляется основной кадр заставки, называемый *меню*, в котором дается список действий, необходимых для настройки программы или для получения той или иной информации (рис. 1.2).

Рис. 1.2. Меню программы EAGLES NEST

Меню - это что-то вроде диспетчерского пункта, из которого можно вызвать любой кадр заставки. Обычно в меню выбирается способ управления игрой: один из типов джойстика (Kempston, Sinclair и т. д.) или клавиатура (Keyboard); начальный уровень (Level); запуск игры (Start Game). После выбора управления от клавиатуры часто появляется еще один кадр - Define Keys, в котором предлагается назначить удобные именно для вас управляющие клавиши. В ряде игр как отдельные кадры заставки выводится таблица рекордов (Hi Score), подсказка (Help), рассказ о сюжете (History) и прочая информация.

ИГРОВОЕ ПРОСТРАНСТВО

Игровое пространство мы ранее (см. [1]) определили как совокупность всех подвижных и неподвижных объектов, появляющихся на экране во время игры. Однако во многих программах игровое пространство имеет значительную протяженность и в каждый момент времени можно видеть лишь небольшую его часть, в то время как все остальное остается за кадром. Поэтому иногда мы будем пользоваться и другим термином - *игровое поле*. Это как раз и есть тот фрагмент игрового пространства, который можно наблюдать на экране.

Попытаемся теперь разобраться в вопросе о том, из каких частей состоит игровое пространство и, в самых общих чертах, как они создаются. Заметим, что более подробно эти вопросы будут рассмотрены в соответствующих главах.

В большинстве случаев игровое пространство может быть разбито на три существенно отличающиеся части. К первой отнесем все неподвижные элементы изображения, имитирующие небо, землю, воду, космическое пространство со звездами и планетами, а также «мелкие» объекты: здания, деревья, мосты, лабиринты и т. д. В дальнейшем все это для краткости будем называть *пейзажем*. Пример простейшего пейзажа - шахматная доска в игре

CHESS. А в таких программах как SABOTEUR, DIZZY или ROBOCOP пейзажи представляют собой настоящие произведения искусства (рис. 1.3). Они состоят из большого количества различных предметов и выглядят весьма правдоподобно.

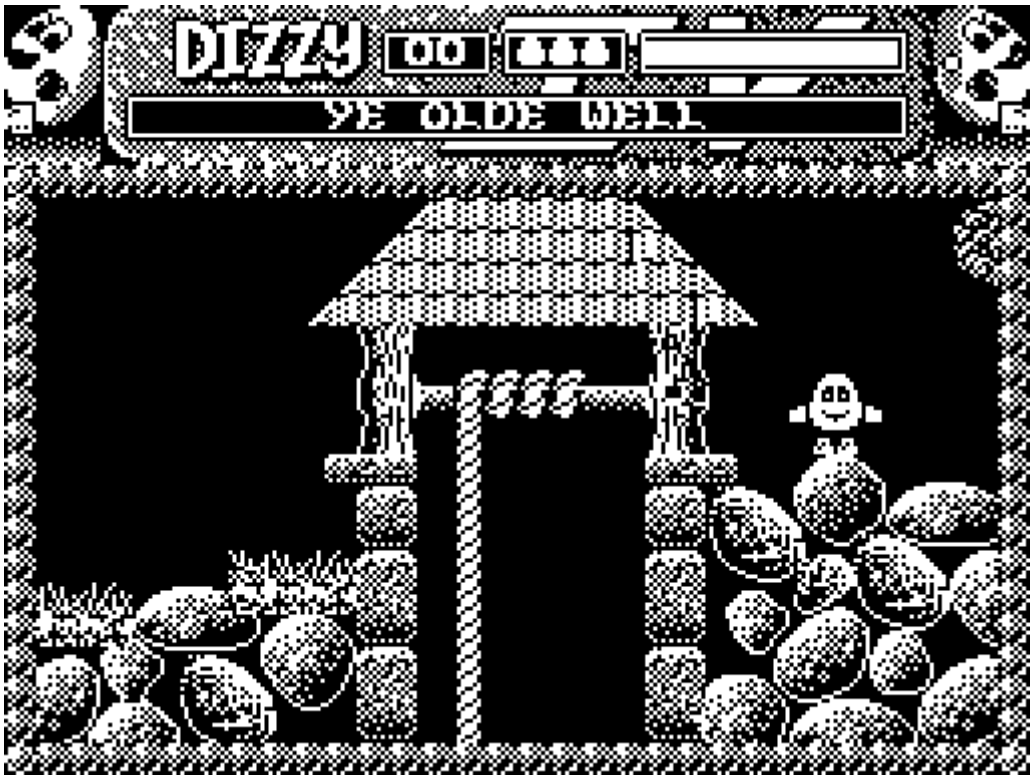


Рис. 1.3. Игровое пространство программы DIZZY4

Вторую часть составляют подвижные объекты, которые перемещаются сами по себе и на которые вы не можете оказывать непосредственного воздействия. Как правило, к этой группе относятся изображения ваших противников, например, силуэты вертолетов, кораблей и самолетов в игре RIVER RAID или проносящиеся мимо вас мотоциклы в SPEED KING2.

Наконец, к третьей группе объектов игрового пространства относится одно (а реже - несколько, как в игре CHESS) изображение, действиями которого вы можете управлять с помощью клавиатуры или джойстика. Так, в игре LEGEND OF KAGE это неустрашимый герой, преодолевающий множество препятствий и ведущий бескомпромиссную борьбу с многочисленными врагами-ниндзя. Персонаж игр серии BOULDER DASH - маленький забавный человечек, бегающий по головоломным лабиринтам и собирающий алмазы, за которым охотятся страшные бабочки, живые «бомбы» и кровожадные челюсти. Наконец, в играх OPERATION WOLF, STAR RADIERS II и во многих других «милитаристских» играх управляемым элементом является всего лишь перекрестие прицела.

Таким образом, если обобщить сказанное, то становится понятно, что любое сложное изображение на экране можно представить в виде набора небольших по размерам и вполне законченных объектов. Некоторые из этих объектов могут двигаться, а основная их масса составляет фоновый рисунок, на котором разворачивается действие игры. Все перечисленные объекты в компьютерных играх носят название *спрайты* (от английского слова Sprite - эльф, светлячок). Более точно спрайт можно определить как перемещаемый графический объект с неизменным рисунком и размером (см. [4]). Трудно представить хорошую игру без таких объектов (хотя, в некоторых играх, например, в MICRONAUT ONE или DARK SIDE, изображение создается программными средствами), поэтому, чтобы научиться писать приличные игровые программы, прежде всего нужно освоить технику создания спрайтов.

О том, как делать маленькие и большие спрайты, а также сложные пейзажи, мы расскажем дальше, а здесь только перечислим способы их изготовления. Небольшие спрайты вполне могут быть изготовлены вручную. Для этого спрайт рисуется на клетчатой бумаге (каждая клетка соответствует одной точке на экране), а затем изображение переводится в последовательность чисел. Значительно интереснее, а главное, эффективнее, создать рисунок будущего объекта в каком-либо графическом редакторе (например, в Art Studio или The Artist II), после чего автоматически закодировать его с помощью специальной программы, называемой генератором спрайтов, и записать на ленту или диск в виде готового к использованию спрайт-файла.

Все эти способы были подробно расписаны в [1], поэтому в настоящей книге мы не станем подолгу останавливаться на них и рассмотрим лишь некоторые особенности, имеющие непосредственное отношение к языку ассемблера.

ВЗАИМОДЕЙСТВИЕ С ИГРАЮЩИМ

Думается, ни для кого не секрет, в чем кроется причина популярности компьютерных игр. Ведь в них любой играющий с помощью несложных манипуляций может влиять на события, происходящие на экране и тем самым без особого вреда для здоровья почувствовать себя в роли супергероя или суперзлодея. И каким бы сложным ни был сюжет игры, устройства управления остаются одними и теми же - клавиатура или джойстик.

Приведем несколько примеров. В игре RIVER RAID можно ускорять или замедлять полет вашего самолетика, поворачивать его вправо или влево. И для этого достаточно лишь наклонить ручку джойстика в нужном направлении, а нажав кнопку «огонь», можно выпустить по объекту противника ракету.

Особенно поражают реальностью происходящего игры, посвященные какому-либо виду спортивной борьбы (FIST, ORIENTAL GAMES и другие). В них с помощью все тех же пяти клавиш или джойстика можно заставить главного персонажа выполнять множество различных приемов, прыжков, ударов и прочих телодвижений.

Подобные примеры можно продолжать до бесконечности, но для нас важно другое: все эти изменения на экране возможны благодаря особой части игровой программы, которая носит название *блок взаимодействия с играющим*. Когда вы нажимаете на какую-либо из управляющих клавиш или наклоняете ручку джойстика, программа реагирует должным образом и обычно переключается на выполнение той части, которая «заведует» выполнением соответствующего действия. Так, например, если ваш танк может поворачивать вправо и влево, а время от времени еще и стрелять, то в блоке управления такой программы должно быть ровно три части, каждая из которых отвечает за свой «участок работы».

У вас может создаться впечатление, что вся задача сводится лишь к тому, чтобы отслеживать нажатия клавиш или наклон ручки джойстика да в ответ на воздействие извне изменять направление движения управляемого объекта. Однако это пока еще не все. В реальной игре, когда по экрану движется созданный вами человечек, самолет или любой другой персонаж, возникает масса проблем, которые нужно решать. Прежде всего необходимо восстанавливать фон позади движущегося объекта, чтобы за ним не протянулся след девственно чистого экрана. А что делать, если движущийся объект «натолкнулся» на какое-то препятствие: стенку, дерево или встретился лицом к лицу со своим противником. Наконец, как поступить с

врагом, в которого попал снаряд, выпущенный из вашей пушки. На все эти вопросы мы постараемся дать ответ на страницах нашей книги.

ОЦЕНКА ИГРОВОЙ СИТУАЦИИ

Во время игры на экране появляются не только предметы и персонажи, определенные сюжетом, но еще и выводится различная информация, помогающая играющему разобраться в текущем положении. Например, в игре BOULDER DASH ведется постоянный подсчет собранных алмазов, а на экране показывается, сколько их еще нужно собрать до перехода на следующий уровень. Это очень важно, так как вам объявлена настоящая вендетта по-корсикански, а времени, чтобы уйти от погони, у вас в обрез (оставшиеся в вашем распоряжении секунды также высвечиваются на экране, чтобы безжалостная надпись «Out of time» не показалась слишком неожиданной). Поэтому, только забрав последний алмаз, вы можете вздохнуть с некоторым облегчением (правда, ненадолго).

В более сложных играх, таких как INTO THE EAGLES NEST, VIDEO POOL или OPERATION WOLF количество выводимой информации также становится более значительным. Кроме чисел на экране можно увидеть и тексты, поясняющие результаты ваших действий или предупреждающие о возникшей опасности (например, в SILENT SERVICE). Чаще всего такая информация выводится в виде «бегущей строки» или в специальном окне, предназначенном именно для этих целей (рис. 1.4). А во многих стратегических играх оценочная информация настолько обильна, что в них для вывода различных сообщений отводятся уже целые кадры, занимающие полный экран (скажем, как в играх SIM CITY или DICTATOR).

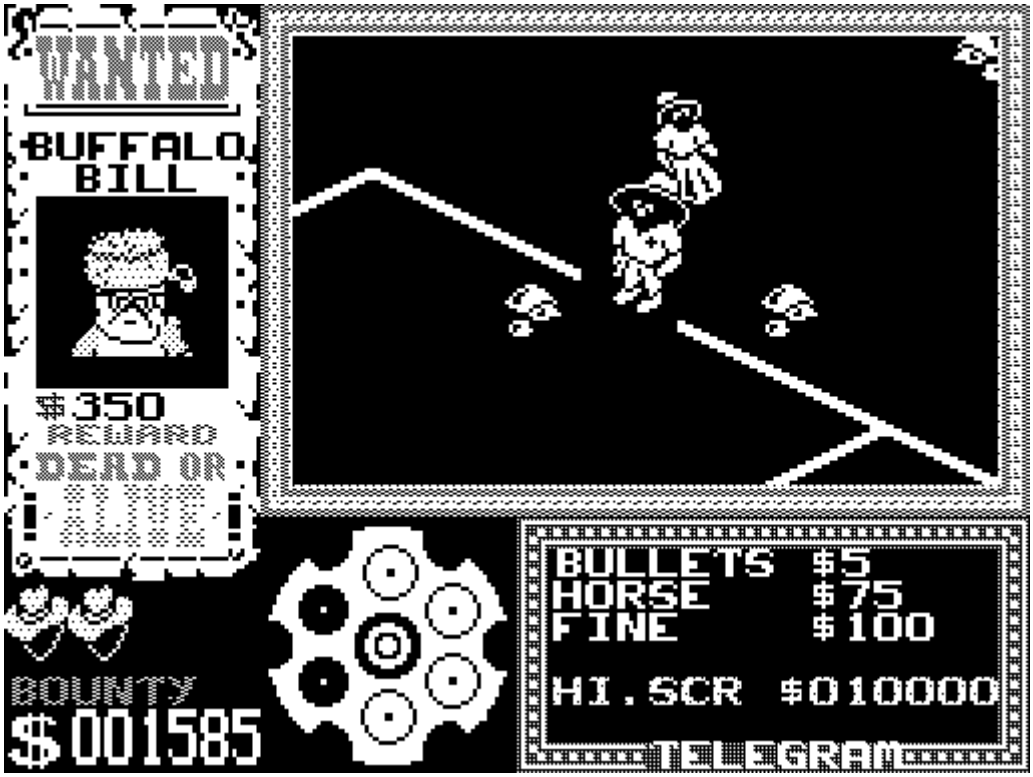


Рис. 1.4. Оценка игровой ситуации в игре GUN FRIGHT

Несмотря на невероятное разнообразие реализованных в игровых программах сюжетов, все же есть нечто, что объединяет выводимую на экран дополнительную информацию: она представляет собой *оценку игровой ситуации*. Следовательно, в программе должна присутствовать часть, которая оценивает действия играющего и его противников на всех стадиях игры и выводит на экран результаты этой оценки.

В большинстве игр ведется подсчет очков по тому или иному критерию, что позволяет легко выяснить, как со временем растет мастерство играющего или сравнить силу нескольких игроков. Очень часто выводится время, оставшееся до конца игры, число попаданий в противника, а также количество оставшихся у вас ресурсов (бомб, торпед, патронов). Все эти характеристики могут использоваться для оценки игровой ситуации и должны быть взяты вами «на вооружение» при создании собственных программ.

ЗВУКОВОЕ СОПРОВОЖДЕНИЕ ИГРЫ

Трудно даже назвать игру, тем более фирменную, где бы совсем не использовались те или иные звуки. Речь может идти лишь о том, что в одних играх музыкального сопровождения меньше, а в других каждое нажатие клавиши приводит ко все новым музыкальным шедеврам. Как образцы блестящего музыкального оформления стоит назвать такие игры как DEFINDER OF THE CROWN, PINK PONG, FLYING SHARK. Мы привели данный список с той мыслью, что, прочитав книгу до конца и накопив некоторый опыт, вам будет вполне по силам дизассемблировать эти игры и найти, а затем использовать лучшие образцы компьютерной музыки в своих программах. Ведь чего греха таить, не все, даже очень хорошие программисты, рождаются с музыкальными способностями.

В отличие от других частей игровой программы, выделить какой-то особый музыкальный блок довольно трудно, поскольку разные по сложности и назначению звуки формируются и в заставке, и в блоке взаимодействия с играющим, и даже в блоке оценки игровой ситуации. Тем не менее, все, что связано с музыкальным оформлением игры, можно рассматривать с точки зрения программирования как нечто, напоминающее отдельный блок. Все дело в том, что разные звуки и музыкальные фрагменты формируются одними и теми же подпрограммами и отличие может состоять лишь в размерах блоков данных (в особенности это справедливо для подпрограмм, создающих звуки в компьютере ZX Spectrum 128).

Говоря о звуках в компьютерных играх, никак нельзя пройти мимо возможностей музыкального сопроцессора, с помощью которого можно создавать не просто интересные звуковые эффекты, но и настоящие музыкальные произведения с полноценной оркестровой аранжировкой. На страницах этой книги мы постараемся поделиться своим опытом в данной области, подробно расскажем о программировании звуковых эффектов, сопровождая все это примерами ассемблерных программ с их подробным описанием.

ГЛАВА ВТОРАЯ,

из которой вы узнаете о том, что же такое ассемблер и чем он отличается от Бейсика и машинных кодов, а также усвоите некоторые основные понятия

Прежде чем приступать к изучению ассемблера, нужно усвоить несколько новых терминов, чтобы понимать, о чем вообще идет речь. Эту главу можно считать введением в совершенно новый для вас язык программирования (если же вы знакомы хотя бы с азами ассемблера, то можете лишь бегло пролистать эту и следующую главы, ибо они предназначены, в первую очередь, для новичков, делающих первые шаги в изучении машинного языка).

МАШИННЫЕ КОДЫ И АССЕМБЛЕР

Вы уже, вероятно, имеете некоторое представление о том, что такое машинные коды, а если нет, то мы сейчас постараемся объяснить это, хотя бы немного, что называется, «на пальцах», не вдаваясь в теорию проектирования компьютерной техники.

Нетрудно догадаться, что микропроцессор, будучи куском железа, пусть даже довольно интеллектуального, но все же железа, не способен понимать слова человеческого языка, а тем более - складывать отдельные слова в осмысленные фразы. Как и всякая порядочная электронная машина, он может воспринимать только электрические заряды. Но в отличие от пылесоса микропроцессор принимает заряды одновременно с восьми контактов (строго говоря, микропроцессор обрабатывает гораздо больше сигналов, но сейчас нас интересуют только восемь из них) и в зависимости от поступившего сигнала выполняет то или иное действие.

У каждого из этих восьми контактов может быть лишь два состояния - есть заряд или нет заряда. Поэтому его наличие можно представить как 1, а отсутствие - как 0. Последовательности из единиц и нулей дают числа в двоичном представлении, но их несложно перевести в привычный десятичный формат. Напомним, как это делается на примере числа 00111100. Самый младший разряд (то есть, крайнюю правую цифру) умножаем на 1, второй разряд - на 2, третий - на 4, следующий - на 8 и так далее. Иными словами, значение каждого разряда умножается на 2 в степени n, где n - номер разряда, который может изменяться от 0 до 7 (то есть, говоря научно, 2 - это основание системы счисления). Если вам не очень понятно такое определение, воспользуйтесь простой формулой для перевода нашего двоичного числа в десятичное:

```
00111100=0×128+0×64+1×32+1×16+1×8+1×4+0×2+0×1=60
```

Точно так же можно перевести в десятичное и любое другое число, представленное в двоичном формате. Попутно напомним, что разряды двоичных чисел в информатике принято называть битами, а последовательности из 8 битов составляют байты.

Теперь вы вправе спросить, а что же будет, если микропроцессору дать такую команду? В ответ мы напишем похожую строку на хорошо известном вам Бейсике:

```
LET A=A+1
```

то есть содержимое переменной A увеличивается на единицу. И если последнюю запись понять не так уж трудно, то на свете найдется не так уж много людей, способных не только воспринимать, но и писать достаточно сложные программы, оперируя голыми числами, да еще двоичными. Простой и логичный выход из создавшегося затруднения - заменить все коды машинного языка человеческими словами или, хотя бы сокращениями, поставив каждой

команде микропроцессора в соответствие единственное обозначение. Именно такой язык и был назван ассемблером. Он стоит всего лишь на одну ступеньку выше машинных кодов, однако общаться с компьютером на таком языке несравненно проще, чем на языке цифр. Приведенная выше комбинация единиц и нулей 00111100 на ассемблере будет выглядеть так:

```
INC   A
```

где INC - сокращение от английского слова increase (увеличиваться). Сразу же скажем, что сокращенные имена команд микропроцессора называют *мнемониками*. Запомните это слово хорошенько, так как оно не раз встретится в нашей книге.

ЧТО МОЖЕТ МИКРОПРОЦЕССОР Z80?

Поскольку ассемблер непосредственно связан с машинными командами, то начинать его изучение будет резонно с вопроса «а что же может микропроцессор?» Так вот, если вы считаете, что он способен играть музыку, рисовать картинки или печатать текст, то глубоко заблуждаетесь. Ничего такого микропроцессор не умеет. Он может выполнять лишь самые элементарные действия вроде «2 + 2», а более сложным, таким как «2 × 2», его еще нужно научить. В этом и состоит задача программиста. Но у микропроцессора есть одно преимущество - за одну секунду он способен выполнить многие тысячи операций, поэтому в реальном времени он и кажется достаточно одаренным.

Вот краткий и не совсем полный список операций, доступных микропроцессору:

- простейшие арифметические действия сложения и вычитания;
- операции с памятью, такие как запись в определенную ячейку или считывание из памяти чисел (подобно `POKE` и `PEEK` в Бейсике);
- связь с внешними устройствами через порты (то, чем занимаются в Бейсике `OUT` и `IN`);
- обработка отдельных битов (разрядов двоичных чисел) (напомним, что в Бейсике можно задавать битовые константы с помощью ключевого слова `BIN`);
- логические операции с двоичными числами;
- различные вызовы других подпрограмм;
- условные и безусловные переходы;
- работа с прерываниями (это средство, совершенно недоступное Spectrum Бейсику, будет обсуждаться в отдельной главе).

Вам может показаться странным, что компьютер, располагая столь незначительными средствами, умудряется создавать на экране целые миры, исполнять сложные музыкальные произведения и даже управлять какими-то механизмами вроде принтера. Чтобы прояснить, как это получается, приведем маленький примерчик на Бейсике, построенный по тому же принципу, что и большинство программ в машинных кодах:

```
10 LET ADDR1=16384
20 LET ADDR2=15880
30 LET N=8
40 LET A=PEEK (ADDR2)
50 POKE (ADDR1),A
60 LET ADDR1=ADDR1+256
70 LET ADDR2=ADDR2+1
80 LET N=N-1
90 IF N<>0 THEN GO TO 40
```

Выполнив эту программку, вы увидите в верхнем левом углу экрана букву **A**. Наверное, нет большой необходимости подробно расписывать, как работает приведенный пример, тем не менее, кратко поясним, что же здесь происходит.

В переменную **ADDR1** помещаем адрес (напоминаем, что адресом называется порядковый номер байта в памяти; в ZX Spectrum адреса имеют номера от 0 до 65535) начала экранной области памяти, а переменная **ADDR2** указывает на начало данных, находящихся в ПЗУ и описывающих внешний вид символа **A**. В данном примере адрес **ADDR2** рассчитан заранее, хотя обычно все вычисления возлагаются на программу. Далее в цикле последовательно считываются 8 байтов, составляющих символ, и переносятся на экран. При этом переменная **ADDR1** изменяется с шагом 256, что обеспечивает заполнение одного знакоместа (чуть позже мы подробно остановимся на строении экрана и методах вычисления его адресов, а пока примите это как данность). Обратите внимание на способ организации цикла в этом примере. С точки зрения Бейсика вся эта программка выглядит довольно неказисто, но зато она довольно точно отражает последовательность действий микропроцессора при выполнении аналогичной задачи.

Вообще-то все на самом деле выглядит несколько сложнее, а здесь мы продемонстрировали лишь принцип работы одного из самых популярных операторов Бейсика - оператора `PRINT`. Но пусть вас это не пугает, ведь процедура вывода символов на экран уже имеется в компьютере, и совершенно не обязательно воспроизводить ее еще раз в своей собственной программе. Достаточно знать, как ее можно вызвать - и значительная часть проблем отойдет в сторону.

Однако, не будем забегать вперед, а прежде разберемся до конца с темой этой главы.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ ЯЗЫКА АССЕМБЛЕРА

Бейсик, Паскаль, Си и подобные им языки относятся к так называемым языкам высокого уровня. Это означает, что операторы, составляющие их словарный запас, по сути своей являются целыми программами, написанными, как правило, на языке низкого уровня - на изучаемом нами ассемблере, а по сути, в машинных кодах, которыми «говорит» компьютер. Что касается ассемблера, то он дает возможность писать программы на среднем или низком уровне (точнее, почти на самом низком, как мы уже говорили). Разница между средним и низким уровнем достаточно условная и заключается в том, что первый в полную силу использует возможности операционной системы, обращаясь к готовым подпрограммам, «зашитым» в ПЗУ, а второй - нет. Выбор уровня зависит от целей программиста, у каждого из них есть свои преимущества и недостатки, а весь этот разговор мы затеяли лишь затем, чтобы помочь вам сориентироваться в таком выборе.

Итак, что же мы теряем и что приобретаем с переходом к более низким уровням программирования?

Преимущества:

- значительное увеличение скорости выполнения программ;
- большая гибкость (отсутствуют рамки Бейсика, независимость от операционной системы, более оптимально используются возможности компьютера);
- полученные программы занимают меньше памяти.

Недостатки:

- программы требуют больше времени и внимательности при написании;
- сложность отладки (отсутствуют привычные сообщения об ошибках, текст трудно читать);
- трудно выполнять арифметические действия (микропроцессор не может обрабатывать дробные числа, да и применение целых чисел имеет ряд ограничений).

Чтобы понять, почему программы, написанные на ассемблере, обычно работают во много раз быстрее, давайте посмотрим, какими методами пользуются интерпретаторы (например, тот же Бейсик) и компиляторы.

Метод, используемый интерпретаторами можно сравнить с переводом со словарем. Микропроцессор последовательно считывает текст программы слово за словом, оператор за оператором, затем лезет в специальную таблицу, содержащую имена команд и адреса подпрограмм, выполняющих заданное действие. И только после того, как весь оператор прочитан до конца, начинается его исполнение. Не увеличивает скорость перевода еще и то, что у компьютера весьма «короткая память», и надо за каждым словом вновь и вновь лезть в словарь, даже если это слово только что встречалось.

Несколько быстрее работают компиляторы. Полученные с их помощью программы можно сравнить с подстрочником, составленным довольно неумелым переводчиком, поэтому микропроцессору над каждой фразой приходится еще поломать голову, что же хотел сказать этим автор. (Если быть более точным, компилятор каждую фразу исходного языка заменяет кусочком машинного кода, а то, как эффективно он это делает, зависит от авторов данного компилятора - *Примеч. ред.*) Кроме того, большинство компиляторов имеет дурную привычку «навешивать» на программу воз и маленькую тележку совершенно никому не нужного хлама, что при 48K максимальной свободной памяти кажется, мягко говоря, несколько расточительным.

Что же касается машинных кодов, то это родной язык компьютера, и совершенно естественно, что программу на таком языке микропроцессор может выполнить в самые кратчайшие сроки - ведь в этом случае не приходится прибегать к услугам переводчиков. Безусловно, и тут при желании можно «загнуть» такую заумную фразу, которая надолго оставит компьютер в недоумении, но это уже будет на совести автора программы.

Конечно, умение писать на ассемблере не означает полный отказ от Бейсика и других языков высокого уровня, особенно на первых порах. Поэтому мы ставим цель прежде всего научить вас создавать коротенькие фрагменты, позволяющие значительно обогатить игры и придать им динамичность. Большинство предлагаемых в этой книге подпрограмм построено по принципу широкоизвестного набора процедур в машинных кодах под названием **Supercode**. Причем некоторые из предлагаемых примеров будут работать в «тандеме» с программами на Бейсике.

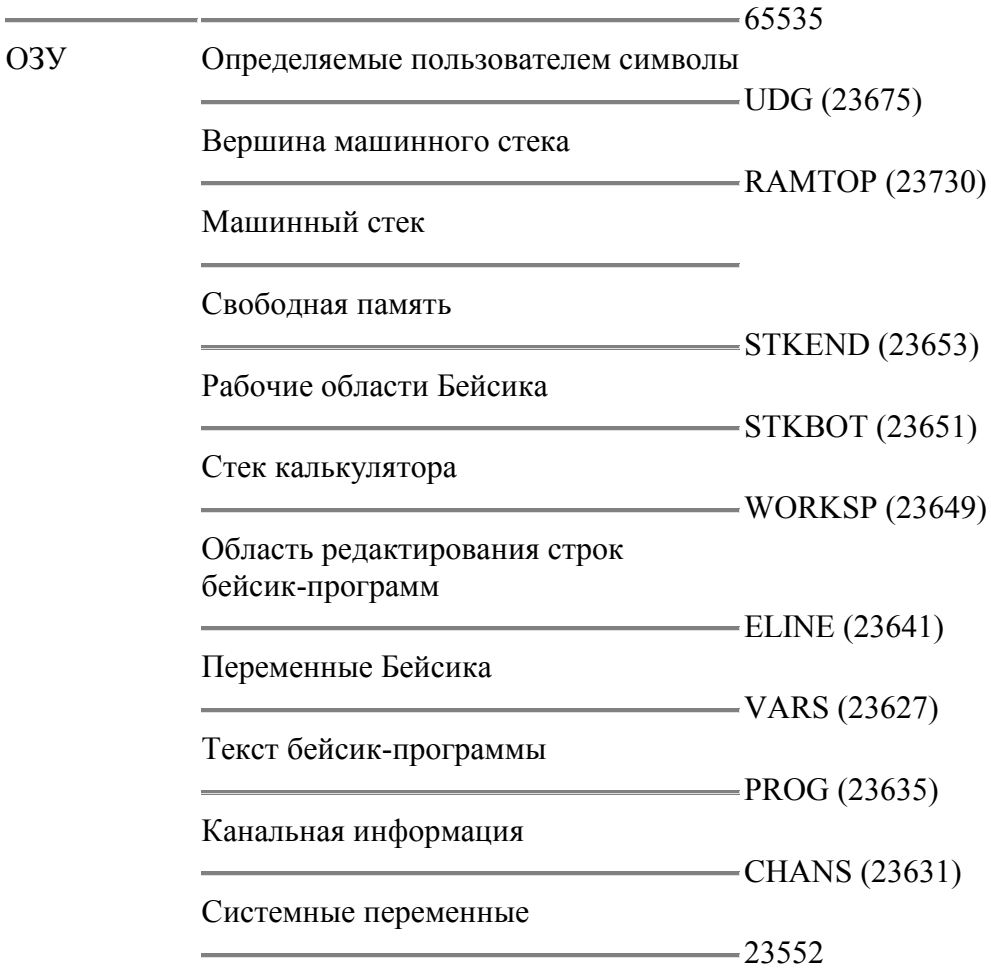
ОРГАНИЗАЦИЯ ПАМЯТИ

При создании программ на ассемблере вы в той или иной степени лишаетесь опеки операционной системы и вынуждены самостоятельно следить за размещением в памяти кодов программы, переменных, массивов и различных рабочих областей, ежели таковые потребуются. Отчасти подобные проблемы уже могли вставать перед вами, если в своих

программах на Бейсике вы использовали дополнительные шрифты или процедуры из пакетов Supercode и NewSupercode. Но в Бейсике задача по размещению кодов решается довольно просто - нужно только опустить RAMTOP чуть ниже адреса загрузки кодового блока, выполнив оператор CLEAR. Когда же вы начнете программировать на ассемблере, то во многих случаях этого окажется недостаточно. Поэтому необходимо четко представлять, как распределяется память между различными областями, а также какие области памяти вообще существуют и для чего они предназначены. Не обойтись и без знания строения некоторых из них. Например, для успешной обработки изображений (скажем, вывода спрайтов, скроллинга окон и т. п.) нужно уметь по координатам экрана быстро определять адрес соответствующего байта в видеобуфере. Именно этим вопросам и будет посвящен данный раздел. Не вдаваясь в подробности сразу скажем, что описываемое здесь распределение памяти будет одинаково и для стандартной конфигурации Speccy с 48 килобайтами оперативной памяти, и для ZX Spectrum 128 со 128К, и даже для таких монстров, у которых ОЗУ занимает 256 или 512К.

ПЗУ и ОЗУ

Вся память компьютера делится на две основные области: постоянное запоминающее устройство (ПЗУ) и оперативная память (ОЗУ). ПЗУ начинается с адреса 0 и содержит коды операционной системы Бейсик. В этой области памяти ничего нельзя изменить, все, что там записано, сохраняется и при выключенном питании компьютера. Тем не менее, в ПЗУ имеется множество полезных подпрограмм, которыми мы в дальнейшем будем пользоваться с большим успехом, кроме того, мы часто будем обращаться к кодам *знакогенератора*, расположенным в самых последних ячейках ПЗУ, начиная с адреса 15616, и представляющим собой полный набор символов, печатаемых на экране. Простирается постоянная память вплоть до адреса 16384, с которого начинается область ОЗУ (рис. 2.1).



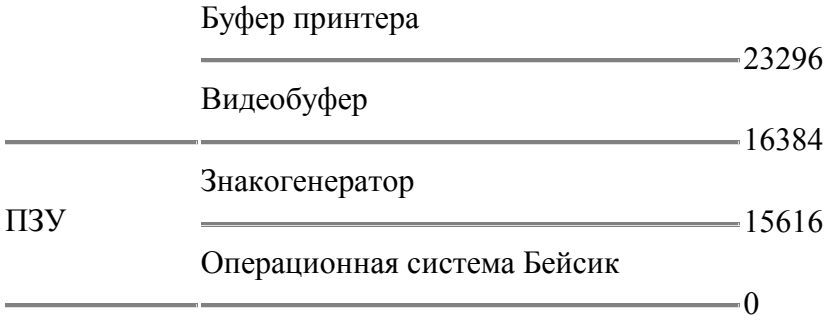


Рис. 2.1. Распределение областей памяти

Если в ПЗУ все уже предопределено, то оперативная память служит для временного хранения и обработки информации. Это могут быть различные программы на Бейсике или в машинных кодах, текстовые файлы, блоки данных и т. п. Все программы, которые приведены в книге, должны размещаться именно в оперативной памяти.

Для успешного программирования в машинных кодах и на ассемблере нужно четко представлять, какие области оперативной памяти для каких целей служат. Наиболее важной из них является видеобуфер, так как никакими программными средствами невозможно изменить его местоположение, размер или строение. Экранная память начинается с адреса 16384 и занимает 6912 байт. Вся остальная память, с адреса 23296 и до 65535 включительно, находится в вашем безраздельном распоряжении. Правда, это только в том случае, если вы создаете программу, полностью независимую от операционной системы компьютера. Но пока программа не заблокирует систему, вы не можете нарушать содержимого и структуры некоторых областей, о назначении и расположении которых вы должны быть хорошо осведомлены, начиная программировать на ассемблере.

Системные области

Сразу за видеобуфером следует небольшая область памяти, называемая буфером принтера. Она используется только при работе с различными печатающими устройствами, поэтому если вы не предполагаете в своей программе делать какие-то распечатки на бумаге, смело можете занимать память в диапазоне адресов от 23296 до 23551 включительно (то есть 256 байт) под любые нужды. Во всяком случае, буфер принтера может использоваться для временного хранения информации или как рабочий массив.

Однако нужно помнить, что все сказанное о буфере принтера справедливо лишь для стандартной конфигурации компьютера, то есть для ZX Spectrum 48. Если вы пишете программы, которые должны работать на модели Spectrum 128 или Scorpion ZS 256, то столь произвольно обращаться с этой областью памяти нельзя, потому как в данных адресах указанные модели содержат жизненно важную информацию, при разрушении которой компьютер не сможет нормально продолжать работу (хотя, разумеется, можно выполнить программу и в режиме «эмуляции» обычного Speccy - *Примеч. ред.*).

С адреса 23552 начинается наиболее важная из системных областей. Вы уже частично (а может быть, и полностью) знакомы с системными переменными Бейсика. В различных ячейках этой области хранится различная информация о текущем состоянии всех без исключения параметров операционной системы, в том числе и информация о расположении всех прочих областей памяти, которые не имеют жесткой привязки к конкретным адресам

(описание всех системных переменных Spectrum-Бейсика, а также ZX Spectrum 128 и TR-DOS можно найти в [2]).

Системная переменная CHANS, находящаяся в ячейках 23631 и 23632, адресует область канальной информации, содержащей необходимые сведения о расположении процедур ввода/вывода (напоминаем, что на первом месте стоит младший байт адреса и для перевода двухбайтового значения в число требуется содержимое старшего байта умножить на 256 и прибавить к нему число из младшего байта; например, для определения значения переменной CHANS нужно выполнить команду `PRINT PEEK 23631+256*PEEK 23632`). Далее следует область, хранящая текст бейсик-программы. Ее начальный адрес содержит переменная PROG (23635/23636). Сразу за бейсик-программой располагаются переменные Бейсика. Их начало можно определить, прочитав значение системной переменной VARS по адресу 23627/23628. После переменных Бейсика расположена область, предназначенная для ввода и редактирования строк бейсик-программ. Ее адрес записан в системной переменной E_LINE (23461/23642). За областью редактирования строк находится рабочая область Бейсика WORKSP (23649/23650), предназначенная для самых разных нужд. Сюда, например, считываются заголовки файлов при загрузке программ с ленты, там же размещаются строки загружаемой оператором MERGE программы до объединения их со строками программы в памяти и т. д.

Следом за областью WORKSP расположена весьма важная область, называемая *стеком калькулятора*. Название говорит само за себя: сюда записываются числовые значения, над которыми производятся различные математические операции, здесь же остается до востребования и результат расчетов. В дальнейшем мы не раз будем прибегать к помощи этой области, так как многие процедуры операционной системы, которыми мы будем пользоваться, берут параметры именно отсюда. Системная переменная STKBOT (23651/23652) указывает на начало стека калькулятора, а STKEND (23653/23654) - на его вершину. Иногда бывает важно учитывать, что каждое значение, заносимое на вершину стека калькулятора, имеет длину 5 байт.

Системная переменная RAMTOP (23730/23731) указывает на местоположение в памяти еще одной важной области - *машинного стека* (не путайте со стеком калькулятора!). Но надо помнить, что в ассемблерных программах стек вполне может потерять всякую связь с RAMTOP, ибо он не является неотъемлемой частью бейсик-системы, а скорее уж, находится в «собственности» микропроцессора. Вообще же стек - это удивительно удобная штука для временного хранения различной информации, потому как при его использовании не приходится запоминать, где, по какому адресу или в какой переменной находится то или иное число. Важно лишь соблюсти очередность обмена данными, а чтобы не нарушить установленный порядок, следует знать, по какому принципу работает стек. Этот принцип часто называют «Last In, First Out» (LIFO), что значит «Последним вошел, первым вышел». Совсем как в автобусе в час пик - чтобы выпустить какого-нибудь пассажира, прежде должны выйти все вошедшие за ним. Поэтому данные, которые понадобятся в первую очередь нужно заносить в стек последними (это же, кстати, в полной мере относится и к порядку обмена данными со стеком калькулятора).

Говоря о машинном стеке, нужно отметить один довольно интересный факт. В отличие от способа организации других областей памяти (а также и от стека калькулятора) он растет «головой вниз», то есть каждое следующее значение, отправленное в стек, будет располагаться по адресу на 2 байта ниже предыдущего (машинный стек работает только с двухбайтовыми величинами). Поэтому вас не должны вводить в заблуждение такие выражения как «Положить значение на вершину стека» или «Снять значение с вершины стека» - эта самая «вершина» всегда будет не выше основания.

Существуют и другие области памяти, как то: UDG, системные переменные TR-DOS или карта микродрайва. Область определяемых пользователем символов UDG мы рассмотрим в следующих главах, а о других разделах памяти (в том числе и об архитектуре Spectrum 128) вы можете получить дополнительные сведения, например, в книге [2].

Строение экрана

Как мы уже говорили, важность знания структуры экранной области трудно переоценить. Умение быстро рассчитывать адрес в видеобуфере по координатам экрана понадобится нам в большинстве графических построений, особенно, если вы хотите научиться обходиться без опеки операционной системы, которая почти всегда оказывается не достаточно быстродействующей.

Строение экрана «на высоком уровне» вам уже должно быть известно (рис. 2.2), но тем не менее, напомним, из каких частей он состоит. Внешняя область, называемая *бордюром*, может только изменять свой цвет, никакую графическую информацию, за исключением быстро бегущих по нему полос, в эту область поместить невозможно. Внутри бордюра находится *рабочий экран*, сюда может быть выведена любая текстовая или графическая информация. Говоря об экране, мы всегда будем подразумевать именно эту его часть. Рабочий экран в свою очередь делится на *основной экран* и *служебное окно*, которое обычно занимает две нижние строки, но в некоторых случаях может увеличиваться или уменьшаться. Всего экран имеет 24 текстовые строки, и в каждой строке можно напечатать 32 символа. Эти стандартные площадки для вывода символов называются *знакоместами*. Любое изображение на экране состоит из маленьких квадратиков, называемых *пикселями*, и каждое знакоместо имеет размеры 8×8 таких элементарных «точек».

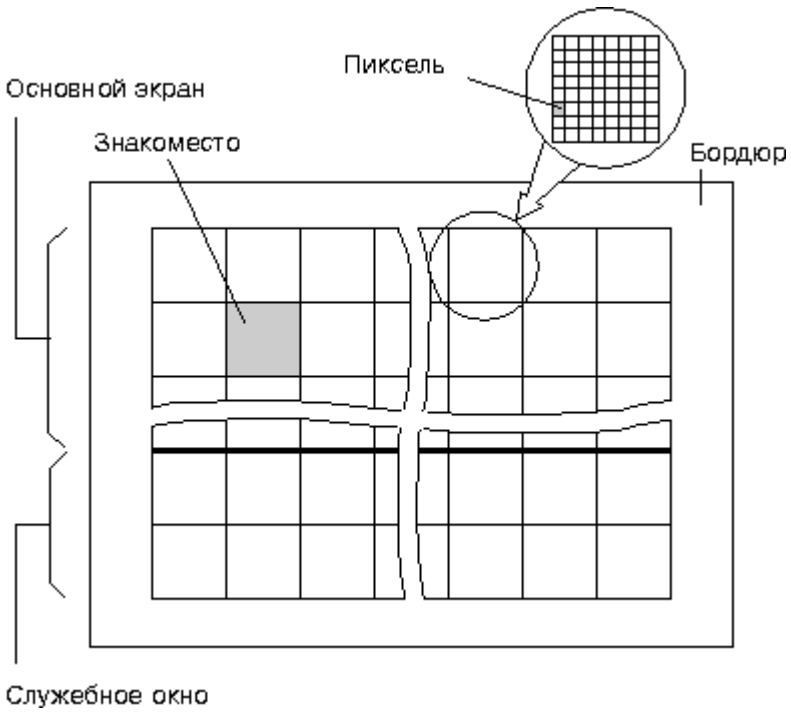


Рис. 2.2. Строение экрана

Теперь перейдем к более низкому уровню и посмотрим, как адресуется область видеобуфера. Вы, наверное, не раз наблюдали, как грузятся с магнитофона стандартные экранные файлы: область экрана заполняется не последовательно, строка за строкой, а довольно хитрым

способом. Сначала один за другим появляются верхние ряды пикселей восьми первых текстовых строк, затем в этих же строках рисуются все вторые ряды и так далее, пока не сформируется изображение всей верхней трети экрана. Затем, в том же порядке, заполняется средняя часть экрана, а потом и нижняя. И только в самом конце последовательно выводятся атрибуты всех знакомест. Однако это говорит не о каком-то изощренном способе загрузки именно экранных файлов - они грузятся так же, как и любые другие, последовательно заполняя ячейки памяти от младших адресов к старшим. Это свидетельствует, напротив, о нелинейном строении экранной области памяти.

На первый взгляд такая организация экранной области кажется исключительно неудобной, особенно при решении задач определения адреса по заданным координатам. Но это лишь до тех пор, пока вы используете в расчетах только десятичные числа. Ведь даже начальный адрес видеобуфера 16384 в десятичном виде представляется просто «взятым с потолка». В таких случаях гораздо удобнее пользоваться несколько иным представлением числовой информации. Мы имеем в виду шестнадцатеричный формат чисел, который от десятичного отличается «емкостью» разрядов.

В десятичном представлении каждый разряд числа может изменяться от 0 до 9, а в шестнадцатеричном - от 0 до 15. Цифры от 0 до 9 при этом записываются так же, как и в десятичных числах, а дальше применяются буквы латинского алфавита от A до F. Вот соответствие чисел в десятичном и шестнадцатеричном форматах:

```
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18...
0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F 10 11 12...
```

Если адрес 16384 привести к шестнадцатеричному виду, то он вдруг окажется совершенно «ровным» - #4000 (знак # перед числом говорит о том, что оно представлено как шестнадцатеричное). Взгляните на схему, изображенную на рис. 2.3 и вы заметите явную закономерность в распределении адресного пространства видеообласти. В недалеком будущем мы подробно расскажем о методах вычисления адресов экрана, а сейчас перейдем к рассмотрению других не менее важных понятий.

Данные	Атриб.	Строка	ВИДЕОБУФЕР	Строка	Данные	Атриб.
#4000	#5800	0		0	#401F	#581F
#4020	#5820	1		1	#403F	#583F
#4040	#5840	2		2	#405F	#585F
#4060	#5860	3		3	#407F	#587F
#4080	#5880	4		4	#409F	#589F
#40A0	#58A0	5		5	#40BF	#58BF
#40C0	#58C0	6		6	#40DF	#58DF
#40E0	#58E0	7		7	#40FF	#58FF
#4800	#5900	8		8	#481F	#591F
#4820	#5920	9		9	#483F	#593F
#4840	#5940	10		10	#485F	#595F
#4860	#5960	11		11	#487F	#597F
#4880	#5980	12		12	#489F	#599F
#48A0	#59A0	13		13	#48BF	#59BF
#48C0	#59C0	14		14	#48DF	#59DF
#48E0	#59E0	15		15	#48FF	#59FF

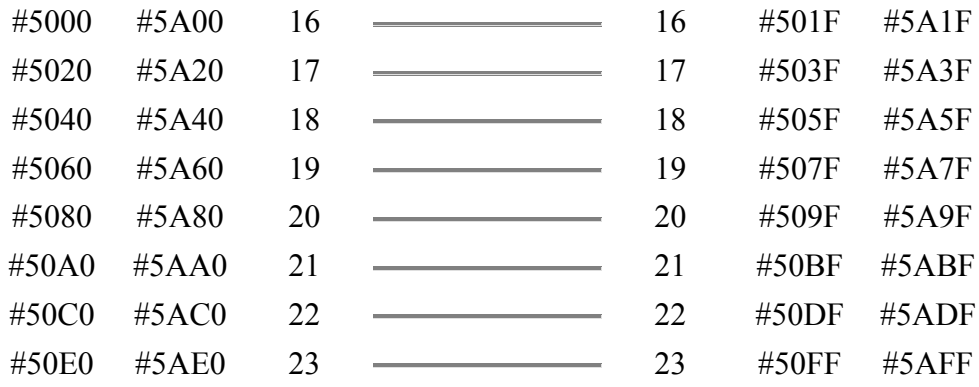


Рис. 2.3. Адресация видеобуфера

РЕГИСТРЫ И РЕГИСТРОВЫЕ ПАРЫ

Для общения с компьютером на уровне машинных кодов необходимо усвоить еще одно новое понятие кроме таких, как память, адрес, байт, бит, с которыми, надеемся, вы уже достаточно неплохо разобрались. Речь идет о *регистрах* микропроцессора. Регистры можно представить как совершенно особые внутренние ячейки памяти, являющиеся неотъемлемой частью центрального процессора. Роль их настолько важна, что практически ни одна операция не обходится без участия регистров, а различные арифметические и логические действия без них и вовсе невозможны.

Мы назвали регистры особыми ячейками, но в чем же их особенность и чем они отличаются от ячеек обычной оперативной памяти? В первую очередь их особенность проявляется в том, что регистры не равноценны, то есть действия, допустимые с использованием одного регистра невозможны с другими и наоборот. Кроме того, если значения одних регистров можно изменять непосредственно, записывая в них те или иные числа, то другие изменяются автоматически, и узнать их содержимое возможно только лишь косвенными методами.

Другая особенность регистров состоит в том, что для обращения к ним используются не адреса, а собственные имена, состоящие из одной или двух букв латинского алфавита (конечно же, имена присутствуют только в языке ассемблера, а не в машинных кодах команд).

Есть и еще одно свойство, отличающее регистры от ячеек памяти - это способность их объединяться определенным образом, составляя *регистровые пары*. Во всем же остальном они очень схожи с отдельными ячейками памяти компьютера. Они также имеют размер байта (8 бит), в них можно записывать числа и читать их значение (за исключением системных регистров), информация в них может сохраняться, как и в памяти, до тех пор, пока не будет изменена программой.

Все регистры могут быть подразделены на несколько групп, учитывая характер функций, которые они выполняют. Начнем с самой многочисленной и наиболее важной группы - с так называемых *регистров общего назначения* или *регистров данных*. Их насчитывается семь: A, B, C, D, E, H и L. Как уже говорилось, каждый регистр может использоваться лишь в строго определенных операциях и каждый из них в этом смысле уникален. Например, регистр A (часто называемый аккумулятором) участвует во всех арифметических и логических операциях, результат которых мы получаем в том же регистре A. Использование регистра B наиболее удобно при организации циклов. При этом он выполняет роль, схожую с

обязанностями управляющих переменных циклов FOR...NEXT в Бейсике. Другие регистры проявляют свою индивидуальность, преимущественно, объединившись в пары. Возможны следующие регистровые пары: BC, DE и HL. И вам следует запомнить, что никаких других вариантов соединения регистров не существует.

Из сказанного может создаться впечатление, что аккумулятор остался в одиночестве, не найдя своей половинки. Однако это не совсем так. На самом деле существует пара и для него. Просто пока мы сознательно умалчиваем об этом, так как регистр, дополняющий аккумулятор до пары, имеет совершенно особый статус и заслуживает отдельного разговора, который мы поведем в разделе «Организация циклов в ассемблере» главы 5.

Каждая из регистровых пар так же, как и любой из отдельных регистров, выполняет вполне конкретные, возложенные именно на нее функции. Так пара BC часто используется, подобно регистру B в качестве счетчика в циклах. HL несет наибольшую нагрузку, играя примерно ту же роль, что и аккумулятор: только с этой парой можно выполнять арифметические действия. Пара DE зачастую адресует пункт назначения при перемещениях данных из одной области памяти в другую.

Работая с регистровыми парами, приходится иметь дело с двухбайтовыми величинами. Поэтому необходимо четко представлять, как такие числа хранятся в памяти и каким образом они размещаются на регистрах. В Бейсике вам, вероятно, уже доводилось сталкиваться с подобной задачей. Если вы пользовались оператором POKE и функцией PEEK, например, для изменения или чтения системных переменных, то вам уже должно быть известно, что двухбайтовые значения хранятся в памяти, как правило, в обратном порядке - сначала младший байт, затем старший. Это можно продемонстрировать на таких примерах: число 1 запишется в памяти в виде последовательности байтов 1 и 0; у числа 255 старшая часть также равна нулю, поэтому оно будет представлено как 255 и 0; следующее число 256, расположившись в двух ячейках, будет выглядеть как 0 и 1. На всякий случай напомним вам способ, позволяющий разложить любое число из диапазона 0...65535 на два байта и определить значения старшей и младшей половинки:

```
LET high= INT(N/256): REM Старшая часть
LET low=N-256*high: REM Младшая часть
```

Вам также часто придется сталкиваться с необходимостью изменять только старший или только младший регистр в регистровых парах. Поэтому следует хорошенько запомнить правило, которому подчиняются регистры при объединении. Оказывается, порядок здесь прямо противоположный по сравнению с числами в памяти - первым записывается старший регистр, а за ним младший. То есть в паре BC старшим окажется регистр B, в DE - D, а в HL - H. Чтобы лучше запомнить это, можете представить имя регистровой пары HL как сокращения английских слов HIGH (высокий, старший) и LOW (низкий, младший), а то, что порядок следования старшей и младшей половинок в остальных парах аналогичен, это уже само собой разумеется.

К следующей группе относятся два индексных регистра, имена которых начинаются с буквы I (Index) - IX и IY. В отличие от регистров данных, индексные регистры состоят из 16 разрядов, то есть являются как бы неделимыми регистровыми парами. (На самом деле существуют методы разделения индексных регистров на 8-разрядные половинки, что уже относится к программистским изощрениям. Об этих методах вы можете узнать из Приложения II.) В основном они применяются при обработке блоков данных, массивов или разного рода таблиц, но также вполне могут использоваться и как обычные регистры общего назначения. Удобство употребления этих регистров заключается в том, что они позволяют обратиться к любому элементу массива или таблицы без изменения содержимого самого регистра, а лишь указанием величины смещения для данного элемента (иначе, его номера или

индекса, например, IX+5). Заметим, что регистр IY обычно адресует область системных переменных Бейсика и поэтому отчасти и только в компьютерах ZX Spectrum может быть отнесен к следующей группе - системным регистрам.

К системным или иначе - аппаратным регистрам относятся: указатель вершины стека SP (Stack Point), вектор прерываний I (Interrupt) (точнее, этот регистр содержит старший байт адреса векторов прерываний; позднее мы подробно расшифруем это понятие) и регистр регенерации R. Первый из них, так же, как и индексные регистры, имеет 16 разрядов, разделить которые на 8-битовые половинки нет никакой возможности. Но это и не нужно, ведь регистр SP служит для вполне определенных целей - указывает адрес вершины области машинного стека, как это и следует из его названия. Хотя с ним и можно обращаться, как с обычным регистром данных (записывать или читать из него информацию), но делать это нужно, совершенно точно представляя, что при этом происходит. Обычно же за регистром SP следит микропроцессор и изменяет его так, как надо при выполнении некоторых команд. Например, без этого регистра оказались бы совершенно невозможны вызовы подпрограмм с нормальным возвратом из них в основную программу.

Регистры I и R, в противоположность всем прочим, никогда не объединяются в пары и существуют только по отдельности. Содержимое вектора прерываний I также может быть изменено программным путем, однако делать этого не стоит до тех пор, пока вы не разберетесь с таким достаточно сложным вопросом, как прерывания. Что же касается регистра R, то читать из него можно, а вот записывать в него информацию в большинстве случаев бесполезно, так как он изменяется аппаратно. Правда, используется для аппаратных нужд только 7 младших разрядов, так что, если вам для чего-то окажется достаточно одного бита, можете хранить его в старшем разряде регистра регенерации.

В свое время мы подробно расскажем о применении всех существующих регистров, а сейчас закончим этот краткий обзор и займемся другими вопросами.

ПРОСТЕЙШАЯ ПРОГРАММА В МАШИННЫХ КОДАХ

Как мы уже говорили, ассемблер невысоко поднялся над машинным языком, поэтому, прежде чем бросаться с головой в программирование, будет полезно составить хотя бы несколько коротеньких программок чисто в машинных кодах. Это позволит гораздо лучше прочувствовать идеологию ассемблера и понять, как работает микропроцессор.

Большинство программ на машинном языке, имеющих возврат в Бейсик, заканчивается кодом 201 (в мнемоническом обозначении - RET), который аналогичен оператору `RETURN`. Поэтому простейшая программа может состоять всего из одного байта. Давайте сейчас создадим такую программу, а потом и выполним ее. Введите с клавиатуры оператор

```
POKE 40000,201
```
а затем, чтобы проверить действие полученной «программы», запустите ее с помощью функции `USR`, например, так:

```
RANDOMIZE USR 40000
```
Вроде бы ничего особенного не произошло - никаких видимых или слышимых эффектов. Но, по крайней мере, ваш компьютер выдержал подобное испытание и при этом не «завис» и не

«сбросился» (если он исправен, конечно, на что мы надеемся). Программа нормально завершила свою работу и благополучно вышла в Бейсик с сообщением `0 OK`.

Теперь попробуем запустить ее несколько иным способом. Заменим оператор `RANDOMIZE` на `PRINT`:

```
PRINT USR 40000
```

То, что вы увидели на экране - весьма существенно и может очень пригодиться в будущем. Компьютер напечатал то же самое число, которое мы использовали в качестве аргумента функции `USR`. Можете проверить, что это не случайное совпадение, введя строку

```
POKE 40001,201: PRINT USR 40001
```

Можете попробовать проделать то же самое и с другими адресами, только не слишком увлекайтесь, чтобы не залезть в «запрещенные» области памяти. Для подобных экспериментов лучше не выходить из диапазона адресов от 30000 до 60000, да и то лишь в том случае, если память компьютера свободна от каких-либо других программ.

Для того, чтобы каким-то образом использовать полученный результат, необходимо понять причину такого странного поведения компьютера. Ответ на эту загадку заключается в том, что `USR` - это функция, а любая функция, по своему определению, должна получать что-то на входе и возвращать нечто на выходе. Поэтому остается лишь выяснить сущность этих «что-то» и «нечто» - и вопрос можно считать решенным.

Как вы увидите позже, большинство процедур в машинных кодах, если это необходимо, получают входные параметры и возвращают значения на регистрах. Как правило, это оказывается наиболее удобно. Поскольку функция `USR` предназначена для вызова машинных процедур, то и она может обмениваться числовыми данными с программой на Бейсике также через регистры, а именно - через регистровую пару BC. В качестве входного параметра используется аргумент функции, а на выходе значение пары BC передается бейсик-программе. А так как в приведенных выше примерах никакие регистры не изменялись, то и на экране появлялось то же самое число, которое использовалось в качестве аргумента.

Теперь модернизируем нашу программку так, чтобы содержимое регистровой пары BC изменялось. Можно, например, просто записать в нее какое-нибудь число. Обычно запись в регистры числовых значений называют загрузкой, поэтому в мнемоническом обозначении такие команды начинаются с LD (сокращение от известного вам по Бейсику слова LOAD - загрузить). А выражение «загрузить регистровую пару BC значением 1000» записывается как

```
        LD    BC,1000
```

Эта команда всегда состоит из трех байт: первый равен 1, а второй и третий соответствуют двухбайтовому представлению числа в памяти. Таким образом, программа из двух команд

```
        LD    BC,1000
        RET
```

в памяти будет представлена последовательностью кодов

```
1, 232, 3 и 201
```

Введите их последовательно, начиная, например, с адреса 60000 и выполните закодированную программку оператором

```
PRINT USR 60000
```

Если вы ничего не напутали, то на экране должно появиться число 1000.

Надо думать, на этих примерах вы уже почувствовали «прелесть» программирования в машинных кодах и догадались, что подобным методом может пользоваться только сумасшедший или неукротимый фанатик. Однако и фанатик в конце концов понимает, что лучше все же воспользоваться ассемблером, благо фирма HISOFT подарила синклеристам весьма недурную реализацию этого языка, по многим параметрам могущую считаться вполне профессиональной. (Лучшей реализацией языка ассемблера для компьютеров семейства ZX Spectrum считается транслятор фирмы OCEAN Software из пакета Laser Genius, однако ассемблер фирмы HISOFT остается непревзойденным по минимальному объему занимаемой самим транслятором памяти и, соответственно, максимальному размеру области, отводимой для создаваемого им кода программы. Как и Бейсик 48, этот ассемблер использует несколько усеченный строчный текстовый редактор. Это, конечно, немного хуже, чем экранный редактор (как, например, в Бейсике 128), но за все приходится чем-то расплачиваться - *Примеч. ред.*)

ГЛАВА ТРЕТЬЯ,

обучающая вводу и редактированию игровых программ в ассемблере GENS4

Существует несколько версий программы GENS, но для определенности мы будем рассматривать GENS4. Тем не менее, почти все, о чем пойдет речь в нашей книге, справедливо и для других версий ассемблера фирмы HISOFT, поэтому мы не всегда станем уточнять версию, а чаще будем писать просто GENS.

ЗАГРУЗКА GENS4

Для начала нужно сказать несколько слов о самой программе. Ее часто можно встретить в пакете с монитором-отладчиком MONS под общим названием **DEVPAC**. Этот пакет состоит из трех файлов:

- Загрузчик на Бейсике (изначально пакет DEVPAC не содержал загрузчика на Бейсике, а появился он там благодаря Н. Родионову [2] - *Примеч. ред.*);
- Ассемблер GENS4 (или GENS3);
- Монитор-отладчик MONS4 (MONS3).

На самом деле вовсе не обязательно загружать в память весь пакет. Во-первых, отладчик нам сейчас не нужен - он понадобится позже, когда мы начнем дизассемблировать (то есть получать исходные тексты) программы в машинных кодах, а во-вторых, GENS (так же, впрочем, как и MONS) может быть загружен по любому удобному адресу, отличному от заданного в загрузчике. Поэтому чаще достаточно загружать только коды ассемблера, введя с клавиатуры, например, такую строку:

```
LOAD "GENS4"CODE 25000
```

После этого (или до) опустите RAMTOP на один байт ниже адреса загрузки ассемблера, выполнив оператор

```
CLEAR 24999
```

и теперь, чтобы GENS можно было запустить командой RUN, введите строку

```
10 RANDOMIZE USR 25000
```

Если вас по какой-то причине не вполне устраивает предложенный адрес, можете заменить его на любой другой. Помните только, что он не должен быть слишком низким (чтобы не повредить бейсик-систему) или слишком высоким (чтобы поместился не только сам GENS, но и текст будущей программы, который расположится следом за кодами ассемблера). Допустимый диапазон адресов загрузки лежит примерно в пределах от 24500 до 54000.

СТРУКТУРА АССЕМБЛЕРНОЙ СТРОКИ

Как и в любом другом языке, программа на ассемблере начинается с набора исходного текста. Наверное, нет надобности говорить, что для этих целей редактор Бейсика, встроенный в компьютер, совершенно непригоден. Посему в первую очередь необходимо освоить специальный редактор ассемблерных текстов, в который вы попадаете сразу после запуска GENS. В этой главе мы научим вас вводить строки программы, просматривать полученный текст и, наконец, транслировать готовую программу.

Прежде всего нужно сказать о структуре строк исходного текста программы на ассемблере. Они состоят из нескольких полей и имеют следующий формат:

```
Номер Поле меток Поле мнемоник Поле операндов
12345 LABEL      LD            BC,1000
```

Каждое поле служит для определенных целей, поэтому мнемоника, например, не может быть записана в поле меток - это неминуемо приведет к ошибке.

Рассмотрим назначение каждого их этих полей.

Хотя строки в программах для GENS и нумеруются, но в ассемблере ссылки на номера строк, в отличие от Бейсика, нигде не применяются. Нумерация нужна исключительно для определения очередности следования строк, вывода листинга на экран или принтер и для вызова строк на редактирование. Допустимые значения этого поля - от 1 до 32767.

Но как же, спросите вы, в ассемблере отмечаются точки переходов, как обозначаются начала подпрограмм и данных, если отсутствуют ссылки на номера строк? Просто в ассемблере для этого используется другая методика, гораздо более удобная, как вы почувствуете позже. Здесь для любого рода ссылок используются специальные имена, задаваемые программистом, которые называются *метками*. Метки записываются в следующем после номера строки поле - в поле меток. Конечно, это поле не обязательно должно заполняться в каждой строке. Метки расставляются только там, где в них действительно возникает необходимость. Дальше мы более подробно поговорим о них и вообще о методах адресации, а пока перейдем к следующему полю строки.

Как мы уже говорили, в ассемблере команды микропроцессора представляются в виде сокращений английских слов и такие аббревиатуры называются мнемониками или мнемокодами. При наборе строк программы мнемоники команд должны располагаться обязательно в своем поле, иначе GENS не сможет их распознать, что приведет к появлению ошибки. В этом же поле записываются и специальные инструкции - директивы ассемблера, о которых мы также обязательно расскажем, но несколько позже. При необходимости это поле, так же, как и предыдущее, может быть пустым.

Если у команды имеются какие-либо операнды, они записываются в следующем и последнем поле строки - в поле операндов. Таким образом полная команда может занимать в строке программы одно или два поля.

ВВОД СТРОКИ

Давайте напишем небольшую программку, находясь в редакторе GENS. Для этого загрузите и запустите ассемблер, как было сказано выше, для определенности выбрав адрес загрузки равным 25000.

Вводить строки можно двумя способами: так же, как и в Бейсике, то есть каждый раз набирать номер и текст, или воспользоваться автоматической нумерацией, что на наш взгляд несравненно удобнее. Для автоматической нумерации строк нужно на подсказку редактора (символ > с курсором C или L) ввести команду **I** (Insert - режим вставки строк) и нажать **Enter**. Первая строка будет иметь номер 10 и последующие номера будут увеличиваться с шагом 10. Если вы пожелаете изменить порядок нумерации строк, введите ту же команду **I**, но с двумя параметрами, разделенными запятой: первое число - начальный номер, второе - шаг. Например, **I100,5**.

Итак, введите команду **I** с параметрами или без - после старта GENS оба параметра по умолчанию равны 10, поэтому в самом начале работы их можно и не указывать.

Так как в первой строке метки нам не понадобятся, сразу перейдем к полю мнемоник. Переход к следующему полю строки осуществляется вводом символа табуляции, который получается при одновременном нажатии клавиш **Caps Shift** и **8**. (Вместо символов табуляции можно вводить просто один или несколько пробелов - после окончательного ввода строки они автоматически будут заменены кодами табуляции, и при просмотре листинга текст будет выглядеть аккуратно выровненным по границам полей.) Нажмите эту комбинацию клавиш и увидите, как курсор перескочит вправо сразу на несколько позиций печати. Теперь можно набрать какую-нибудь инструкцию.

Как правило, программы на ассемблере начинаются с указания начального адреса, то есть того адреса, с которого полученная программа будет загружаться в память. Это достигается включением в программу директивы **ORG** (от англ. Origin - начало), следом за которой пишется десятичный или шестнадцатеричный адрес. (При записи шестнадцатеричных чисел они должны начинаться с символа #, например, #3FC.) Наберите в первой строке эту директиву и укажите после символа табуляции адрес 60000. Если вы случайно нажмете не ту клавишу, удалить последний введенный символ можно обычным образом, нажав клавиши **Caps Shift/0** или **Delete**. Но вот переместить курсор к любому символу строки стандартными методами не удастся: ведь клавиши **Caps Shift/8**, как вы уже знаете, отвечают за ввод табуляции, а **Caps Shift/5** удалит весь текст строки и переместит курсор в самое ее начало. Закончите ввод нажатием клавиши **Enter**, после чего строка должна принять вид

```
10      ORG    60000
```

а ниже появится номер следующей строки с курсором.

Аналогичным образом введите еще две строки:

```
20      LD     BC,1000
```

```
30      RET
```

Теперь нам нужно вернуться в строчный редактор GENS. Для этого достаточно нажать клавишу **Edit** или **Caps Shift/1**. На экране снова появится подсказка, на которую можно вводить команды редактора.

ПРОСМОТР ТЕКСТА

После ввода достаточно больших программ часто возникает необходимость просмотра полученного текста. Команда **L** позволяет вывести на экран листинг программы с любой строки по любую. Если команда введена без параметров, листинг будет выводиться с самой первой строки и до конца порциями по 15 строк. После вывода очередной порции будет ожидаться нажатие любой клавиши кроме **Edit**, которая прервет вывод и вернет управление редактору. Общий вид команды такой:

```
L[начальная строка[,конечная строка]]
```

Квадратные скобки указывают на необязательность параметров.

Если хотите, можете воспользоваться командой **L** и еще раз взглянуть на только что набранный текст.

ТРАНСЛЯЦИЯ ПРОГРАММЫ

Оттранслируйте теперь полученный исходный текст. Введите команду **A** безо всяких параметров. На экране должна появиться примерно такая надпись:

```
HiSoft GEN Assembler ZX Spectrum
ZX Microdrive Version 4.1b

Copyright (C) HiSoft 1987
V4.1 All right reserved

Pass 1 errors: 00

Pass 2 errors: 00

Table used: 13 from 104
```

Текст «шапки», естественно, может быть и другим, все зависит от версии, которой вы пользуетесь. Здесь нам важно, в общем-то, одно - то, что ни в первом, ни во втором проходе ассемблирования (Pass) ошибок (errors) не обнаружено. Надо сказать, что ассемблер обрабатывает исходный текст в два прохода: на первом проверяются синтаксические ошибки и выстраивается таблица меток (ее размер показывает самая нижняя строка сообщения Table used); во втором проходе генерируется машинный код программы.

ВЫХОД ИЗ GENS

Убедившись, что ошибок нет (по крайней мере, грубых), неплохо было бы проверить работоспособность получившейся программы. Работая с GENS вы в любой момент можете временно попрощаться с ассемблером и выйти в Бейсик, дав редактору команду **B** (Bye - «до встречи» или Basic, если вам так легче запомнить).

После выхода из GENS4 мы рекомендуем вам прежде всего очищать экран оператором `CLS` или простым нажатием клавиши **Enter**. Дело в том, что эта версия пользуется для вывода на экран встроенным драйвером, позволяющим выводить в строке до 51 символа мелким шрифтом, и при выходе стандартный канал вывода по каким-то причинам не сразу восстанавливается. Поэтому до очистки экрана оператор `PRINT` может не дать ожидаемых результатов.

Вероятно, вы обратили внимание на то, что предложенный пример в точности повторяет программку, которую мы недавно вводили с помощью оператора `POKE`. Поверьте, что сделали это мы не от скудости воображения, а для того, чтобы вы имели возможность удостовериться в идентичности полученных тогда и сейчас результатов. Введите с клавиатуры строку

```
FOR n=60000 TO 60003: PRINT PEEK n: NEXT n
```

и увидите столбик чисел, совпадающих с теми, которые мы вводили ранее вручную.

Снова вызовите редактор GENS и введите команду **L**. Как видите, текст набранной программки никуда не исчез.

ВНЕСЕНИЕ В ТЕКСТ ИЗМЕНЕНИЙ

Следующий по важности момент после набора строк - редактирование исходного текста. Ведь трудно себе представить, что какая-то программа может быть написана одним заходом, сразу начисто, без ошибок и опечаток. Для этого нужно научиться быстро изменять текст строк, не переписывая их целиком, вставлять недостающие команды, удалять лишние и, наконец, записывать программу на внешний носитель, чтобы плоды вашего труда не пропали даром.

Начнем со способов внесения правок в уже имеющийся текст. Для вызова строки на редактирование служит команда **E** (Edit) с номером нужной строки (рис. 3.1). Давайте в нашей программке подправим, например, строку 20, изменив число, загружаемое в регистровую пару BC. Введите команду E20. То, что вы увидели, возможно, несколько удивило вас. На экране появилась вызванная строка, но курсор на ней не задержался, а проскочил ниже, напечатав только номер. Весь остальной текст куда-то подевался. Может создаться такое впечатление, что теперь необходимо набирать весь текст заново. Однако, это только впечатление, и на самом деле строка сейчас готова к редактированию.

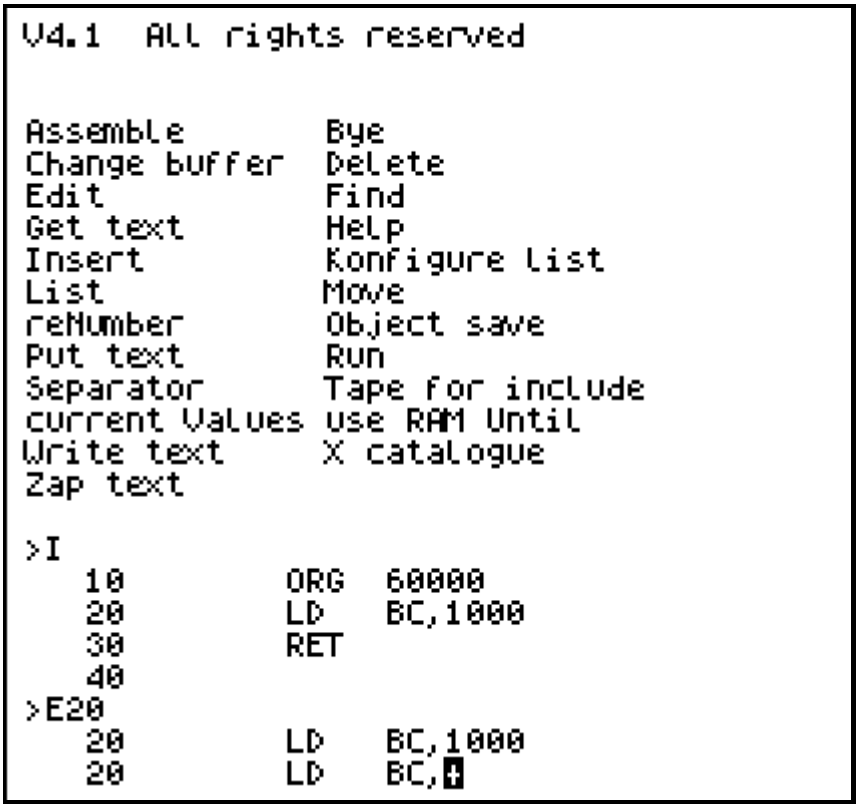


Рис. 3.1. Редактирование строки в GENS4

Вы можете вносить изменения в текст строки двумя способами: в *режиме вставки*, в котором вновь введенные символы «раздвигают» строку и не затирают имеющийся текст, и в *режиме замены*, в котором новые символы ложатся поверх прежних. В любом случае сначала необходимо подвести курсор к тому месту строки, где требуются какие-либо изменения. Вправо курсор можно перемещать по одной позиции, нажимая пробел, или разом перескакивать к следующему полю при нажатии **Caps Shift/8**. Перемещая курсор, вы увидите появляющийся под ним текст строки. Возврат на одну позицию влево происходит при нажатии клавиши **Delete** (**Caps Shift/0**)(обратите внимание на тот факт, что пока еще ничего не удаляется - вы просто перемещаете курсор, а верхняя строка нужна для подсказки).

Подведите курсор к числу 1000 так, чтобы он находился точно под единицей. Затем нажмите клавишу **C** для перехода в режим замены символов (в режим вставки можно перейти, нажав клавишу **I**, но это - на будущее). Обратите внимание, что курсор при этом изменяет свой вид. Теперь он выглядит как инвертированный символ + (в режиме вставки курсор имеет вид символа *, и это тоже - на будущее). Замените цифру 1 на 5, а следующий ноль, скажем, на 3. Затем выйдите из режима замены, нажав **Enter** и, наконец, введите строку в программу, еще раз нажав **Enter**. Редактирование строки закончено, и сейчас она должна иметь вид

```
20     LD    BC,5300
```

Можете ассемблировать новый текст и, выйдя в Бейсик, выполнить получившуюся программку оператором

```
PRINT USR 60000
```

Прежде чем показать вам, как можно сохранить исходный текст и готовый машинный код, приведем и остальные команды редактирования строки. Вот они:

L (List) - показать на экране текущий вид строки;
K (Kill) - удалить символ в позиции курсора;
Z (Zap) - удалить весь текст от курсора до конца строки;

X (eXpand) - переместить курсор в конец строки и войти в режим вставки;
Q (Quit) - отменить все сделанные правки и закончить редактирование строки;
R (Reload) - отменить все исправления и начать редактирование заново.

Все эти команды выполняются до включения режимов вставки или замены символов (команды **I** или **C**). Если же один из этих режимов уже включен, следует прежде выйти из него, нажав **Enter**.

СОХРАНЕНИЕ И ЗАГРУЗКА ТЕКСТОВ И ПРОГРАММ

Разобравшись с методами редактирования строк, пора узнать, как сохраняются результаты. В отличие от Бейсика, редактор GENS позволяет сохранить не только весь текст целиком, но и произвольную его часть, для чего в команде **P** (Put text) прежде указывается начальная строка сохраняемого фрагмента, а затем - конечная. Самым последним параметром команды служит имя, под которым вы хотите сохранить текст. (У строчного редактора есть одно полезное, но в некоторых случаях опасное свойство: часть команд, будучи введенными без параметров, выполняется с параметрами предыдущей команды. Поэтому, на первых порах будьте особо внимательны.) Поскольку в этой команде нет необязательных параметров, то для записи всего текста можно задать границы номеров сохраняемых строк «с запасом» и ввести, например, такую строчку:

```
P1,20000,ASMTEXT
```

Если вы работаете с дисководом и имеете в своем распоряжении дисковую версию ассемблера GENS4D или gens4b, то можете записать исходный текст на дискету. Формат команды в этом случае останется, в общем-то, тем же самым, только имя файла должно начинаться с указания номера дисковода, к которому вы собираетесь обратиться. Например, для записи на дисковод A нужно ввести

```
P1,20000,1:ASMTEXT
```

Заметьте, что двоеточие между номером дисковода и именем обязательно.

Обратное действие - загрузка ранее созданного исходного текста - осуществляется с помощью команды редактора **G** (Get text). Формат ее похож на формат предыдущей команды, правда, значение имеет только последний параметр - имя файла. Поэтому для загрузки программы GAME с магнитофона следует ввести

```
G,,GAME
```

а для загрузки одноименного файла с дисковода B эта же команда примет вид

```
G,,2:GAME
```

По сравнению с оператором Бейсика `LOAD` у команды **G** есть одно существенное отличие: она не уничтожает уже имеющийся в памяти текст, а добавляет новый в конец. После объединения строки программы будут перенумерованы с номера 1 и с единичным шагом. Часто такая особенность команды **G** бывает не только полезна, но и просто необходима, однако если вы собираетесь поработать над новой программой, прежде чем ее загружать, надо убедиться, что редактор свободен от какого бы то ни было текста, а если нет, то удалить его (как это сделать, мы скажем чуть позже).

Для сохранения полученного машинного кода (нередко называемого объектным, хотя это и не совсем верно) можно воспользоваться командой **O** (Object). Она имеет синтаксис, аналогичный команде **G**, то есть для записи оттранслированной программы в файл MYPROG при работе с лентой нужно ввести строку

```
O,,MYPROG
```
а при работе с дисководом A эта команда запишется как

```
O,,1:MYPROG
```
Записывая тексты и программы на диск, вы иногда можете увидеть на экране надпись

```
File exists Delete (Y/N)?
```
сообщающую о том, что одноименный файл уже имеется на диске. В этом случае вам предлагается решить, удалять этот файл или отказаться от записи с тем, чтобы заменить дискету и попытаться сохранить текст еще раз. Если на этот запрос ответить нажатием клавиши **Y**, то файл на диске будет переписан, нажатие любой другой клавиши приведет к отказу от выполнения команды.

УДАЛЕНИЕ ТЕКСТА

Для удаления ненужных строк исходного текста в редакторе GENS имеется несколько возможностей. Первая из них в точности повторяет способ избавления от лишних строк в Бейсике: достаточно набрать номер удаляемой строки и нажать **Enter**.

Но, как вы понимаете, такой способ приемлем для удаления двух-трех строчек, когда же требуется ликвидировать значительную часть текста, подобная методика превращается в настоящую пытку. Фирма HISOFT учла интересы программистов и включила в редактор возможность удаления произвольной группы строк. Для этих целей служит команда **D** (Delete - удалить) с двумя параметрами, разделенными запятой. Первым указывается номер начальной строки удаляемого фрагмента, а затем - номер последней удаляемой строки. Например, для исключения из текста строк, начиная с сотой и по стопятидесятую включительно, нужно ввести команду

```
D100,150
```
Обращаться с этой командой нужно достаточно осторожно, так как выполняется она сразу же после ввода, безо всяких дополнительных подтверждающих запросов, а восстановить удаленный текст можно будет только если вы успели сохранить его на ленте или дискете. Ни в коем случае не следует использовать команду **D** без параметров, потому как при этом будут взяты параметры предыдущей введенной команды (параметры по умолчанию), а они могут далеко не соответствовать желаемым.

В редакторе GENS есть еще одна полезная команда, предназначенная для удаления текста. Это команда **Z**. С ее помощью можно быстро уничтожить сразу весь текст. Она не требует никаких параметров, а после ввода просит подтвердить ваши намерения запросом

```
Delete text (Y/N)?
```
Если вы не передумали стирать текст, нажмите клавишу **Y**, в противном случае - любую другую клавишу.

ГЛАВА ЧЕТВЕРТАЯ,

показывающая, как сделать надпись на экране и создать простейшие изображения

Получив из предыдущей главы некоторое представление о структуре ассемблерной строки, редакторе GENS и обо всем прочем, что совершенно необходимо для работы с ассемблером, можно, наконец, приступать к программированию. С чего же начать? Наверное, мы не сильно ошибемся, если предположим, что первейшее желание любого, начинающего изучать новый язык - это получить что-то на экране. Пусть это будет всего лишь какая-нибудь надпись, или даже просто одна-единственная буква.

Именно с таких простых действий мы и начнем наши опыты в программировании, постепенно усложняя задания и изучая все новые и новые команды ассемблера. На первых порах вы должны научиться выводить в определенное место экрана символы и числа, причем с заранее заданными атрибутами, уметь ставить точки, проводить линии, дуги и окружности. Освоив «джентльменский» набор команд и приемов программирования, можно попытаться создать на экране нечто полезное, например, текст, заключенный в рамку или один из кадров заставки с надписями как русскими, так и латинскими буквами.

ВЫВОД БУКВЕННЫХ И ЦИФРОВЫХ СИМВОЛОВ

Из предыдущего описания вы, возможно, сделали вывод, что сейчас нам предстоит долгая и кропотливая работа над созданием подпрограммы вывода символов на экран, затем придется писать еще одну процедуру, устанавливающую цвета печатаемых букв и цифр, а напоследок, собрав остатки угасающих сил, придумывать способ, как выводить сразу целые строки...
И после всего этого решить, что гораздо проще и эффективнее воспользоваться оператором Бейсика `PRINT` и забыть об ассемблере как о кошмарном сне.

Так вот, смеем вас заверить, что ничего подобного вам не грозит. Очень скоро вы убедитесь, что большинство операций, доступных Бейсику, в ассемблере выполняется почти так же просто. Ведь, как мы уже говорили, в ПЗУ компьютера имеются необходимые подпрограммы для выполнения всех бейсиковских операторов. Поэтому во многих случаях достаточно знать лишь две вещи: первое - по какому адресу расположена та или иная подпрограмма, и второе - как этой подпрограмме передать требуемые параметры. Ну и, конечно же, нужно представлять, каким образом вообще вызываются подпрограммы в ассемблере. А выполняет это действие команда микропроцессора CALL (звать, вызывать), которую можно сравнить с известным вам оператором `GO SUB`. Только вместо номера строки после команды указывается адрес перехода (еще раз напомним, что адреса обозначаются числами в диапазоне от 0 до 65535).

Сначала разберемся, что требуется для печати символов.

Общаясь с Бейсиком, вы могли заметить, что оператор `PRINT` весьма универсален и используется для многих целей. С его помощью можно выводить символы и строки не только на основной экран, но и в служебное окно, если написать `PRINT #0` или `PRINT #1`. В системе TR-DOS этот же оператор применяется для записи в файлы прямого и последовательного доступа, а для вывода на принтер имеется другая его разновидность - оператор `LPRINT`.

Для многих, вероятно, не будет новостью, что `LPRINT`, в сущности, это уже некоторое излишество Бейсика, так как часто удобнее бывает заменять его на `PRINT #i`, где i=3, который выводит информацию в поток `#3` (подробно о каналах и потоках можно прочитать

в [2]), то есть на принтер. Если же номер потока в операторе PRINT не конкретизирован, то по умолчанию вывод осуществляется в поток #2 - на основной экран.

В то время как в Бейсике нужный поток устанавливается автоматически, в ассемблере программист сам должен позаботиться о своевременном и правильном включении текущего потока. Для этой цели в ПЗУ имеется специальная подпрограмма, расположенная по адресу 5633 (или в шестнадцатеричном формате - #1601). Перед ее вызовом в аккумулятор следует поместить номер требуемого потока. Вы, наверное, еще не забыли, что для занесения в какой-либо регистр или регистровую пару некоторого значения используется команда LD. Таким образом, назначить поток #2 для вывода на основной экран можно всего двумя командами микропроцессора:

```
        LD    A,2
        CALL  5633
```

После этого можно что-нибудь написать на экране.

Подпрограмма, соответствующая оператору PRINT (или LPRINT) располагается по адресу 16, а перед ее вызовом в регистре A следует указать код выводимого символа. То есть, чтобы напечатать, например, букву **A**, загрузим в аккумулятор код 65 и вызовем подпрограмму с адресом 16:

```
        LD    A,65
        CALL  16
```

Теперь остается дописать команду RET и программка, печатающая на экране букву **A**, будет, в принципе, готова. Но, прежде чем привести законченный текст, хотелось бы сказать еще вот о чем. Во-первых, для определения кода нужного символа не обязательно каждый раз заглядывать в таблицу, можно предложить ассемблеру самостоятельно вычислять коды. Достаточно нужный символ заключить в кавычки. В нашем случае это будет выглядеть так:

```
        LD    A,"A"
```

И еще один момент.

Если вы дизассемблируете даже целую сотню фирменных игрушек, то вряд ли где-то обнаружите инструкцию CALL 16, хотя добрая половина из них не отказывает себе в удовольствии попользоваться возможностями ПЗУ. Объясняется это тем, что в системе команд микропроцессора Z80 для вызова подпрограмм помимо CALL имеется еще одна инструкция, более ограниченная в применении, но зато и более эффективная. Это команда RST. Она отличается от CALL, в сущности, только одним: с ее помощью можно обратиться лишь к нескольким первым, причем строго фиксированным, адресам. В частности, к адресу 16. А основное преимущество этой команды состоит в том, что она очень компактна и занимает в памяти всего один байт вместо трех, требуемых для размещения кодов команды CALL. Поэтому вместо

```
        CALL  16
```

значительно выгоднее писать

```
        RST   16
```

Подводя итог сказанному, можно написать программку, которая будет работать подобно оператору PRINT #2;"A". Загрузите GENS и в редакторе наберите такой текст:

```
10      ORG   60000
20      LD    A,2
30      CALL  5633
```

```
40      LD    A,"A"
50      RST   16
60      RET
```

Чтобы проверить работу этой программки, оттранслируйте ее, введя в редакторе команду **A**, а затем выйдите в Бейсик и, предварительно очистив экран, запустите ее с адреса 60000 оператором `RANDOMIZE USR 60000`.

После этого можете поэкспериментировать, подставляя в строке 40 другие значения для регистра A. Посмотрите, что получится, если указать коды псевдографических символов, `UDG` или ключевых слов Бейсика, которые имеют значения от 128 до 255. Правда, в этом случае придется отказаться от символьного представления кодов и нужно будет вводить непосредственные числовые величины. Тем не менее, вы убедитесь, что команда RST 16 превосходно справляется с поставленной задачей и работает точно так же, как работал бы в этом случае и оператор `PRINT`.

Эксперименты экспериментами, но здесь мы должны сделать небольшое предупреждение. В программировании на ассемблере имеется множество «подводных камней», поэтому не очень удивляйтесь, если после очередного опыта ваша программа «улетит» в неизвестном направлении. Дабы избежать стрессов, вызванных подобной неприятностью, **всегда перед стартом программы сохраняйте измененный исходный текст**, а лучше - не заходите в своих изысканиях чересчур далеко, пока не изучите книгу до конца. В свое время мы, по возможности, расскажем обо всех (ну, по крайней мере, о многих) «крутых поворотах», подстерегающих программистов-ассемблерщиков на пути создания полноценных программ.

Сделав первый, самый трудный шаг, двинемся дальше и несколько усложним нашу программку. Попробуем напечатать символ в определенном месте экрана, например, в 10-й строке и 8-м столбце, то есть попытаемся воспроизвести оператор `PRINT AT 10,8;"X"`. Оказывается, и это в ассемблере сделать не многим труднее, чем в Бейсике.

Помимо обычных «печатных» символов (так называемых, ASCII-кодов), псевдографики, `UDG` и токенов (ключевых слов) Бейсика существует ряд специальных кодов, которые не выводятся, а служат для управления печатью. Часто их так и называют - управляющие символы. Они имеют коды от 0 до 31, хотя при выводе на экран используются не все, а только некоторые из них.

Директиве `AT` соответствует управляющий символ с кодом 22. И кроме этого кода необходимо вывести еще два, указывающих номера строки и столбца на экране. То есть, команду RST 16 нужно выполнить трижды:

```
        LD    A,22
        RST   16
        LD    A,10
        RST   16
        LD    A,8
        RST   16
```

После этого можно вывести и сам символ:

```
        LD    A,"X"
        RST   16
```

Управляющие коды имеются и для всех прочих директив оператора `PRINT`: `TAB`, `INK`, `PAPER`, `FLASH`, `BRIGHT`, `OVER`, `INVERSE`, а также для запятой и апострофа. В табл. 4.1 приведены значения всех управляющих кодов, а также указано, какие байты требуется передать в качестве параметров. Как видите, для кодов 6, 8 и 13 дополнительных данных не требуется, коды 16...21 нуждаются еще в одном байте, а 22 и 23 ожидают ввода двух значений. Обратите

внимание, что код 23 (TAB), вопреки ожиданиям, требует не одного, а двух байт, хотя на самом деле роль играет только первый из них, а второй игнорируется и может быть каким угодно (на это в таблице указывает вопросительный знак).

Таблица 4.1. Коды управления печатью

Код	Байты параметров	Значение
6	-	Запятая
8	-	Забой
13	-	Перевод строки (апостроф)
16	colour	Цвет INK
17	colour	Цвет PAPER
18	flag	FLASH
19	flag	BRIGHT
20	flag	INVERSE
21	flag	OVER
22	Y, X	Позиция AT
23	X, ?	Позиция TAB

Допустимые значения для параметров следующие:

```
colour - 0...9
flag   - 0 или 1
X      - 0...31
Y      - 0...21
```

Теперь напишем на ассемблере пример, соответствующий оператору

```
PRINT AT 20,3; INK 1; PAPER 5, BRIGHT 1; "OK."
```

Для большей наглядности снабдим нашу программку комментариями. В ассемблере комментарии записываются после символа «точка с запятой» (**;**), который может находиться в любом месте программы. Весь текст от этого символа до конца строки при трансляции пропускается и на окончательном машинном коде никак не сказывается. Само собой, при наборе примеров вы можете пропускать все или часть комментариев, тем более, что в книге многие из них даны на русском языке, а GENS, к сожалению, с кириллицей не знаком.

```
 10    ORG   60000
 20    LD    A,2         ; вывод на основной экран (PRINT #2).
 30    CALL  5633
 40 ;----------------
 50    LD    A,22        ; AT 20,3
 60    RST   16
 70    LD    A,20
 80    RST   16
 90    LD    A,3
100    RST   16
110 ;----------------
120    LD    A,16        ; INK 1
130    RST   16
140    LD    A,1
150    RST   16
160 ;----------------
170    LD    A,17        ; PAPER 5
180    RST   16
190    LD    A,5
200    RST   16
```

```
210 ;----------------
220    LD    A,19        ; BRIGHT 1
230    RST   16
240    LD    A,1
250    RST   16
260 ;----------------
270    LD    A,"O"       ; печать трех символов строки OK.
280    RST   16
290    LD    A,"K"
300    RST   16
310    LD    A,"."
320    RST   16
330    RET
```

Не правда ли, получилось длинновато? Даже не верится, что этот пример после трансляции будет занимать в памяти меньше полусотни байт. А на самом деле его можно сократить еще в несколько раз. Для этого нужно воспользоваться подпрограммой ПЗУ, позволяющей выводить строки символов, да научиться формировать такие строки в программе.

Ассемблер предоставляет несколько директив для определения в программе текстовых строк и блоков данных. Вот они:

```
DEFB - через запятую перечисляется последовательность
       однобайтовых значений;
DEFW - через запятую перечисляется последовательность
       двухбайтовых значений;
DEFM - в кавычках задается строка символов;
DEFS - резервируется (и заполняется нулями) область памяти
       длиной в указанное число байт.
```

Эти директивы чем-то напоминают оператор Бейсика DATA, но в отличие от него не могут располагаться в произвольном месте программы. Мы уже говорили, что ассемблер, как никакой другой язык, «доверяет» программисту. Это, в частности, объясняется тем, что микропроцессор не способен сам отличить, к примеру, код буквы **A** от кода команды LD B,C - и то и другое обозначается десятичным числом 65. Поэтому недопустимо размещать блоки данных, скажем, внутри какой-либо процедуры, так как в этом случае они будут восприниматься микропроцессором как коды команд, и чтобы избежать конфликтов, все данные лучше размещать в самом конце программы или уж, по крайней мере, между процедурами, после команды RET.

Для преобразования вышеприведенного примера выпишем последовательность кодов, выводимых командой RST 16, следом за директивой DEFB:

```
       DEFB  22,10,8,16,1,17,5,19,1,"O","K","."
```

Подпрограмма вывода последовательности кодов располагается по адресу 8252 и требует передачи двух параметров: адрес блока данных перед обращением к ней нужно поместить в регистровую пару DE, а длину строки - в BC. И если вычисление второго параметра не должно вызвать трудностей, то об определении адреса стоит поговорить.

В предыдущей главе, в разделе «Структура ассемблерной строки», мы упоминали о существовании такого понятия как метка, но еще ни разу им не воспользовались - не было особой надобности. Но теперь нам без них просто не обойтись. Как уже говорилось, метки ставятся в самом начале строки, в поле, которое мы до сих пор пропускали, и служат для определения адреса первого байта команды или блока данных, записанных следом. Имена меток в GENS должны состоять не более чем из шести символов (если метка состоит более чем из 6 символов, лишние при трансляции автоматически отбрасываются, поэтому более

длинные имена возможны, но исключительно ради наглядности), среди которых могут быть такие:

```
0...9 A...Z a...z _ [ ] \ # $  и J
```

но помните, что они не могут начинаться с цифры или знака #. Кроме того, метки не должны совпадать по написанию с зарезервированными словами, то есть с именами регистров и мнемониками условий. Например, недопустимо использование метки **HL**, однако, благодаря тому, что GENS делает различие между строчными и прописными буквами, имя **hl** вполне может быть меткой. Ниже перечислены все зарезервированные слова в алфавитном порядке:

Зарезервированные слова GENS

```
$    A    AF   AF'  B    BC   C    D    E
E    H    HL   I    IX   IY   L    M    NC
NZ   P    PE   PO   R    SP   Z
```

И еще один очень важный момент, связанный с метками. Их имена должны быть уникальными, то есть одно и то же имя не может появляться в поле меток дважды. Хотя, конечно, ссылок на метку может быть сколько угодно.

Можно наконец переписать наш пример с использованием блока данных, присвоив последнему имя TEXT:

```
        ORG   60000
        LD    A,2
        CALL  5633
        LD    DE,TEXT     ;в регистровую пару DE записывается
                          ; метка TEXT, соответствующая адресу
                          ; начала блока данных.
        LD    BC,12       ;в регистровую пару BC заносится число,
                          ; соответствующее количеству кодов
                          ; в блоке данных.
        CALL  8252        ;обращение к подпрограмме ПЗУ,
                          ; которая печатает строку на экране.
        RET
TEXT    DEFB  22,10,8,16,1,17,5,19,1,"O","K","."
```

Не удивляйтесь, что в этом примере отсутствуют номера строк. В дальнейшем мы везде будем приводить тексты программ именно в таком виде, во-первых, потому, что нумерация не несет никакой смысловой нагрузки, а во-вторых, многие фрагменты в ваших программах, скорее всего, будут пронумерованы совершенно иначе.

В заключение этого раздела расскажем еще об одной полезной подпрограмме ПЗУ, связанной с печатью символов. Вы знаете, что в Бейсике при использовании временных атрибутов в операторе `PRINT` их действие заканчивается после выполнения печати, и следующий `PRINT` будет выводить символы с постоянными атрибутами. В ассемблере же команда RST 16 временные установки не сбрасывает и для восстановления печати постоянными атрибутами нужно вызвать подпрограмму, расположенную по адресу 3405. Продемонстрируем это на таком примере:

```
        ORG   60000
        LD    A,2
        CALL  5633
        LD    DE,TEXT1    ;печать текста, обозначенного меткой
        LD    BC,16       ; TEXT1, длиной в 16 байт.
        CALL  8252
        CALL  3405        ;восстановление постоянных атрибутов.
```

```
        LD    DE,TEXT2    ;печать текста, обозначенного меткой
        LD    BC,11       ; TEXT2, длиной в 11 байт.
        CALL  8252
        RET
TEXT1   DEFB  22,3,12,16,7,17,2
        DEFM  "TEMPORARY"
TEXT2   DEFB  22,5,12
        DEFM  "CONSTANT"
```

После трансляции и выполнения этой программки вы увидите на экране две надписи: верхняя (TEMPORARY) выполнена с временными атрибутами (белые буквы на красном фоне), а нижняя (CONSTANT) - постоянными.

ПОДГОТОВКА ЭКРАНА К РАБОТЕ ИЗ АССЕМБЛЕРА

Разобравшись с выводом символов, неплохо было бы научиться очищать экран, устанавливать постоянные атрибуты и цвет бордюра также из программы в машинных кодах, а не производить эту предварительную подготовку в Бейсике. Тем более, что выполняются все эти операции достаточно просто.

Прежде всего необходимо задать постоянные атрибуты. Сделать это можно по-разному, но проще всего рассчитать байт атрибутов и поместить его в системную переменную `ATTR_P` по адресу 23693. Напомним, что в байте атрибутов биты 0..2 определяют цвет «чернил» `INK`, биты 3..5 отвечают за цвет «бумаги» `PAPER`, а 6-й и 7-й биты устанавливают или сбрасывают соответственно атрибуты яркости `BRIGHT` и мерцания `FLASH`. Поэтому требуемое значение цвета можно подсчитать по формуле

```
INK+PAPER×8+BRIGHT×64+FLASH×128
```

Так для

```
INK 6: PAPER 0: BRIGHT 1: FLASH 0
```

искомый байт будет равен

```
6+0×8+1×64+0×128=70
```

А если вам лень считать, можете поступить проще: очистите экран и введите с клавиатуры последовательно две строки

```
PRINT INK 6; PAPER 0; BRIGHT 1; FLASH 0; " "
PRINT ATTR (0,0)
```

В верхнем левом углу экрана появится черный квадратик, а под ним - искомое число 70.

Теперь остается полученное число поместить в ячейку с адресом 23693, то есть выполнить инструкцию, аналогичную оператору Бейсика `POKE 23693,70`. Но вот беда - микропроцессор Z80 не располагает командами пересылки в память или из памяти непосредственных значений. Поэтому такую простую операцию приходится выполнять в два захода: сначала число нужно поместить в аккумулятор (и только в аккумулятор - никакой другой регистр для этого не подходит!), а затем значение из него переписать в ячейку. Команда записи в память очень напоминает загрузку регистров, только адрес или метка в этом случае заключается в круглые скобки. То есть предложение «загрузить ячейку с адресом 23693 значением из аккумулятора» записывается как LD (23693),A. Обратите внимание, что данный тип команд может выполняться только с регистром A!

Раз уж мы заговорили о способах пересылки значений между регистрами и памятью, приведем и другие инструкции, относящиеся к этой группе. Действие, обратное LD (Address),A и аналогичное функции Бейсика `PEEK Address`, выполняется командой LD A,(Address). Все прочие регистры могут обмениваться числовыми значениями с памятью только в парах. Выглядят такие команды следующим образом:

```
        LD    (Address),rp
        LD    rp,(Address)
```

где rp - одна из регистровых пар BC, DE или HL. (Забегая вперед, добавим, что в указанных командах могут участвовать также регистры IX, IY и SP.) Первая из них загружает две смежные ячейки памяти значением из регистровой пары, а вторая, наоборот, пересылает из памяти двухбайтовое число в обозначенные регистры. Заметим, что всегда предпочтительнее в данных командах применять пару HL, так как с ее участием эти инструкции занимают на байт меньше памяти и выполняются быстрее.

Таким образом, для установки постоянных атрибутов можно написать две строки вроде:

```
        LD    A,70          ;байт атрибутов
        LD    (23693),A     ;помещаем в системную переменную ATTR_P
```

Хотя таким способом можно пользоваться в большинстве случаев, он оказывается не всегда удобен. Например, если нужно установить какой-то один из атрибутов, то придется изменять не весь байт, а только некоторые его биты. А если требуется указать режимы `OVER` или `INVERSE`, либо для `INK` и `PAPER` задать значения 8 или 9, то описанный метод и вовсе непригоден.

В этих случаях можно поступить так. Первым делом необходимо установить текущий поток, связанный с основным экраном так же, как мы это делали раньше. Затем вызвать уже известную вам подпрограмму 3405 для «сброса» временных атрибутов. Следующим этапом с помощью команды RST 16 или процедуры 8252 установить новые временные атрибуты. И, наконец, временные атрибуты перевести в постоянные, для чего лучше всего вызвать соответствующую подпрограмму ПЗУ, находящуюся по адресу 7341.

Для иллюстрации этого способа напишем фрагмент, устанавливающий режимы `OVER 1` и `PAPER 8`:

```
        LD    A,2
        CALL  5633          ;определяем вывод на основной экран
        CALL  3405          ;«сбрасываем» временные атрибуты
        LD    DE,ATTR1
        LD    BC,4
        CALL  8252          ;выводим управляющие коды для новых
                            ; временных атрибутов
        CALL  7341          ;переводим временные атрибуты
                            ; в постоянные
        RET
ATTR1   DEFB  21,1,17,8     ;последовательность управляющих кодов
                            ; для OVER 1 и PAPER 8
```

Стоит ли говорить, что подобное действие может выполняться в реальной игровой программе неоднократно и при этом наверняка потребуется каждый раз изменять различные атрибуты. Посему было бы очень полезно иметь универсальную процедуру, которая работала бы по-разному в зависимости от входных параметров.

Сложность здесь заключается лишь в том, как до подпрограммы 8252 «донести» содержимое регистровых пар BC и DE - ведь перед ней должны выполниться две процедуры (CALL 5633 и CALL 3405), которые обязательно изменят значения нужных регистров. Значит, до поры до времени их нужно как-то сохранить.

Решение может показаться простым и очевидным: нужно запомнить значения регистров где-то в памяти и тем самым освободить их для каких-либо нужд, а затем восстановить их первоначальный вид, прочитав из памяти записанные ранее числа. Да, действительно, иногда так и делают. Так же поступает и большинство компиляторов, но как вы знаете, они не отличаются сообразительностью и используют ресурсы компьютера не самым оптимальным образом. Ведь известно, что команды пересылок между регистрами и памятью выполняются заметно дольше, чем обмен данными непосредственно между регистрами. Кроме того, дополнительные временные переменные лишь попусту транжирят память. И ведь еще необходимо помнить, где что лежит!

Применение такой методики чревато и еще одной неприятностью. Когда вы начнете писать на ассемблере достаточно большие программы (а мы надеемся, что это время не за горами), то очень скоро обнаружите, что шести символов для меток маловато. В результате этого очень легко можно ошибиться и поставить метку с уже существующим именем. Поэтому старайтесь везде, где только можно, обходиться без лишних меток.

Всего перечисленного можно избежать, если пойти другим путем, используя гораздо более удобное и эффективное средство - машинный стек. Во второй главе мы уже объясняли, что это такое, но не рассказывали, как с ним работать. Вообще-то, к помощи стека мы уже прибегали много раз, даже не подозревая об этом. Дело в том, что все команды вызова подпрограмм, будь то CALL или RST, прежде всего заносят в стек адрес возврата, то есть адрес следующей за вызовом команды. После выполнения подпрограммы завершающая команда RET снимает со стека этот адрес и тем самым возвращает управление основной программе.

Кроме такого косвенного взаимодействия со стеком имеется возможность непосредственного обмена с ним числовыми значениями из регистровых пар. Для этих целей служат две команды: PUSH (втолкнуть), которая помещает в стек значение из регистровой пары и POP (вытолкнуть, выскочить), забирающая с вершины стека двухбайтовое число в регистровую пару. Например, предложение «Запомнить в стеке значение регистровой пары BC» запишется так:

```
        PUSH   BC
```

а команда «Взять в регистровую пару HL значение с вершины стека» будет выглядеть следующим образом:

```
        POP    HL
```

Как мы уже говорили, существует определенный порядок работы со стеком, и числа, занесенные на его вершину последними, должны быть сняты в первую очередь. При этом нужно очень внимательно следить не только за очередностью обмена со стеком, но и за тем, чтобы его состояние при выходе из подпрограммы было таким же, как и при входе. Иными словами, необходимо, чтобы количество команд PUSH и POP в каждой подпрограмме было одинаковым (хотя, заметим, что вовсе не обязательно забирать числа со стека в те же регистровые пары, из которых производилась запись). Несоблюдение этих правил может привести к совершенно непредсказуемым результатам. Кстати, стековые ошибки относятся к наиболее распространенным, поэтому, если при отладке программы вы обнаружите, что в какой-то момент компьютер «зависает», «сбрасывается» или ведет себя как-то странно, то первым делом следует проверить те строки, где встречаются команды обмена со стеком.

Все только что сказанное верно, однако, изучая фирменные игрушки, можно заметить, что опытные программисты порой обращаются со стеком весьма своевольно, совершенно не придерживаясь общепринятых правил. В будущем мы поговорим о некоторых оригинальных приемах программирования, позволяющих иногда заметно сократить машинный код, а главное, увеличить быстродействие программы. Но пока вы не прочувствуете хорошенько идею стека, слишком увлекаться экспериментами в этой области мы вам все же не советуем.

Теперь, зная кое-что о машинном стеке, можно переписать предыдущий пример, оформив его в виде самостоятельной процедуры:

```
ATTRIB PUSH  BC          ;сохраняем в стеке значения регистровых
       PUSH  DE          ; пар BC и DE
       LD    A,2
       CALL  5633
       CALL  3405
; Снимаем с вершины стека сохраненные ранее значения в обратном порядке:
       POP   DE          ;сначала в DE,
       POP   BC          ; а затем в BC
       CALL  8252
       CALL  7341
       RET
```

Для вызова этой процедуры необходимо задать строку DEFB с перечислением управляющих кодов, занести адрес этой строки в DE и в регистровой паре BC указать ее длину:

```
       LD    DE,ATTR2
       LD    BC,6
       CALL  ATTRIB
       RET
ATTR2  DEFB  16,5,17,1,19,1
```

Если установка постоянных атрибутов иногда может оказаться довольно сложным процессом, то остальные настройки экрана не должны вызвать никаких трудностей. Окрасить бордюр в любой нужный цвет можно двумя способами. Первый из них почти в точности повторяет оператор Бейсика

```
OUT 254,color
```

При записи в порт, так же, как и при записи в память, в ассемблере нельзя использовать непосредственные значения, поэтому код цвета прежде необходимо поместить в регистр A. Адрес порта обязательно нужно заключать в круглые скобки. Например, для установки красного бордюра можно воспользоваться такими командами:

```
       LD    A,2
       OUT   (254),A
```

Как вы знаете из Бейсика, установленный подобным образом цвет бордюра обычно надолго не задерживается. Для более долговечного его изменения используется оператор `BORDER`, а выполняет эту процедуру подпрограмма ПЗУ по адресу 8859. Перед обращением к ней в аккумуляторе должен содержаться код цвета. Скажем, для установки голубого бордюра следует написать:

```
       LD    A,5
       CALL  8859
```

Для полноты информации напомним также, что байт атрибутов для бордюра обычно сохраняется в области системных переменных по адресу 23624.

Что же касается очистки экрана, то это самая простая операция. Соответствующая подпрограмма находится по адресу 3435 и не требует никаких входных параметров. Единственное, что нужно помнить - после выполнения команды CALL 3435 текущим

устанавливается поток, связанный с выводом в служебное окно экрана. Таким образом, после вызова этой процедуры необходимо вновь назначить требуемый поток. Например:

```
        CALL  3435
        LD    A,2
        CALL  5633
```

В завершение этого раздела предлагаем вам законченную процедуру, окрашивающую экран и бордюр в черный цвет, устанавливающую голубой цвет для выводимых символов и тем самым подготавливающую основной экран к приему текстовой и графической информации:

```
SETSCR LD    A,5
       LD    (23693),A
       LD    A,0
       CALL  8859
       CALL  3435
       LD    A,2
       CALL  5633
       RET
```

Мы присвоили этой процедуре собственное имя (метку) SETSCR не случайно.
В последующих разделах и главах эта подпрограмма будет использоваться неоднократно, и чтобы не переписывать ее каждый раз заново, мы будем просто ссылаться на нее, вставляя в текст команду CALL SETSCR. Вам же мы посоветуем записать ее (вместе с процедурой ATTRIB) на ленту или дискету в виде отдельного файла, тогда в будущем вам достаточно будет только подгрузить ее к основному тексту с помощью команды редактора **G**. Точно так же рекомендуем вам поступить и с другими подпрограммами, которые встретятся в книге, и в конце концов в вашем распоряжении появится библиотека наиболее важных и часто используемых в игровых программах процедур.

ПЕЧАТЬ ЦЕЛЫХ ЧИСЕЛ

Если печать символов не вызывает особых затруднений, то вот с выводом чисел дела обстоят несколько сложнее. Объясняется это тем, что любое число прежде всего необходимо представить в виде последовательности цифровых символов, а затем уже выводить их на экран (или на принтер). Задача осложняется еще тем, что числовые значения могут храниться в памяти в различных форматах, занимая один, два или больше байт. Так, например, Бейсик пользуется пятибайтовым представлением чисел, а обрабатывает их большая и сложная программа ПЗУ, называемая *калькулятором*.

Используя калькулятор, можно выполнять множество математических расчетов с любыми целыми и дробными числами, доступными Бейсику, но пока мы не будем объяснять, как это делается, поскольку тема эта носит самостоятельный характер, а также требует определенной предварительной подготовки. (Работе с калькулятором будет посвящен отдельный раздел девятой главы.) Сейчас же мы сосредоточимся на выводе только целых чисел из диапазона 0...65535, для чего используем особую рабочую область памяти, именуемую стеком калькулятора. Мы уже говорили во второй главе, что эта область не имеет определенного строго фиксированного адреса, да в большинстве случаев нам и не обязательно знать это, поскольку операционная система сама следит за ее местоположением и размером. Но при желании вы можете выяснить и то и другое. Адрес дна стека калькулятора можно прочитать из системной переменной STKBOT (23651/23652), а адрес его вершины определяется другой переменной - STKEND (23653/23654).

Для печати чисел можно вызвать подпрограмму, находящуюся по адресу 11747. Она напечатает число, расположенное в данный момент на вершине стека калькулятора, значит, требуемое число прежде нужно туда поместить. В этом вам поможет другая подпрограмма - 11563, размещающая на стеке калькулятора значение из регистровой пары BC. Следовательно, программа, которая выводит на экран, например, число 12345, может иметь следующий вид:

```
        ORG   60000
        ENT   $
        CALL  SETSCR       ;установка экрана
        LD    BC,12345     ;требуемое число 12345 загружаем
                           ; в регистровую пару BC
        CALL  11563        ;подпрограмма ПЗУ заносит в стек
                           ; калькулятора число из BC
        CALL  11747        ;подпрограмма ПЗУ печатает число,
                           ; находящееся на вершине
                           ; стека калькулятора
        RET
SETSCR .........           ;здесь должна располагаться подпрограмма,
                           ; описанная в предыдущем разделе
```

В примере мы использовали еще одну полезную директиву ассемблера - ENT. Она дает возможность запускать оттранслированную программу непосредственно из редактора GENS, без выхода в Бейсик. Это бывает особенно удобно при отладке небольших фрагментов, в которых требуется подобрать тот или иной параметр методом «научного тыка», не прибегая к расчетам.

Знак доллара (**$**) при трансляции заменяется адресом начала строки, в которой он встретился, и в нашем случае он примет значение 60000. После трансляции программы, кроме обычной информации, вы увидите на экране строку

```
Execute 60000
```

говорящую о том, что программа может быть выполнена с адреса 60000. Теперь для запуска полученного машинного кода достаточно ввести команду редактора **R**, завершив ввод, как всегда, нажатием клавиши **Enter**.

Описанный способ позволяет выводить любые числа от 0 до 65535. Для печати же больших значений требуется совершенно иной подход, и здесь нет столь простых решений, поэтому мы оставим этот вопрос «на потом». А сейчас представим еще одну подпрограмму из ПЗУ компьютера, предназначенную для вывода целых чисел.

В тех случаях, когда можно обойтись гораздо меньшими числовыми величинами, не превышающими 9999, лучше пользоваться подпрограммой, находящейся по адресу 6683. Она печатает число, находящееся в регистровой паре BC без привлечения стека калькулятора, что в некоторой степени облегчает обращение к ней. В качестве примера приведем такой фрагмент:

```
        ORG   60000
        ENT   $
        CALL  SETSCR
        LD    BC,867
        CALL  6683         ;печать 4-значного десятичного числа
                           ; из регистровой пары BC
        RET
SETSCR .........
```

Для любознательных можем добавить относительно подпрограммы 6683, что она используется операционной системой компьютера для вывода номеров строк бейсик-программ. Этим, в общем-то и объясняется то, что она может работать только с четырехзначными числами. Если же попытаться напечатать с ее помощью число, превышающее 9999, то ничего страшного не произойдет, просто вы получите довольно бессмысленную последовательность символов, имеющую мало общего с заданным значением.

При описании обоих методов печати чисел мы пользовались постоянными атрибутами, но это ни в коем случае не означает, что здесь недопустима установка временных цветов. Все сказанное в разделе «Вывод буквенных и цифровых символов» в полной мере относится и к числам. Вы можете устанавливать любые допустимые режимы печати, указывать координаты экрана, использовать табуляцию и так далее. Словом, с числовыми значениями вы можете обращаться в точности так, как и с отдельными символами. И чтобы подтвердить это, приведем пример программы, печатающей число 3692 в 5-й строке и 14-м столбце экрана с применением инверсии:

```
        ORG   60000
        ENT   $
        CALL  SETSCR
        LD    DE,DATA1
        LD    BC,5
        CALL  8252
        LD    BC,3692
        CALL  6683
        RET
DATA1   DEFB  22,5,14      ;управляющие коды для AT 5,14
        DEFB  20,1         ;управляющие коды для INVERSE 1
SETSCR .........
```

Если в данном примере заменить команду CALL 6683 на CALL 11563 и CALL 11747, результат окажется тем же самым.

РИСОВАНИЕ ГРАФИЧЕСКИХ ПРИМИТИВОВ

Как вы знаете, ни одна игровая программа, за исключением лишь некоторых игр жанра Adventure, не обходится без более или менее сложной графики. И даже в упомянутых текстовых играх развитие сюжета часто сопровождается различными изображениями на экране (как, например, в программе The Hobbit). В дальнейшем мы уделим достаточно внимания созданию и выводу графики, а в этом разделе только приступим к данному вопросу и начнем с построения графических примитивов - точек, прямых линий, дуг и окружностей.

Точки

Обратимся, как и прежде, к огромному вместилищу различных процедур в машинных кодах, к ПЗУ компьютера. Оператор Бейсика PLOT реализует подпрограмма, имеющая адрес 8933. Понятно, что для получения точки на экране необходимо указать, как минимум, ее координаты. Поэтому перед обращением к процедуре рисования точек следует занести в регистр C смещение по горизонтали, а в B - по вертикали, отсчитывая пиксели, как и Бейсике, от левого нижнего угла экрана. Например, для получения результата, аналогичного

выполнению оператора `PLOT 45,110` нужно вызвать такую последовательность команд микропроцессора:

```
        LD    C,45
        LD    B,110
        CALL  8933
```

Так как регистры C и B относятся к одной паре, то часто можно сокращать запись, загружая их не последовательно, а одновременно. При этом удобнее воспользоваться шестнадцатеричной системой счисления:

```
        LD    BC,#6E2D    ;45 = #2D, 110 = #6E
        CALL  8933
```

Чтобы нарисовать точку с определенными атрибутами так же, как и при выводе символов, можно воспользоваться подпрограммой **ATTRIB**, описанной в разделе «Подготовка экрана к работе из ассемблера», или установить временные цвета, изменив системную переменную `ATTR_T` (23695). Сразу заметим, что это в равной степени относится и к подпрограммам рисования линий и окружностей.

Для примера поставим точку красного цвета на желтом фоне. (Задавая атрибуты при выводе графики, помните, что в Speccy цвета определяются для целого знакоместа.) Рассчитав значения байта атрибутов для заданной комбинации цветов, получим число 50 (2+6×8). Перед выводом точки занесем это число в ячейку 23695, предварительно подготовив экран подпрограммой **SETSCR** из раздела «Подготовка экрана к работе из ассемблера»:

```
        ORG   60000
        ENT   $
        CALL  SETSCR
        LD    A,50
        LD    (23695),A
        LD    BC,#6E2D
        CALL  8933
        RET
SETSCR .........
```

Прямые линии

Продвинемся еще на один шаг вперед и научимся проводить прямые линии. Для этого потребуется сперва указать начальные координаты, поставив точку, от которой протянется линия, а затем задать величины смещения по горизонтали и вертикали до конечной точки линии. В Бейсике это должно выглядеть примерно так:

```
PLOT 120,80: DRAW 35,-60
```

Оператор `DRAW` реализует подпрограмма ПЗУ, находящаяся по адресу 9402. Перед обращением к ней в регистры C и B необходимо последовательно занести значения параметров, взятые по абсолютной величине, то есть в C в нашем примере помещается число 35, а в B нужно загрузить не -60, а 60. Но чтобы не потерять знаки, их следует разместить на регистрах E и D. Это значит, что в регистр E заносится единица, а в D - минус единица (или, что то же самое, 255). Таким образом, приведенная выше строка Бейсика на ассемблере запишется так:

```
        LD    BC,#5078    ;C = 120 (#78), B = 80 (#50)
        CALL  8933
        LD    BC,#3C23    ;C = 35 (#23), B = 60 (#3C)
        LD    DE,#FF01    ;E = 1 (#01), D = -1 = 255 (#FF)
```

```
        CALL  9402
        RET
```

Вроде бы все просто, однако здесь вы можете столкнуться с одной серьезной проблемой. Если запустить эту программку не из ассемблера (использовав директиву ENT и команду редактора **R**), а из Бейсика с помощью функции USR, то вы заметите, что компьютер ведет себя довольно странно. В лучшем случае появится какое-нибудь сообщение об ошибке, а в худшем - компьютер «зависнет» или «сбросится». А происходит это оттого, что при выполнении подпрограммы 9402 теряется некоторая информация, необходимая для нормального завершения функции USR. Значит, нам нужно выяснить, что это за информация и где она находится, чтобы можно было сохранить ее на входе и восстановить на выходе из нашей программы.

До сих пор мы говорили только о семи регистрах общего назначения, но на самом деле микропроцессор Z80 имеет их гораздо больше. Постепенно мы изучим их все, а сейчас скажем еще несколько слов об уже известных вам регистрах. Дело в том, что имеется не один, а два набора регистров данных, но активным в каждый момент времени может быть только какой-то один из них, то есть одновременно вы все равно можете работать только с семью регистрами общего назначения.

Регистры второго, или, как говорят, альтернативного набора абсолютно ничем не отличаются от регистров первого набора. Они имеют те же имена (для отличия альтернативных регистров от активных в данный момент времени после имени пары, включающей в себя этот регистр, ставят символ апострофа, например, DE') и выполняют те же функции, поэтому нет никакой возможности определить, какой из двух наборов активен в данный момент - об этом должен позаботиться программист.

В любой момент вы можете переключиться на альтернативные регистры, использовав команду EXX. Выполнив эту же команду повторно, вы вернете прежние значения регистров, ничего не потеряв. Потому данная команда, как правило, в программах встречается парами, подобно PUSH и POP.

Применяя команду EXX, нужно также помнить, что она переключает на альтернативный набор не все семь регистров, а только 6: BC, DE и HL. Для переключения аккумулятора существует другая команда. Не вдаваясь пока в смысл символики, скажем, что записывается она так:

```
        EX    AF,AF'
```

Добавим еще к сказанному, что мнемоника EX и EXX происходит от английского слова exchange - обменивать.

Вернемся снова к функции USR. В простых и небольших по объему программах семи регистров общего назначения, как правило, вполне хватает (во всяком случае, мы вам советуем не слишком злоупотреблять командой EXX, лучше при необходимости для временного хранения информации пользоваться стеком). Но в таких больших и сложных программах, как операционная система ZX Spectrum, иногда возникает необходимость привлекать и альтернативный набор регистров. Так функция USR перед вызовом программы в машинных кодах заносит важную информацию в регистровую пару HL', поэтому для нормального выхода в Бейсик ее необходимо сохранять всегда, когда она может измениться. В частности, при использовании подпрограммы рисования линий 9402.

Перепишем предыдущий пример таким образом, чтобы его можно было вызвать из Бейсика:

```
ORG   60000
EXX               ;в начале программы меняем
                  ; на альтернативный набор
PUSH  HL          ;сохраняем регистровую пару HL
LD    BC,#5078
CALL  8933
LD    BC,#3C23
LD    DE,#FF01
CALL  9402
POP   HL          ;восстанавливаем значение HL
EXX               ;делаем его альтернативным
RET
```

Поскольку перед вызовом подпрограммы в машинных кодах функция `USR` загружает регистр HL' всегда одним и тем же значением, а именно, числом 10072, то можно не сохранять его в стеке, а просто записать перед выходом в Бейсик:

```
LD    HL,10072
EXX
RET
```

При желании вы можете проверить содержимое пары HL', оттранслировав такую программку:

```
ORG   60000
EXX               ;меняем на альтернативный набор
PUSH  HL          ;запоминаем в стеке значение HL'
EXX               ;возвращаем «стандартные» регистры
POP   BC          ;забираем число из стека в пару BC
                  ; для передачи в Бейсик
RET               ;возврат в Бейсик
```

и затем выполнив ее строкой

```
PRINT USR 60000
```

в результате чего на экране должно появиться число 10072.

Дуги

Как вы знаете, у оператора `DRAW` имеется возможность рисования не только отрезков прямых линий, но и фрагментов дуг, для чего кроме двух параметров относительного смещения нужно задать еще один - величину угла, образованного дугой. В ассемблере вы вполне можете воспроизвести и эту возможность, правда, описанная выше подпрограмма с такой задачей справиться не в состоянии. Для этого придется воспользоваться процедурой, «зашитой» по адресу 9108. Что же касается передачи параметров для нее, то здесь нужно поступить примерно так же, как и при печати чисел: последовательно положить три значения в стек калькулятора, а затем вызвать саму подпрограмму.

Как пример, приведем программку, соответствующую строке Бейсика

```
PLOT 100,80: DRAW 30,50,3
```

Для занесения чисел в стек калькулятора можно, конечно, воспользоваться уже известной подпрограммой 11563, но в данном случае нам не требуются числа, превышающие байтную величину (255), поэтому программа получится короче, если в стеке калькулятора размещать значения из аккумулятора, применив процедуру ПЗУ 11560, исходным данным для которой и является содержимое регистра A. Порядок действий будет совершенно таким же, как и при использовании подпрограммы 11563. Например, для занесения в стек калькулятора числа 123 можно написать такую последовательность инструкций:

```
        LD    A,123
        CALL  11560
```

Зная это, можно написать такой фрагмент на ассемблере, соответствующий приведенной выше строке Бейсика:

```
        ORG   60000
        ENT   $
        LD    BC,#5064     ;C = 100 (#64), B = 80 (#50)
        CALL  8933         ;PLOT 100,80
        LD    A,30         ;заносим в стек калькулятора
        CALL  11560        ; первый параметр
        LD    A,50         ;второй параметр
        CALL  11560
        LD    A,3          ;третий параметр
        CALL  11560
        CALL  9108         ;DRAW 30,50,3
        LD    HL,10072     ;восстанавливаем значение пары HL'
        EXX                ; для нормального выхода в Бейсик
        RET
```

Окружности

В Бейсике имеется еще один графический оператор - `CIRCLE`, предназначенный для рисования окружностей. Значит, можно без особых хлопот, прибегнув к помощи ПЗУ, реализовать в программах на ассемблере и его. Сразу же сообщим, что соответствующая процедура находится по адресу 9005, а параметры ей передаются так же, как и при рисовании дуг, через стек калькулятора. Перед вызовом CALL 9005 нужно последовательно занести в стек три числа: координаты центра окружности по горизонтали и вертикали, а также ее радиус в пикселях.

Для примера напишем программку, выполняющую то же самое, что и оператор

```
CIRCLE 120,80,60
```

Внешне она очень напоминает программку, рисующую на экране фрагменты дуг:

```
        ORG   60000
        ENT   $
        CALL  SETSCR       ;Установка атрибутов экрана
        LD    A,120        ;Заносим в стек калькулятора
        CALL  11560        ; X-координату центра окружности
        LD    A,80         ;Y-координата центра окружности
        CALL  11560
        LD    A,60         ;Радиус
        CALL  11560
        CALL  9005         ;CIRCLE 120,80,60
        LD    HL,10072
        EXX
        RET
SETSCR .........
```

Процедура 9005 также «портит» регистровую пару HL', то есть при необходимости возврата в Бейсик ее обязательно нужно восстановить.

Еще раз повторим, что при желании вы можете рисовать все графические примитивы, описанные в этом разделе, заданным цветом, для чего нужно воспользоваться одним из способов, перечисленных в разделе «Подготовка экрана к работе из ассемблера».

СТАТИЧЕСКИЕ ЗАСТАВКИ

Сейчас в нашем распоряжении имеется уже достаточно средств, чтобы попытаться изобразить на экране нечто вполне осмысленное и несколько более привлекательное, чем просто точки или окружности.

Во время игры на экран обычно выводится масса различной информации. Играющему нет никакой необходимости задумываться над тем, к какому типу эта информация принадлежит, анализировать и классифицировать происходящее на экране. Достаточно правильно нажимать клавиши в соответствии с указаниями, данными в программе. Но совсем другое отношение ко всему происходящему должно быть у человека, планирующего свою собственную игру.

Все кадры, появляющиеся на экране во время игры, можно условно разделить на два типа: статические и динамические. К первым можно отнести такие, в которых изображение со временем не меняется. Это могут быть, например, отдельные кадры многокадровой заставки, такие как Правила Игры или Таблица Результатов. Сюда же можно отнести различные информационные панели: схемы лабиринтов, карты места боевых действий и т. п.

Динамическими заставками (или кадрами) будем называть такие, в которых с течением времени что-то изменяется на экране. В простейшем случае это может быть изменение цвета рамки или надписей, подвижный курсор, возможность ввода с клавиатуры. В более сложных динамических кадрах появятся элементы мультипликации, перемещения спрайтов и пейзажей.

О создании динамических кадров мы поговорим позже, а сейчас разберемся со способами изготовления статических заставок.

Как и при программировании на Бейсике, мы рекомендуем вам прежде всего нарисовать на листе клетчатой бумаги прямоугольник размером 32 клетки по горизонтали и 24 - по вертикали (то есть по размеру экрана в знакоместах) и внутри изобразить все то, что вы хотите увидеть на экране. А затем составить блоки данных DEFB так, чтобы их осталось только распечатать командой CALL 8252.

Рис. 4.1. Пример статической заставки

Предположим, что первая заставка выглядит, как показано на рис. 4.1. Как видно, на экране должны появляться не только тексты, но и достаточно сложные графические изображения, составляющие рамку. В Бейсике для получения такого рисунка мы воспользовались бы определяемыми пользователем символами UDG, закодировав в них отдельные элементы целого изображения. Но и в ассемблере можно поступить так же. На рис. 4.2 показаны четыре элемента, из которых состоят углы рамки, а справа от каждого квадратика расставлены десятичные значения восьми байтов изображенного символа. Таким же образом следует нарисовать и все остальные элементы, а затем рассчитать коды каждого байта в каждом из них.

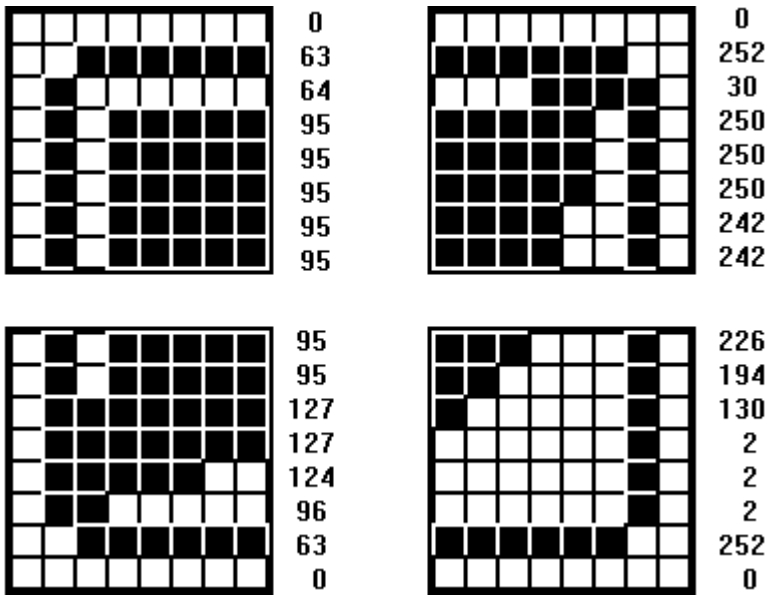


Рис. 4.2. Элементы углов рамки

Для упрощения кодирования мы можем предложить простую сервисную программку на Бейсике, перед использованием которой следует создать полный набор символов, прибегнув к помощи одного из редакторов фонтов (например, из Art Studio). Но не обязательно все

символы будут взяты из полученного шрифта, например, в нашей программе заставки их использовано только 16. Вот текст кодировщика символов (в дальнейшем везде, где в программах потребуется ввести более одного пробела подряд, для наглядности будем обозначать их символом «·».):

```
  10 BORDER 0: PAPER 0: INK 5: CLEAR 64255
  30 INPUT "Font file name:"; LINE a$
  40 LOAD a$CODE 64256
  50 INPUT "start symbol:"; LINE a$
  60 CLS : PRINT AT 15,0; INK 6;"'E' - to exit"
  70 LET a$=a$(1)
  80 LET start=64256+((CODE a$)-32)*8
  90 LET s=CODE a$: LET c=0
 100 GO TO 110
 110 FOR m=start TO 64256+768
 120 LET a=PEEK m: LET byte=a
 130 LET c=c+1
 140 DIM b(8)
 150 FOR n=1 TO 8
 160 LET a=a/2: LET b(n)=a<>INT a: LET a=INT a
 170 NEXT n
 180 PRINT AT 2,1; INK 2;"Symbol: "; INK 7; PAPER 1;CHR$ s;
     PAPER 0; INK 2;" Code: "; INK 7;s
 190 FOR n=1 TO 8
 200 IF b(n)=0 THEN PAPER 1
 210 IF b(n)=1 THEN PAPER 6
 220 PRINT AT c+3,9-n;" "
 230 NEXT n
 240 PAPER 0: PRINT AT c+3,10;byte;"··"
 250 IF c>7 THEN LET c=0: LET s=s+1: GO TO 270
 260 NEXT m
 270 REM ----Если нажата клавиша E----
 280 LET k$=INKEY$
 290 IF k$="E" OR k$="e" THEN BEEP .01,0: GO TO 50
 300 IF k$<>"" THEN BEEP .01,20: GO TO 260
 310 GO TO 280
```

Использование этой программы не вызовет никаких затруднений - достаточно отвечать на вопросы, появляющиеся на экране. Добавим только, что при работе с дисководом следует включить в программу строку

```
20 RANDOMIZE USR 15619: REM : CAT
```

а строку 40 заменить на такую:

```
40 RANDOMIZE USR 15619: REM : LOAD a$CODE 64256
```

Мы пропустим утомительный этап кодирования символов, в результате чего получится довольно длинный ряд цифр, которые запишем после инструкции DEFB и присвоим первому байту этого блока данных метку UDG:

```
UDG     DEFB  0,63,64,95,95,95,95,95           ;A (144)
        DEFB  0,252,30,250,250,250,242,242     ;B (145)
        DEFB  95,95,127,127,124,96,63,0        ;C (146)
        DEFB  226,194,130,2,2,2,252,0          ;D (147)
        DEFB  0,63,0,95,107,95,107,95          ;E (148)
        DEFB  0,244,0,208,234,208,234,208      ;F (149)
        DEFB  107,95,107,95,107,0,63,0         ;G (150)
        DEFB  234,208,234,208,234,0,244,0      ;H (151)
        DEFB  107,95,107,95,107,95,107,95      ;I (152)
        DEFB  234,208,234,208,234,208,234,208  ;J (153)
        DEFB  0,31,85,74,95,74,95,95           ;K (154)
        DEFB  0,255,85,170,255,170,255,255     ;L (155)
        DEFB  95,95,85,74,21,64,21,0           ;M (156)
```

```
        DEFB  255,255,85,170,85,0,85,0        ;N (157)
        DEFB  0,248,82,170,250,170,250,250    ;O (158)
        DEFB  250,250,82,170,80,2,80,0        ;P (159)
```

В комментариях справа от каждой строки указаны клавиши, соответствующие символам, вводимым в режиме курсора [G], и их десятичные коды. Буквы выписаны, скорее, как воспоминание о Бейсике, а вот коды нам понадобятся при составлении следующего блока, так как в редакторе GENS нет возможности вводить символы UDG непосредственно с клавиатуры.

Для получения блока данных, описывающего внешний вид рамки, за неимением какого-либо специализированного редактора лучше всего «включить» полученный набор UDG и составить программу сначала на Бейсике. При таком способе не нужно будет после каждого введенного символа транслировать программу заново и проверять правильность ввода, достаточно дать команду RUN. Кроме того, в Бейсике вы застрахованы от возникновения таких критических ошибок, после которых всю работу нужно начинать сначала. В конце останется только переписать полученный блок уже в редакторе GENS, не опасаясь, что где-то закралась ошибка.

«Включить» новые символы UDG в ассемблере до смешного просто. В Бейсике для этого требуется прежде всего определить адрес размещения символов в памяти, затем в цикле последовательно считывать коды из блока данных и переносить их по рассчитанному адресу. В ассемблере же ничего никуда перемещать не требуется, достаточно изменить адрес области UDG в системных переменных. А выполняется это всего двумя командами микропроцессора:

```
        LD    HL,UDG
        LD    (23675),HL
```

Мы избавим вас от неблагодарной работы по составлению блока данных для печати рамки и приведем его в уже готовом виде:

```
RAMKA   DEFB  22,4,0,16,5
        DEFB  144,145,154,155,155,155,155,155
        DEFB  155,155,158,154,155,155,155,155
        DEFB  155,155,155,155,158,154,155,155
        DEFB  155,155,155,155,155,158,144,145
        DEFB  146,147,16,4,156,157,157,157,157,157
        DEFB  157,157,159,156,157,157,157,157
        DEFB  157,157,157,157,159,156,157,157
        DEFB  157,157,157,157,157,159,16,5,146,147

        DEFB  16,5,148,16,4,149,22,6,30,16,5,148,16,4,149
        DEFB  16,5,152,16,4,153,22,7,30,16,5,152,16,4,153
        DEFB  16,5,152,16,4,153,22,8,30,16,5,152,16,4,153
        DEFB  16,5,150,16,4,151,22,9,30,16,5,150,16,4,151

        DEFB  16,5,148,16,4,149,22,10,30,16,5,148,16,4,149
        DEFB  16,5,152,16,4,153,22,11,30,16,5,152,16,4,153
        DEFB  16,5,152,16,4,153,22,12,30,16,5,152,16,4,153
        DEFB  16,5,150,16,4,151,22,13,30,16,5,150,16,4,151

        DEFB  16,5
        DEFB  144,145,154,155,155,155,155,155
        DEFB  155,155,158,154,155,155,155,155
        DEFB  155,155,155,155,158,154,155,155
        DEFB  155,155,155,155,155,158,144,145
        DEFB  146,147,16,4,156,157,157,157,157,157
        DEFB  157,157,159,156,157,157,157,157
        DEFB  157,157,157,157,159,156,157,157
        DEFB  157,157,157,157,157,159,16,5,146,147
```

Вы можете заметить, что кроме кодов символов UDG в блок данных включены также и управляющие коды, изменяющие цвет и позицию печати.

Значительно проще составить блок данных для печати текста заставки. Он будет выглядеть примерно так:

```
TEXT   DEFB  22,2,8,16,6
       DEFM  "*"
       DEFB  22,2,10,16,3
       DEFM  "F I G H T E R"
       DEFB  22,2,24,16,6
       DEFM  "*"
       DEFB  22,7,10,16,7
       DEFM  "Written by :"
       DEFB  22,9,7
       DEFM  "Kapultsevich···Igor"
       DEFB  22,12,5
       DEFM  "Saint-Petersburg··1994"
       DEFB  22,17,3,16,6
       DEFM  "Press any··key to continue"
```

Покончив с самой утомительной частью работы, нам остается вывести на экран подготовленные блоки данных. Вы уже знаете, как это можно сделать, но тем не менее позвольте дать небольшой совет. Все было очень просто, пока мы не сталкивались с блоками данных внушительных размеров. До сих пор мы ограничивались выводом десятка-другого символов. Но попробуйте-ка подсчитать, сколько байт занимает, например, блок под названием RAMKA. Думается, вас не очень вдохновит такая работа. А что если в текст заставки захочется внести какие-то изменения или дополнения? Снова пересчитывать его длину?

Ни в коем случае! Ведь ассемблер сам может выполнять некоторые несложные расчеты, оперируя как с числами, так и с метками. Расположим блоки данных в таком порядке: сначала блок TEXT, затем - RAMKA и самым последним запишем блок UDG. Таким образом, длина блока TEXT будет равна разности его конечного и начального адресов, а так как сразу за ним следует блок RAMKA, то его длина определяется выражением RAMKA-TEXT. (Обратите внимание на тот факт, что ассемблер может вычислять выражения с метками даже до того, как эти метки встретятся в программе. Именно для обеспечения таких «ссылок вперед» и необходим второй проход ассемблирования - *Примеч. ред.*) Аналогично длина блока RAMKA вычисляется выражением UDG-RAMKA.

Теперь можно написать программку, формирующую на экране статическую заставку:

```
       ORG   60000
       ENT   $
; Подготовка экрана
       LD    A,5
       LD    (23693),A
       LD    A,0
       CALL  8859
       CALL  3435
       LD    A,2
       CALL  5633
; «Включение» символов UDG
       LD    HL,UDG
       LD    (23675),HL
; Вывод заставки на экран
       LD    DE,RAMKA
       LD    BC,UDG-RAMKA
```

```
        CALL  8252
        LD    DE,TEXT
        LD    BC,RAMKA-TEXT
        CALL  8252
        RET
; Подготовленные заранее блоки данных
; --------------
TEXT    .........
; --------------
RAMKA   .........
; --------------
UDG     .........
```

ИЗГОТОВЛЕНИЕ НОВЫХ НАБОРОВ СИМВОЛОВ (ФОНТОВ)

Поскольку ни одна компьютерная игра не обходится без различных надписей и текстов, очень важно уметь заменять стандартный символьный набор новыми шрифтами произвольного начертания. Вы, наверное, много раз обращали внимание, что даже в фирменных играх, дошедших к нам с родины Speccy, из Англии, далеко не всегда используется встроенный шрифт, а чаще заменяется на символы более привлекательной формы, хотя в этом и нет суровой необходимости. Для простых же российских граждан, как правило, не владеющих свободно английским языком, было бы великим облегчением увидеть на экране родные русские буквы.

Многие из вас наверняка знакомы с различными видами русификации ZX Spectrum - кто-то из книг, например, из [1] или [2], а кому-то повезло раздобыть готовые знакогенераторы. Тем не менее, в этом разделе мы хотим напомнить способы создания новых фонтов и привести два полных символьных набора: латинский и русский.

В некоторых программах для русификации используется область графических символов, определяемых пользователем (UDG). Но это, в основном, касается программ на Бейсике, а в ассемблере, как вы могли заметить из примера, приведенного в предыдущем разделе, прибегать к помощи символов UDG не слишком удобно. Кроме всего прочего в этом случае возможно получить не более 22 новых символов, что для полного набора букв явно маловато. Поэтому в ассемблерных программах область определяемых символов практически никогда не используется (тем не менее, некоторое время мы еще будем к ним прибегать, пока не научимся выводить символы и графику без использования команды RST 16).

Полный набор символов можно изготовить, воспользовавшись специальным редактором шрифта (Font Editor). Эту программу, как вы знаете, можно найти в качестве составной части в таких графических редакторах как Art Studio или The Artist II. Как работать с этими программами не раз объяснялось в литературе (см. [1] или [2]) и, наверное, нет нужды еще раз распространяться на сей счет. Вместо этого мы сразу приведем программки, содержащие коды символьных наборов, созданных именно таким образом. Набрав и выполнив их, вы получите в свое распоряжение готовые кодовые блоки, которые затем можете сразу использовать в своих программах или предварительно подкорректировать в упомянутых редакторах.

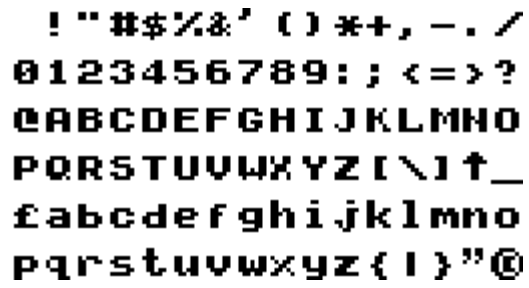


Рис. 4.3. Полный набор латинских символов

Первая программа создает полный набор, состоящий из латинских букв, цифр, знаков препинания и специальных символов (рис. 4.3). Наберитесь терпения и введите ее строки в компьютер. Львиную долю программы составляют блоки данных, содержащие коды всех символов. Если вы нечаянно ошибетесь при вводе хотя бы одного числа, программа в конце работы сообщит об этом и текстом и звуком. А чтобы ошибку легче было обнаружить, все символы, один за другим, выводятся на экран по мере выполнения программы. Если текст программы введен без ошибок, то через некоторое время на экране появится стандартное сообщение `Start tape then press any key`. Включите магнитофон на запись и нажмите любую клавишу. Коды нового фонта запишутся на ленту. Если вы работаете с дисководом, слегка измените строку 90:

```
90 RANDOMIZE USR 15619: REM: SAVE "latfont"CODE 64000,768
```

В этом случае нужно вставить дискету в карман дисковода еще до запуска программы, а полученные коды запишутся сразу по окончании обработки блока данных.

```
  10 PAPER 5: BORDER 5: CLEAR 63999: LET s=0: LET ad=64000
  20 PRINT INK 1;"Please wait"''': PLOT 0,160: DRAW INK 6;255,0:
     PLOT 0,116: DRAW INK 6;255,0
  25 RANDOMIZE ad-256: POKE 23606,PEEK 23670:
     POKE 23607,PEEK 23671
  30 FOR i=CODE " " TO CODE "": FOR j=1 TO 8
  40 READ a: LET s=s+a: POKE ad,a: LET ad=ad+1
  50 NEXT j
  60 PRINT CHR$ i;: NEXT i
  70 POKE 23606,0: POKE 23607,60
  80 IF s<>37874 THEN PRINT AT 10,8; INK 2; FLASH 1;
     "Error in DATA!!!": BEEP .5,-20: STOP
  90 SAVE "latfont"CODE 64000,768
1000 DATA 0,0,0,0,0,0,0,0: REM Space
1010 DATA 0,48,48,48,48,0,48,0: REM !
1020 DATA 0,108,108,0,0,0,0,0: REM "
1030 DATA 0,54,127,54,54,127,54,0: REM #
1040 DATA 0,8,62,104,60,22,124,16: REM $
1050 DATA 0,99,102,12,24,51,99,0: REM %
1060 DATA 0,24,44,24,58,108,58,0: REM &
1070 DATA 48,48,96,0,0,0,0,0: REM '
1080 DATA 0,12,24,24,24,24,12,0: REM (
1090 DATA 0,48,24,24,24,24,48,0: REM )
1100 DATA 0,0,54,28,127,28,54,0: REM *
1110 DATA 0,0,24,24,126,24,24,0: REM +
1120 DATA 0,0,0,0,0,48,48,96: REM ,
1130 DATA 0,0,0,0,126,0,0,0: REM -
1140 DATA 0,0,0,0,0,48,48,0: REM .
1150 DATA 0,3,6,12,24,48,96,0: REM /
1160 DATA 0,60,102,110,118,102,60,0: REM 0
1170 DATA 0,24,56,24,24,24,60,0: REM 1
1180 DATA 0,60,102,6,60,96,126,0: REM 2
1190 DATA 0,60,102,12,6,102,60,0: REM 3
```

```
1200 DATA 0,12,28,44,76,126,12,0: REM 4
1210 DATA 0,124,96,124,6,70,60,0: REM 5
1220 DATA 0,60,96,124,102,102,60,0: REM 6
1230 DATA 0,126,6,12,24,48,48,0: REM 7
1240 DATA 0,60,102,60,102,102,60,0: REM 8
1250 DATA 0,60,102,102,62,6,60,0: REM 9
1260 DATA 0,0,48,48,0,48,48,0: REM :
1270 DATA 0,0,48,48,0,48,48,96: REM ;
1280 DATA 0,0,12,24,48,24,12,0: REM <
1290 DATA 0,0,0,126,0,126,0,0: REM =
1300 DATA 0,0,48,24,12,24,48,0: REM >
1310 DATA 0,56,108,12,24,0,24,0: REM ?
1320 DATA 0,60,110,110,110,96,62,0: REM @
1330 DATA 0,60,102,102,126,102,102,0: REM A
1340 DATA 0,124,102,124,102,102,124,0: REM B
1350 DATA 0,60,102,96,96,102,60,0: REM C
1360 DATA 0,124,102,102,102,102,124,0: REM D
1370 DATA 0,126,96,124,96,96,126,0: REM E
1380 DATA 0,126,96,124,96,96,96,0: REM F
1390 DATA 0,60,102,96,110,102,60,0: REM G
1400 DATA 0,102,102,126,102,102,102,0: REM H
1410 DATA 0,60,24,24,24,24,60,0: REM I
1420 DATA 0,28,12,12,12,76,56,0: REM J
1430 DATA 0,100,104,120,104,100,102,0: REM K
1440 DATA 0,96,96,96,96,98,126,0: REM L
1450 DATA 0,99,119,107,107,99,99,0: REM M
1460 DATA 0,102,102,118,110,102,102,0: REM N
1470 DATA 0,60,102,102,102,102,60,0: REM O
1480 DATA 0,124,102,102,124,96,96,0: REM P
1490 DATA 0,60,102,102,102,124,58,0: REM Q
1500 DATA 0,124,102,102,124,108,102,0: REM R
1510 DATA 0,60,96,60,6,102,60,0: REM S
1520 DATA 0,126,24,24,24,24,24,0: REM T
1530 DATA 0,102,102,102,102,102,60,0: REM U
1540 DATA 0,102,102,102,102,60,24,0: REM V
1550 DATA 0,99,99,99,107,127,34,0: REM W
1560 DATA 0,76,76,56,56,100,100,0: REM X
1570 DATA 0,102,102,60,24,24,24,0: REM Y
1580 DATA 0,126,14,28,56,112,126,0: REM Z
1590 DATA 0,28,24,24,24,24,28,0: REM [
1600 DATA 0,96,48,24,12,6,3,0: REM \
1610 DATA 0,56,24,24,24,24,56,0: REM ]
1620 DATA 24,60,126,24,24,24,24,0: REM
1630 DATA 0,0,0,0,0,0,0,255: REM _
1640 DATA 0,28,50,120,48,48,126,0: REM J
1650 DATA 0,0,60,6,62,102,62,0: REM a
1660 DATA 0,96,96,124,102,102,124,0: REM b
1670 DATA 0,0,60,102,96,102,60,0: REM c
1680 DATA 0,6,6,62,102,102,62,0: REM d
1690 DATA 0,0,60,102,124,96,60,0: REM e
1700 DATA 0,28,48,56,48,48,48,0: REM f
1710 DATA 0,0,62,102,102,62,6,60: REM g
1720 DATA 96,96,108,118,102,102,102,0: REM h
1730 DATA 24,0,56,24,24,24,60,0: REM i
1740 DATA 12,0,14,12,12,108,44,24: REM j
1750 DATA 96,96,102,108,120,108,102,0: REM k
1760 DATA 56,24,24,24,24,24,60,0: REM l
1770 DATA 0,0,118,127,107,107,99,0: REM m
1780 DATA 0,0,108,118,102,102,102,0: REM n
1790 DATA 0,0,60,102,102,102,60,0: REM o
1800 DATA 0,0,124,102,102,124,96,96: REM p
1810 DATA 0,0,60,76,76,60,12,14: REM q
1820 DATA 0,0,92,102,96,96,96,0: REM r
1830 DATA 0,0,60,96,60,6,124,0: REM s
```

```
1840 DATA 48,48,120,48,48,54,28,0: REM t
1850 DATA 0,0,102,102,102,110,54,0: REM u
1860 DATA 0,0,102,102,102,60,24,0: REM v
1870 DATA 0,0,99,107,107,127,54,0: REM w
1880 DATA 0,0,70,44,24,52,98,0: REM x
1890 DATA 0,0,102,102,102,62,6,60: REM y
1900 DATA 0,0,126,28,56,112,126,0: REM z
1910 DATA 0,12,24,24,48,24,24,12: REM {
1920 DATA 0,24,24,24,24,24,24,0: REM |
1930 DATA 0,48,24,24,12,24,24,48: REM }
1940 DATA 108,108,36,72,0,0,0,0: REM ~
1950 DATA 60,98,221,217,217,221,98,60: REM ©
```

Получить новый набор, включающий в себя русские буквы (рис. 4.4.), поможет следующая программа. Она отличается от предыдущей фактически только блоком данных, поэтому все вышесказанное в полной мере относится и к ней.

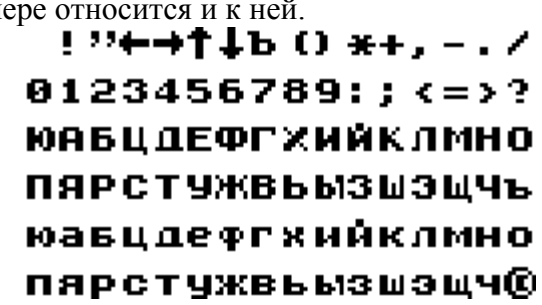


Рис. 4.4. Полный набор русских символов

```
  10 PAPER 5: BORDER 5: CLEAR 64767: LET s=0: LET ad=64768
  20 PRINT INK 1;"Please wait"''': PLOT 0,160: DRAW INK 6;255,0:
     PLOT 0,116: DRAW INK 6;255,0
  25 RANDOMIZE ad-256: POKE 23606,PEEK 23670:
     POKE 23607,PEEK 23671
  30 FOR i=CODE " " TO CODE "": FOR j=1 TO 8
  40 READ a: LET s=s+a: POKE ad,a: LET ad=ad+1
  50 NEXT j
  60 PRINT CHR$ i;: NEXT i
  70 POKE 23606,0: POKE 23607,60
  80 IF s<>43996 THEN PRINT AT 10,8; INK 2; FLASH 1;
     "Error in DATA!!!": BEEP .5,-20: STOP
  90 SAVE "rusfont"CODE 64768,768
1000 DATA 0,0,0,0,0,0,0,0: REM Space
1010 DATA 0,48,48,48,48,0,48,0: REM !
1020 DATA 0,102,102,34,68,0,0,0: REM "
1030 DATA 0,32,96,255,255,96,32,0: REM #
1040 DATA 0,4,6,255,255,6,4,0: REM $
1050 DATA 24,60,126,24,24,24,24,24: REM %
1060 DATA 24,24,24,24,24,126,60,24: REM &
1070 DATA 0,224,96,124,102,102,124,0: REM '
1080 DATA 0,6,12,12,12,12,6,0: REM (
1090 DATA 0,96,48,48,48,48,96,0: REM )
1100 DATA 0,0,54,28,127,28,54,0: REM *
1110 DATA 0,0,24,24,126,24,24,0: REM +
1120 DATA 0,0,0,0,0,48,48,96: REM ,
1130 DATA 0,0,0,0,124,0,0,0: REM -
1140 DATA 0,0,0,0,0,48,48,0: REM .
1150 DATA 0,4,12,24,48,96,64,0: REM /
1160 DATA 0,60,102,110,118,102,60,0: REM 0
1170 DATA 0,24,56,24,24,24,60,0: REM 1
1180 DATA 0,60,70,6,60,96,126,0: REM 2
1190 DATA 0,60,102,12,6,102,60,0: REM 3
1200 DATA 0,12,28,44,76,126,12,0: REM 4
1210 DATA 0,124,96,124,6,70,60,0: REM 5
```

```
1220 DATA 0,60,96,124,102,102,60,0: REM 6
1230 DATA 0,126,6,12,24,48,48,0: REM 7
1240 DATA 0,60,102,60,102,102,60,0: REM 8
1250 DATA 0,60,102,102,62,6,60,0: REM 9
1260 DATA 0,0,48,48,0,48,48,0: REM :
1270 DATA 0,0,48,48,0,48,48,96: REM ;
1280 DATA 0,0,12,24,48,24,12,0: REM <
1290 DATA 0,0,0,126,0,126,0,0: REM =
1300 DATA 0,0,48,24,12,24,48,0: REM >
1310 DATA 0,56,76,12,24,0,24,0: REM ?
1320 DATA 0,102,107,123,123,107,102,0: REM @
1330 DATA 0,60,102,102,126,102,102,0: REM A
1340 DATA 0,124,96,124,102,102,124,0: REM B
1350 DATA 0,100,100,100,100,100,126,2: REM C
1360 DATA 0,30,38,38,38,38,127,0: REM D
1370 DATA 0,126,96,124,96,96,126,0: REM E
1380 DATA 0,126,219,219,219,126,24,0: REM F
1390 DATA 0,126,98,96,96,96,96,0: REM G
1400 DATA 0,70,46,28,56,116,98,0: REM H
1410 DATA 0,102,102,110,126,118,102,0: REM I
1420 DATA 24,90,102,110,126,118,102,0: REM J
1430 DATA 0,100,104,112,120,108,102,0: REM K
1440 DATA 0,30,38,38,38,38,102,0: REM L
1450 DATA 0,99,119,107,107,99,99,0: REM M
1460 DATA 0,102,102,126,102,102,102,0: REM N
1470 DATA 0,60,102,102,102,102,60,0: REM O
1480 DATA 0,126,102,102,102,102,102,0: REM P
1490 DATA 0,62,102,102,62,102,102,0: REM Q
1500 DATA 0,124,102,102,124,96,96,0: REM R
1510 DATA 0,60,102,96,96,102,60,0: REM S
1520 DATA 0,126,24,24,24,24,24,0: REM T
1530 DATA 0,102,102,102,62,6,60,0: REM U
1540 DATA 0,153,90,126,90,153,153,0: REM V
1550 DATA 0,124,102,124,102,102,124,0: REM W
1560 DATA 0,96,96,124,102,102,124,0: REM X
1570 DATA 0,99,99,121,109,109,121,0: REM Y
1580 DATA 0,60,102,12,6,102,60,0: REM Z
1590 DATA 0,98,98,106,106,106,126,0: REM [
1600 DATA 0,60,102,14,6,102,60,0: REM \
1610 DATA 0,98,98,106,106,106,127,1: REM ]
1620 DATA 0,102,102,102,62,6,6,0: REM
1630 DATA 0,0,224,96,124,102,124,0: REM _
1640 DATA 0,0,102,107,123,107,102,0: REM J
1650 DATA 0,0,60,6,62,102,62,0: REM a
1660 DATA 0,0,124,96,124,102,124,0: REM b
1670 DATA 0,0,100,100,100,100,126,2: REM c
1680 DATA 0,0,30,38,38,38,127,0: REM d
1690 DATA 0,0,60,102,124,96,60,0: REM e
1700 DATA 0,0,60,90,90,60,24,24: REM f
1710 DATA 0,0,124,100,96,96,96,0: REM g
1720 DATA 0,0,76,108,56,108,100,0: REM h
1730 DATA 0,0,102,102,110,118,102,0: REM i
1740 DATA 24,24,102,102,110,118,102,0: REM j
1750 DATA 0,0,102,108,120,108,102,0: REM k
1760 DATA 0,0,30,38,38,38,102,0: REM l
1770 DATA 0,0,99,119,107,107,99,0: REM m
1780 DATA 0,0,102,102,126,102,102,0: REM n
1790 DATA 0,0,60,102,102,102,60,0: REM o
1800 DATA 0,0,126,102,102,102,102,0: REM p
1810 DATA 0,0,62,102,62,102,102,0: REM q
1820 DATA 0,0,124,102,102,124,96,96: REM r
1830 DATA 0,0,60,102,96,102,60,0: REM s
1840 DATA 0,0,126,24,24,24,24,0: REM t
1850 DATA 0,0,102,102,102,62,6,60: REM u
```

```
1860 DATA 0,0,219,90,60,90,219,0: REM v
1870 DATA 0,0,124,102,124,102,124,0: REM w
1880 DATA 0,0,96,96,124,102,124,0: REM x
1890 DATA 0,0,99,99,121,109,121,0: REM y
1900 DATA 0,0,60,102,12,102,60,0: REM z
1910 DATA 0,0,98,106,106,106,126,0: REM {
1920 DATA 0,0,60,102,14,102,60,0: REM |
1930 DATA 0,0,98,106,106,106,127,1: REM }
1940 DATA 0,0,102,102,62,6,6,0: REM ~
1950 DATA 60,98,221,217,217,221,98,60: REM ©
```

Теперь попробуем, используя вновь приобретенные фонты, сотворить что-нибудь полезное и продемонстрируем на практике применение новых наборов символов. Например, сделаем один из кадров заставки - страничку «Правила игры» (рис. 4.5).

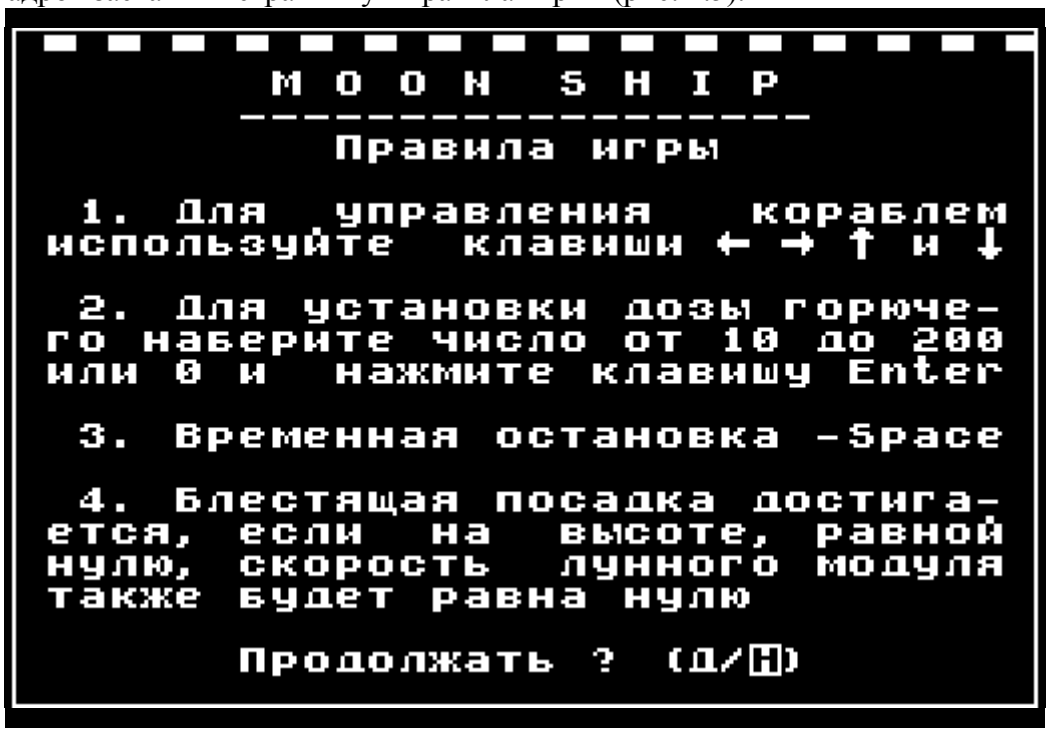


Рис. 4.5. Правила игры

После ввода исходного текста программы следует выйти в Бейсик и подгрузить в память только что полученные кодовые файлы:

```
LOAD "latfont"CODE 64000
LOAD "rusfont"CODE 64768
```

Затем оттранслируйте набранный текст и введите команду редактора **R** для исполнения машинного кода.

Вот текст программы «Правила игры»:

```
        ORG   60000
        ENT   $
LATF    EQU   64000-256
RUSF    EQU   LATF+768
        LD    A,5
        LD    (23693),A
        LD    A,0
        CALL  8859
        CALL  3435
        LD    A,2
```

```
        CALL  5633
; Печать «отрывной части блокнота»
        LD    DE,TXT
        LD    BC,TXT1-TXT
        CALL  8252
; Печать названия игры «MOON SHIP»
        LD    HL,LATF
        LD    (23606),HL
        LD    DE,TXT1
        LD    BC,TXT2-TXT1
        CALL  8252
; Печать русского текста
        LD    HL,RUSF
        LD    (23606),HL
        LD    DE,TXT2
        LD    BC,TXT3-TXT2
        CALL  8252
; Печать слова «Enter»
        LD    HL,LATF
        LD    (23606),HL
        LD    DE,TXT3
        LD    BC,TXT4-TXT3
        CALL  8252
; Печать русского текста
        LD    HL,RUSF
        LD    (23606),HL
        LD    DE,TXT4
        LD    BC,TXT5-TXT4
        CALL  8252
; Печать слова «Space»
        LD    HL,LATF
        LD    (23606),HL
        LD    DE,TXT5
        LD    BC,TXT6-TXT5
        CALL  8252
; Печать русского текста
        LD    HL,RUSF
        LD    (23606),HL
        LD    DE,TXT6
        LD    BC,END-TXT6
        CALL  8252
; Рисование «листка блокнота»
        EXX               ;сохранение HL'
        PUSH  HL
        LD    A,5
        LD    (23695),A
; Левая вертикальная линия
        LD    DE,#101     ;DRAW 169,0
        LD    BC,#A900
        CALL  9402
; Верхняя горизонтальная линия
        LD    DE,#101     ;DRAW 0,255
        LD    BC,255
        CALL  9402
; Правая вертикальная линия
        LD    DE,#FF01    ;DRAW -169,0
        LD    BC,#A900
        CALL  9402
; Нижняя горизонтальная линия
        LD    DE,#1FF     ;DRAW 0,-255
        LD    BC,255
        CALL  9402
        POP   HL          ;восстановление HL'
        EXX
; Восстановление стандартного шрифта
```

```
        LD    HL,15360
        LD    (23606),HL
        RET

; Данные для печати «отрывной части блокнота»
TXT     DEFB  22,1,0,16,5
        DEFB  32,131,32,131,32,131,32,131
        DEFB  32,131,32,131,32,131,32,131
        DEFB  32,131,32,131,32,131,32,131
        DEFB  32,131,32,131,32,131,32,131

; Данные текста правил игры
TXT1    DEFB  22,2,8,16,4
        DEFM  "M O O N··S H I P"
        DEFB  22,3,7,16,6
        DEFM  "------------------"
TXT2    DEFB  22,4,10,16,7
        DEFM  "Prawila igry"
        DEFB  22,6,2,16,2,"1",".",16,5
        DEFM  " Dlq··uprawleniq···korablem"
        DEFB  22,7,1
        DEFM  "ispolxzujte··klawi{i "
        DEFB  16,7
        DEFM  "# $ % "
        DEFB  16,5,"i"
        DEFB  16,7
        DEFM  " &"
        DEFB  22,9,2,16,2,"2",".",16,5
        DEFM  " Dlq ustanowki dozy gorJ~e-"
        DEFB  22,10,1
        DEFM  "go naberite ~islo ot 10 do 200"
        DEFB  22,11,1
        DEFM  "ili 0 i··navmite klawi{u "
TXT3    DEFB  16,3
        DEFM  "Enter"
TXT4    DEFB  22,13,2,16,2,"3",".",16,5
        DEFM  " Wremennaq ostanowka -"
TXT5    DEFB  16,3
        DEFM  "Space"
TXT6    DEFB  22,15,2,16,2,"4",".",16,5
        DEFM  " Blestq}aq posadka dostiga-"
        DEFB  22,16,1
        DEFM  "etsq, esli··na··wysote, rawnoj"
        DEFB  22,17,1
        DEFM  "nulJ, skorostx··lunnogo modulq"
        DEFB  22,18,1
        DEFM  "takve budet rawna nulJ"
        DEFB  22,20,7,16,1
        DEFM  "Prodolvatx ? ("
        DEFB  16,2,18,1,"D"
        DEFB  16,1,18,0,"/"
        DEFB  16,2,20,1,18,1,"N"
        DEFB  16,1,20,0,18,0,")"
END
```

Несколько слов о самой программе. Как вы заметили, здесь впервые встретилась директива ассемблера EQU (Equal - равный), которая служит для определения констант. Эта директива обязательно должна располагаться следом за какой-нибудь меткой, и именно этой метке в результате будет присвоено значение выражения после EQU. Таким образом, метка LATF в нашей программе становится численно равной адресу размещения латинского шрифта, уменьшенного на 256, то есть 63744, а метка RUSF принимает значение 64512.

Директиву EQU особенно удобно использовать в тех случаях, когда какая-либо постоянная величина многократно встречается в тексте программы. Если во время отладки ее потребуется изменить, то не нужно будет редактировать множество строк, достаточно будет лишь скорректировать выражение после EQU.

Есть одна особенность применения констант. Если вы используете в выражении ссылки на какие-нибудь другие имена, например

```
ENDTXT EQU   TEXT+LENTXT
```

то все они (в данном случае TEXT, и LENTXT) должны быть определены в программе до строки EQU, иначе ассемблер не сможет вычислить значение выражения и выдаст сообщение об ошибке.

Использование констант LATF и RUSF в программе «Правила игры» позволяет легко изменить при желании адреса загрузки шрифтов, а «включение» того или иного набора в тексте программы становится более наглядным. Эти «переключения» выполняют строки

```
        LD    HL,LATF
        LD    (23606),HL
```

или

```
        LD    HL,RUSF
        LD    (23606),HL
```

Здесь число 23606 - адрес системной переменной CHARS, которая указывает местоположение в памяти текущего символьного набора. Напомним, что для «включения» нового фонта необходимо прежде уменьшить его адрес на 256, а затем записать два байта полученного числа в ячейки 23606 и 23607 (младший байт, как всегда, на первом месте).

Возможно, вам не совсем понятно, зачем нужно уменьшать адрес загрузки шрифта на 256 перед занесением его в системную переменную CHARS, поэтому поясним, отчего так происходит. Назвав новые наборы символов полными, мы слегка погрешили против истины, так как на самом деле полный набор должен включать все коды от 0 до 255. Мы же пользуемся только «печатными» символами, имеющими коды от 32 до 127 включительно, то есть первые 32 символа оказываются как бы «выброшенными». Каждый символ занимает в памяти 8 байт, поэтому «реальный» адрес размещения фонта и будет равен адресу загрузки нового набора минус 32×8=256 байт. (Вообще говоря, в недрах подпрограммы печати символов из ПЗУ скрывается операция, вновь увеличивающая значение адреса на 256, но этот «тонкий» ход остается на совести разработчиков интерпретатора Бейсика для Speccy *Примеч. ред.*)

ГЛАВА ПЯТАЯ,

в которой изображения становятся подвижными

Как вы понимаете, никакая игра, даже с очень изысканной графикой, не станет достаточно интересна, если все изображения на экране будут неподвижны и персонажи не будут подавать признаков жизни. Для «оживления» картинок в программировании используется тот

же принцип, что и в мультипликации: в одно и то же место экрана последовательно помещаются слегка отличающиеся друг от друга изображения, показывающие разные фазы движения одного объекта.

В этой главе мы объясним, как заставить двигаться сначала отдельные символы, а затем и целые спрайты. А в завершение напишем полноценную динамическую заставку, которую в слегка измененном виде вполне можно использовать в какой-нибудь конкретной игре.

Как вы знаете, в программах на Бейсике для получения эффекта движения обычно используются циклы `FOR...NEXT` и операторы переходов `GO TO` и `GO SUB`, а также условный оператор `IF...THEN`, поэтому прежде всего нам необходимо выяснить, какие средства имеются в языке ассемблера для получения тех же результатов.

ОРГАНИЗАЦИЯ ЦИКЛОВ В АССЕМБЛЕРЕ

Способы организации циклов в ассемблере существенно отличаются от бейсиковского оператора `FOR...NEXT`. Здесь нет столь простых решений, зато вы можете получить множество разновидностей циклов, отвечающих любым требованиям.

Простые циклы

Рассмотрим сначала наиболее простой случай - получить заданное количество повторений некоторого фрагмента программы. Для этих целей в наборе команд микропроцессора имеется специальная инструкция DJNZ, которую можно расшифровать как «уменьшить и перейти, если не ноль». В этой команде уменьшается и проверяется на равенство нулю регистр B, который можно рассматривать в качестве счетчика количества циклов, а переход осуществляется по адресу, определенному меткой, указанной в команде DJNZ.

Проиллюстрируем цикл с использованием команды DJNZ на примере программки, печатающей на экране ряд из 32-х звездочек:

```
        ORG   60000
        ENT   $
        CALL  3435        ;Подготовка экрана к печати
        LD    A,2
        CALL  5633
        LD    B,32        ;В регистре B - количество повторений
LOOP    LD    A,"*"       ;Печать символа «*»
        RST   16
        DJNZ  LOOP        ;Уменьшение регистра B на 1 и если
                          ; B не равно 0, переход на метку LOOP
        RET
```

Такой способ организации циклов наиболее прост и удобен, но у него есть два существенных ограничения. Во-первых, как вы понимаете, количество повторений здесь не может превышать максимального значения для регистров, то есть 256 (если изначально в B записан 0), а во-вторых, между адресом начала цикла и командой DJNZ может быть расстояние не более 126 байт. Со вторым ограничением можно справиться, например, если тело цикла оформить как подпрограмму, тогда между меткой начала цикла и командой DJNZ будет находиться единственная инструкция CALL.

Еще раз напоминаем вам о необходимости сохранять регистры, которые могут быть изменены. В данном случае нужно позаботиться о сохранении регистра B, иначе цикл может никогда не закончиться, то есть произойдет «зависание» компьютера. В приведенном выше примере мы не сделали этого только потому, что команда RST 16 все регистры данных, за исключением аккумулятора, оставляет без изменений, но это не всегда справедливо при использовании других процедур ПЗУ. Если вы не совсем уверены в том, что какая-то подпрограмма сохраняет нужные регистры, лучше на всякий случай перестраховаться. Таким образом, в общем виде цикл может выглядеть так:

```
       LD    B,N          ;Указываем количество повторений
METKA  PUSH  BC           ;Сохраняем счетчик цикла
       .........          ;Тело цикла, которое может быть выделено
                          ; в отдельную подпрограмму, и тогда
                          ; здесь будет располагаться единственная
                          ; инструкция CALL
       POP   BC           ;Восстановление счетчика
       DJNZ  METKA        ;Уменьшение счетчика и если не конец,
                          ; переход на начало цикла
```

Вложенные циклы

Сохранение счетчика совершенно необходимо также при организации вложенных циклов. Приведем пример небольшой программки, заполняющей экран однородной фактурой, в которой использован принцип вложения циклов. Во внешнем цикле будем подсчитывать количество заполненных строк, а внутренний, в общем, аналогичен приведенному выше примеру:

```
       ORG   60000
       ENT   $
       LD    HL,UDG
       LD    (23675),HL  ;адресация области символов UDG
       LD    A,8
       LD    (23693),A   ;синяя «бумага», черные «чернила»
       LD    A,1
       CALL  8859        ;синий бордюр
       CALL  3435        ;подготовка экрана
       LD    A,2
       CALL  5633
       LD    B,22        ;заполняем 22 строки экрана
OUTSID PUSH  BC          ;внешний цикл
       LD    B,32        ;по 32 символа в строке
;      ---------
INSIDE LD    A,144       ;внутренний цикл
       RST   16          ;выводим код символа A из набора UDG
       DJNZ  INSIDE      ;конец внутреннего цикла
;      ---------
       POP   BC
       DJNZ  OUTSID      ;конец внешнего цикла
       RET
; Данные символа A (UDG)
UDG    DEFB  119,137,143,143,119,153,248,248
```

Проверка условий. Флаги

Иногда бывает удобнее задавать счетчик цикла не в регистре B, а в каком-то другом. Это немногим сложнее описанного способа, но тут нам понадобятся, по меньшей мере, еще две новые команды: одна для уменьшения значения выбранного регистра и другая, похожая на бейсиковский оператор `IF...THEN`, для проверки условия достижения конца цикла.

Для изменения содержимого регистров на единицу имеется два типа команд: INC (Increase) - увеличение и DEC (Decrease) - уменьшение. Попутно заметим, что эти же команды применимы не только к отдельным регистрам, но и к регистровым парам. Например, для уменьшения на 1 значения аккумулятора нужно написать команду

```
DEC   A
```

а для других регистров или регистровых пар, как вы догадываетесь, достаточно вместо A написать другое имя. Команды увеличения INC пишутся аналогичным образом.

Несколько сложнее дела обстоят с проверкой условий. Это еще один момент, имеющий со средствами Бейсика лишь очень отдаленное сходство. Чтобы объяснить, как микропроцессор реагирует на то или иное условие, прежде всего нужно ввести еще одно новое понятие - флаги.

Строго говоря, с принципом флагов вы уже встречались и в Бейсике. Вспомните, например, команду `POKE 23658,8`, которая включает режим ввода прописных букв. Флаги применяются в тех случаях, когда нужно передать какое-то сообщение из одной части программы в другую. Установив бит 3 (число 8) системной переменной `FLAGS2`, мы посылаем драйверу клавиатуры сообщение о том, что все вводимые буквы нужно переводить в верхний регистр.

Вернемся теперь к флагам микропроцессора. Здесь они имеют то же назначение и передают программе сообщения о выполнении или невыполнении различных условий. Как вы знаете, компьютер - машина бескомпромиссная и приемлет лишь два ответа: либо да, либо нет, а все остальное от лукавого. Поэтому для сообщения о принятии того или иного решения достаточно одного-единственного бита: если бит установлен в 1, то решение положительное, если он сброшен в 0, то это означает, что условие не выполнено. Другое дело, что самих проверяемых условий может быть несколько, и для каждого из них зарезервирован свой бит специального регистра, который так и называется: флаговый, или регистр F. Кстати, это и есть тот самый загадочный регистр, который образует пару совместно с аккумулятором.

Сначала перечислим все возможные условия, проверяемые микропроцессором, а затем кратко поясним каждое из них в отдельности. Вот они:

- ноль (флаг Z);
- перенос (флаг CY);
- четность/переполнение (флаг P/V);
- отрицательный результат, знак (флаг S);
- вычитание (флаг N);
- вспомогательный перенос (флаг H).

Всего, как видите, можно проверить 6 условий, и в регистре флагов F задействовано, соответственно, 6 битов, а два остались без применения. На рис. 5.1 показано, какой из битов за какое условие ответственен. Крестиками обозначены неиспользуемые биты.

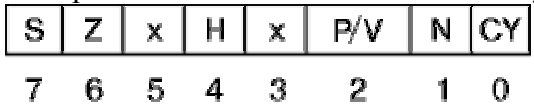


Рис. 5.1. Биты флагов

Флаг нуля Z (zero) устанавливается в том случае, если в результате выполнения последней команды получился нулевой результат. Флаг переноса CY (carry) говорит о том, что в итоге арифметической операции или же после сдвига регистра произошел перенос. Например, при сложении чисел 254 и 2 должно получиться 256, но восьми разрядов регистров хватает только для чисел от 0 до 255, поэтому реально мы получим число 0, а старший девятый бит перейдет во флаг CY. Кроме того, после такой операции будет установлен и флаг нуля. Вообще же эти два флага наиболее важны и поэтому используются чаще других. В принципе, почти всегда можно обойтись только ими, лишь изредка и, скорее, для удобства прибегая к проверке остальных флагов.

Флаг четности/переполнения P/V (parity/overflow) - единственный, на который возложена индикация не одного, а сразу двух различных условий. Это возможно потому, что они относятся к двум принципиально различным типам команд и вместе никогда не встречаются. Флаг четности устанавливается, если в результате логической операции в байте оказалось установлено в 1 четное количество битов, а условие переполнения выполняется тогда, когда после арифметического действия изменился знак операнда.

Флаг S (sign) тоже достаточно важен и после флагов Z и CY используется наиболее часто. Он устанавливается, если при выполнении арифметической или логической операции получился отрицательный результат.

Остальные два флага - N (negative) и H (half-carry) используются довольно редко и преимущественно при работе с так называемыми двоично-десятичными числами.

Надо сказать, что далеко не все команды микропроцессора оказывают какое-то влияние на флаги, а некоторые команды изменяют лишь отдельные биты флагового регистра. Например, все описанные до сих пор типы команд, за исключением упомянутых INC и DEC, вообще на флаги не воздействуют.

По мере надобности мы еще будем возвращаться к вопросу о флагах, а сейчас продолжим прерванный разговор о циклах.

Условные и безусловные переходы

Алгоритм циклов с использованием регистров, отличных от B, принципиально не отличается от порядка выполнения команды DJNZ и может быть выражен словами «уменьшить содержимое регистра и перейти на начало цикла, если не ноль». Как записывается в ассемблерной мнемонике первая часть этого предложения, вам, наверное, уже понятно: если использовать в качестве счетчика, скажем, регистр E, то нужно написать команду DEC E. Что же касается переходов, то в системе команд микропроцессора имеется не одна инструкция (как в Бейсике оператор GO TO), а целых две. Одна из них универсальна и позволяет переходить по любому выбранному адресу, другая же предназначена для более коротких переходов на расстояния, не превышающие 126-127 байт (так же, как и для команды DJNZ). Первая инструкция записывается двумя буквами JP (сокращение от Jump - перепрыгнуть), а вторая имеет мнемоническое обозначение JR (Jump Relative - относительный переход). Сначала приведем несколько примеров безусловных переходов:

```
JP    3435        ;переход по абсолютному адресу 3435
JP    LABEL       ;переход на метку LABEL
JR    $+35        ;относительный переход вперед на
                  ; расстояние в 35 байт
JR    LABEL       ;переход на ту же самую метку LABEL
```

Сразу может возникнуть вопрос, зачем для выполнения одного и того же действия нужны разные команды? Во-первых, команда JR короче JP и занимает в памяти два байта вместо трех (не улыбайтесь: такая, на первый взгляд, мелочная экономия в итоге может вылиться в килобайты!), а во-вторых, команды «коротких» переходов вы сможете оценить в полной мере, если вам когда-нибудь доведется писать программы или процедуры в машинных кодах, которые позволительно загружать по любому удобному адресу. Применение команды JR в таких подпрограммах избавит вас от необходимости выполнения предварительной их настройки на адрес загрузки. Именно так написано большинство процедур из пакетов Supercode и NewSupercode, что значительно облегчает работу с ними. Поэтому при написании собственных программ старайтесь использовать команду JR везде, где это только позволяет расстояние, оставляя команде JP переходы к дальним адресам.

Теперь относительно условных переходов. Эти команды также начинаются с JP или JR, но в поле операндов записывается мнемоника проверки одного из возможных флагов и после запятой - имя метки или абсолютный адрес, например:

```
        JP    Z,8252      ;переход по адресу 8252, если
                          ; установлен флаг нуля
        JR    NC,MAIN     ;относительный переход на метку MAIN,
                          ; если флаг переноса сброшен
```

Перечислим мнемоники всех возможных условий:

- Z - если ноль (установлен флаг нуля Z);
- NZ - если не ноль (флаг нуля Z сброшен);
- C - если перенос (установлен флаг переноса CY);
- NC - если нет переноса (флаг переноса CY сброшен);
- M - если отрицательный результат (установлен флаг знака S);
- P - если результат положительный (флаг знака S сброшен);
- PE - если четность или переполнение (установлен флаг P/V);
- PO - если нет четности/переполнения (флаг P/V сброшен).

Для флагов H и N условия отсутствуют, так как они используются только в неявном виде командами коррекции двоично-десятичных чисел.

Здесь нужно еще добавить, что в команде JP возможно применение всех перечисленных мнемоник условий, а с командой JR допускаются только первые четыре: Z, NZ, C и NC.

Зная все это, можно наконец написать цикл. В общем виде он будет выглядеть так:

```
        LD    E,N         ;заносим в регистр E счетчик
                          ; количества повторений цикла N
LOOP    PUSH  DE          ;сохраняем его в стеке
        .........         ;тело цикла
        POP   DE          ;восстановление счетчика
        DEC   E           ; и уменьшение его на единицу
        JR    NZ,LOOP     ;переход на начало цикла, если счетчик
                          ; не обнулился (в самом общем случае
                          ; здесь может находиться инструкция
                          ; JP NZ,LOOP)
```

Операция сравнения

Программируя на Бейсике, вы привыкли, что в циклах можно задавать любые граничные значения управляющей переменной. Оказывается, в ассемблере это также возможно, хотя здесь и есть ряд ограничений, касающихся выбора регистра и, естественно, диапазона границ его изменения. В этом случае в качестве счетчика удобнее всего использовать аккумулятор, что избавит от необходимости применения дополнительных команд пересылок между регистрами. Перед началом цикла нужно занести в аккумулятор стартовое значение счетчика, а в конце сравнивать его с числом, до которого он должен измениться. Команда сравнения величины регистра A (и только его!) с числовым значением или с содержимым другого регистра записывается как CP (compare - сравнить), а в поле операндов помещается число или имя регистра, например:

```
        CP    5           ;сравнить значение в аккумуляторе с числом 5
```

или

```
        CP    D           ;сравнить содержимое аккумулятора с регистром D
```

Операция сравнения исключительно важна и ее применение, конечно, далеко не ограничивается только циклами, поэтому мы сочли необходимым привести возможные результаты сравнения регистра A с операндом X и используемые при этом мнемоники условий для переходов в табл. 5.1.

Таблица 5.1. Результаты операции сравнения

Результат сравнения	Состояние флагов	Мнемоника условия перехода
A = X	Z = 1	Z
A <> X	Z = 0	NZ
Беззнаковое сравнение (числа от 0 до 255)		
A < X	CY = 0	C
A >= X	CY = 0	NC
Сравнение с учетом знака (числа от -128 до +127)		
A < X	S = 1	P
A >= X	S = 0	M

Используя операцию сравнения, можно написать цикл, в котором регистр A изменяется, например, от 12 до 24:

```
        LD    A,12        ;задаем начальное значение в аккумуляторе
CYCLE   PUSH   AF         ;сохраняем в стеке
        .........         ;тело цикла
        POP   AF          ;восстановление аккумулятора
        INC   A           ;увеличение счетчика
        CP    25          ;сравниваем содержимое регистра A
                          ; с числом 25
        JR    C,CYCLE     ;переход на начало, если меньше 25
                          ; (меньше или равно 24)
```

В данном примере ничего принципиально не изменится, если команду JR C,CYCLE заменить на JR NZ,CYCLE - результат будет тем же, но если шаг цикла окажется отличным от единицы, то второй вариант может не сработать, поэтому предпочтительнее все же применять проверку флага переноса, а не нуля.

Приведенный вариант организации циклов уже почти повторяет такие строки Бейсика:

```
FOR N=12 TO 24
..............
NEXT N
```

Основное отличие, пожалуй, состоит лишь в одном: ассемблерный цикл в любом случае выполнит хотя бы один проход, независимо от граничных условий, ведь проверка здесь находится в конце цикла. Поэтому, чтобы исключить такой нежелательный эффект, в тех случаях, когда заранее неизвестны границы цикла, необходимо проверку производить в самом начале. Это лишь немногим усложнит программу, и она примет следующий вид:

```
;Предположим, что значение аккумулятора до начала цикла неизвестно
CYCLE  CP    25          ;проверяем на достижение конечного
                         ; значения, то есть выполнение условия
                         ; A > 24 (A >= 25)
       JR    NC,AROUND   ;если да, обходим цикл
       PUSH  AF          ;сохраняем счетчик в стеке
       .........         ;тело цикла
       POP   AF          ;восстановление значения счетчика
       INC   A           ; и увеличение его на 1
       JR    CYCLE       ;безусловный переход на начало
AROUND .........         ;продолжение программы
```

Прежде чем продолжить разговор о циклах, скажем еще несколько слов об особенностях использования в ассемблере чисел со знаком. Вы, конечно, понимаете, что ни в какую ячейку памяти невозможно записать отрицательное значение. Попробуйте в редакторе GENS ввести, а затем оттранслировать команду

```
       LD    A,-1
```

после чего просмотрите память, например, функцией Бейсика PEEK. На первом месте вы увидите байт 62, это код самой команды, а следом за ним вместо -1 обнаружите число 255. Казалось бы, что с отрицательными числами в машинных кодах иметь дело совершенно невозможно, но тем не менее, можно считать, что старший бит в байте или двухбайтовом значении иногда будет играть роль знака: если он установлен, то число отрицательное, а если сброшен - положительное. Таким образом, применение чисел со знаком в ассемблере в достаточной степени условно, но все же возможно. А следить за тем, какой тип чисел применяется в каждом конкретном случае должен сам программист.

«Длинные» циклы

Нередко могут понадобиться циклы с количеством повторений значительно больше 256. На первый взгляд решение такой проблемы может показаться весьма тривиальным: достаточно использовать для размещения счетчика не отдельный регистр, а регистровую пару. Это так, но не совсем. Дело в том, что команды увеличения или уменьшения содержимого регистровых пар, оказывается, никак не влияют на флаги. Не получится использовать и операцию сравнения, разве только проверять регистры, составляющие пару, раздельно - сначала один, затем другой. Поэтому в таких случаях применяется совершенно иной подход.

Наиболее простой вариант состоит в проверке на обнуление регистровой пары: она будет содержать нулевое значение только в том случае, если оба ее регистра будут равны нулю. Иными словами, алгоритм такого цикла с использованием в качестве счетчика, для определенности, пары BC можно сформулировать так: «если регистр B не равен нулю ИЛИ регистр C не равен нулю, то перейти на начало цикла». Команда «ИЛИ» выглядит так же, как и в Бейсике - OR. Правда, в ассемблере она служит в основном совершенно для других целей, о чем мы обязательно расскажем. Но сначала приведем общий вид программы, иллюстрирующий «длинные» циклы:

```
        LD    BC,NN       ;записываем в пару BC счетчик
MET     PUSH  BC          ;сохраняем
        .........         ;выполняем тело цикла
        POP   BC          ;восстанавливаем значение счетчика
        DEC   BC          ; и уменьшаем его на единицу
        LD    A,B         ;проверка условия завершения цикла
        OR    C
        JR    NZ,MET
```

Логические операции

О команде OR стоит поговорить более подробно, так как она очень часто будет встречаться в дальнейшем. Наряду с двумя другими командами AND и XOR она относится к логическим операциям, присущим только машинному языку и не имеющим аналогии, по крайней мере, в стандартном «Спектрум-Бейсике». Эти три операции воздействуют на отдельные биты (разряды двоичного числа), изменяя их в соответствии с одним из трех возможных принципов поразрядного сложения. Сразу же скажем, что все три операции могут выполняться только на регистре A и изменяют флаги Z, P/V и S. Флаги CY и N после выполнения любой операции сбрасываются в 0 независимо от результата, а флаг H устанавливается после AND и сбрасывается после OR или XOR.

Принцип *поразрядного объединяющего ИЛИ* (OR) заключается в том, что если хотя бы в одном из двух двоичных чисел определенный разряд не равен нулю, то соответствующий разряд результата также будет ненулевым. Иными словами, если хотя бы в одном из объединяющихся байтов установлен бит, то и в результирующем байте данный бит будет установлен, а ноль получится лишь в том случае, если и там и там бит сброшен. Например, при сложении по принципу OR чисел 1001 и 1100 получится число 1101:

```
        1  0  0  1
        1  1  0  0
        ----------
        1  1  0  1
```

Теперь, наверное, должно быть понятно, что после выполнения команд

```
        LD    A,B
        OR    C
```

нулевой результат в аккумуляторе получится лишь тогда, когда оба регистра (B и C) будут содержать 0.

При сложении двух чисел по принципу *поразрядного исключающего ИЛИ* (XOR) нулевой бит может получиться в двух случаях: если соответствующий бит установлен в обоих слагаемых байтах либо если оба бита сброшены. Если же в одном из чисел бит установлен, а в другом сброшен, в результате получим установленный бит. Для тех же двух чисел результат сложения по XOR будет таким:

```
        1  0  0  1
        1  1  0  0
        ----------
        0  1  0  1
```

этот принцип используется, например, при выводе текста и графики в режиме OVER 1.

Принцип *поразрядного И* (AND) состоит в том, что 1 в каком-то бите может получиться лишь тогда, когда данный бит установлен в обоих числах. В любом другом случае в результате получится нулевой бит:

```
1  0  0  1
1  1  0  0
---------
1  0  0  0
```

Команды, использующие принцип AND, применяются, в тех случаях, когда требуется в двоичном числе выделить несколько битов, оставив их неизменными, а остальные сбросить в 0. В дальнейшем вы увидите, что такие команды очень полезны, например, при различных графических построениях.

ПЕРЕМЕЩЕНИЯ СИМВОЛОВ

В очень многих фирменных игровых программах используются те или иные способы движения отдельных букв, строк, или даже целых текстовых экранов. Так, например, в OCEAN CONQUER осуществляется побуквенный вывод текста (что-то вроде «печатающего квадрата», рассмотренного в [1]), в F-16 COMBAT PILOT можно увидеть бегущую строку, а в играх OVERLORD, ZULU WAR и многих других - скроллинги (перемещения) экранов с текстами. И в этом разделе мы рассмотрим некоторые из подобных эффектов, делающих игровые программы более интересными.

Печатающий квадрат

Чтобы создать программу этого эффектного вывода символов на экран, напоминающего работу пишущей машинки или телетайпа, достаточно воспользоваться простым циклом, содержащим команду DJNZ. Вначале два слова о том, что мы должны увидеть на экране, а затем рассмотрим особенности программы и новые команды микропроцессора, которые нам понадобятся для осуществления этого замысла.

После запуска программы слева направо начнет передвигаться синий квадратик, следом за которым будут постепенно появляться буквы заданного текста. Вывод желательно сопроводить звуком, имитирующим стук клавиш.

Поскольку такой способ печати текстов может встречаться в игровой программе неоднократно, необходимо создать универсальную процедуру, которая сможет выводить любой заданный текст. В этом случае перед ее вызовом достаточно будет лишь указать нужные параметры в некоторых регистрах. Какие это должны быть параметры? Во-первых, конечно, сам текст, а точнее, адрес строки символов. Другим параметром может быть длина выводимой строки, но от этого значения можно избавиться несколькими способами (вообще же везде, где это только возможно, нужно стараться избегать лишних параметров). Например, можно вставить байт длины строки непосредственно перед ее началом, но это не слишком удобно. Более распространенный способ - ограничить строку, вставив в ее конец какой-нибудь конкретный байт, служащий маркером конца текста. Обычно для этих целей используется байт 0 и такой формат строки называют ASCIIZ-форматом. Существуют и другие способы задания строк, но о них мы поговорим чуть позже.

Используя при выводе строк способом «печатающего квадрата» ASCIIZ-строки, перед вызовом процедуры будем заносить их адреса в регистровую пару HL. Алгоритм самой подпрограммы может быть таким: считываем из строки очередной символ и если он равен нулю, заканчиваем вывод; печатаем символ; следом выводим пробел с синим фоном и возвращаем позицию печати на один символ назад; получаем короткий звук, имитирующий стук пишущей машинки и делаем небольшую задержку; возвращаемся на начало цикла. Кроме этого, перед завершением программы нужно стереть с экрана синий квадрат, для чего достаточно просто вывести пробел.

Программа, выполняющая все эти действия будет выглядеть примерно таким образом:

```
T_TAPE LD    A,(HL)       ;читаем из строки очередной символ
       AND   A            ;проверяем на 0
       JR    NZ,TTAPE1    ;если нет, продолжаем
       LD    A," "        ;стираем изображение
       RST   16           ; «печатающего квадрата»
       RET                ;выход
TTAPE1 RST   16           ;печатаем считанный символ
       INC   HL           ;перемещаем указатель текущего
                          ; символа на следующий
       PUSH  HL           ;сохраняем в стеке значение указателя
       LD    DE,TXTCUR    ;формируем изображение
       LD    BC,4         ; «печатающего квадрата»
       CALL  8252
       CALL  3405         ;сбрасываем временные атрибуты
       XOR   A            ;в аккумуляторе 0
       OUT   (254),A      ;«выключаем» динамик
; ------------------
; Задержка (PAUSE 5)
       LD    BC,5         ;пауза 5/50 секунды
       CALL  7997         ;вызов подпрограммы PAUSE
; ------------------
       LD    A,%00010000  ;устанавливаем 4 бит аккумулятора
       OUT   (254),A      ;«включаем» динамик
       POP   HL           ;восстанавливаем значение указателя
       JR    T_TAPE       ;переход на начало цикла
TXTCUR DEFB  17,1," ",8
```

Как вы видите, в этой небольшой процедуре появилось несколько новых команд, поэтому помимо кратких комментариев дадим еще и более основательные пояснения.

Команда LD A,(HL) в принципе очень похожа на известную вам LD A,(Address), но отличается от нее тем, что в аккумулятор загружается значение не из какой-то конкретной ячейки памяти, а из той, на которую указывает регистровая пара HL. Например, если в HL записать число 16384, то в регистр A загрузится содержимое ячейки, находящейся по адресу 16384, то есть выполнится команда LD A,(16384). Эту инструкцию очень удобно применять в тех случаях, когда значение адреса для чтения или записи заранее неизвестно и может быть любой переменной величиной. Попутно скажем, что существует и обратная команда, то есть LD (HL),A, которая записывает содержимое аккумулятора по адресу, указанному в HL.

Существенное преимущество этого типа команд состоит еще и в том, что кроме аккумулятора таким способом можно пересылать и содержимое любого другого регистра данных, в том числе и регистров H и L, например, LD L,(HL) или LD (HL),H. И даже более того, по адресу, указанному в HL, можно записывать не только содержимое регистров, но и непосредственные числовые значения, конечно, не превышающие 255, скажем, LD (HL),153. Что же касается аккумулятора, то для него имеется возможность адресации не только с помощью пары HL, но также и BC либо DE, и эти команды еще не раз появятся в нашей книге.

В приведенной процедуре встретились две логические команды AND A и XOR A, хотя в данном примере они применены и не совсем по назначению, а скорее для сокращения машинного кода. Первая из них выполняет то же действие, что и команда CP 0, а вторая заменяет инструкцию LD A,0 и такой способ записи среди программистов считается хорошим стилем. В самом деле, при объединении аккумулятора по принципу AND с самим собой, ноль может получиться лишь тогда, когда он имеет нулевое значение (кстати, с тем же успехом можно пользоваться инструкцией OR A). Весьма ценно и то, что при этом содержимое аккумулятора не изменяется, а команда воздействует только на флаги, и в частности, на флаг Z, который нам и нужно проверить. Другая команда, XOR A, обнулит содержимое аккумулятора при любом его изначальном значении. Ведь, как мы уже говорили, при объединении двух чисел по принципу XOR любой бит будет сброшен в 0, если соответствующие биты одинаковы в обоих числах. То есть при объединении по XOR двух одинаковых величин результат всегда будет нулевым. Запомните это обстоятельство, поскольку в дальнейшем мы всегда будем пользоваться описанными приемами. Единственный случай, когда команда LD A,0 незаменима, это если нужно сохранить значение флагового регистра для последующей проверки. Помните, что все логические команды воздействуют на флаги, а инструкции загрузки регистров - нет.

Вы, наверное, знаете, что для получения звука в Бейсике, кроме оператора `BEEP`, можно воспользоваться командой `OUT`, которая записывает число в указанный порт. Динамик в Spectrum'е подключен к порту 254, а за его «включение» или «выключение» ответственен 4-й бит посылаемого байта. Остальные биты несут другую нагрузку, а здесь нас интересуют еще биты 0, 1 и 2, которые, как уже было сказано в разделе «Подготовка экрана к работе из ассемблера» главы 4, определяют цвет бордюра. При заданных в процедуре значениях аккумулятора 0 и 16 получится черный бордюр, но его легко изменить на любой другой цвет, просто прибавив к коду нужного цвета значение «бита для динамика» (точнее, код цвета объединяется со значением 4-го бита аккумулятора по принципу OR, для чего, кстати, логические команды в основном и используются). Быстрое чередование значения этого бита приводит к появлению звука. Более подробно о получении различных звуковых эффектов мы расскажем в главе 10.

В процедуре «печатающий квадрат» показано применение еще одной полезной подпрограммы ПЗУ, которая вызывается при интерпретации оператора `PAUSE`. Перед обращением к ней необходимо в пару BC поместить величину задержки, измеряемую в 50-х долях секунды - так же, как и в Бейсике. Если в BC поместить 0, то получится бесконечная пауза, прерываемая только при нажатии на любую клавишу (опять же, как в Бейсике).

Теперь приведем пример использования данной процедуры и напечатаем на экране таким способом какой-нибудь конкретный текст. Например:

```
        ORG   60000
        ENT   $
        CALL  SETSCR      ;подготавливаем экран
        LD    A,22        ;помещаем текущую позицию печати в
                          ; начало 5-й строки экрана (AT 5,0)
        RST   16
        LD    A,5
        RST   16
        XOR   A
        RST   16
        LD    HL,TEXT     ;указываем адрес ASCIIZ-строки с текстом
        CALL  T_TAPE      ;выводим методом телетайпа
        RET
TEXT    DEFM  "COMPUTER Sinclair ZX-Spectrum"
        DEFB  0
```

```
SETSCR .........        ;подпрограмма из раздела «Подготовка
                        ; экрана к работе из ассемблера» главы 4
T_TAPE .........
```

Бегущая строка

Теперь покажем, как можно создать на ассемблере еще один эффект, который, несмотря на относительную простоту реализации, пользуется в игровых программах довольно большой популярностью. В книге [1] мы показывали, как добиться эффекта бегущей строки средствами Бейсика, однако интерпретатор не позволил нам добиться плавного движения букв. В ассемблере же можно без особого труда устранить этот недостаток, правда, для этого нам придется познакомиться еще с одним типом команд, выполняющих различные виды сдвигов битов в регистрах или в памяти.

Смысл сдвигов заключается в том, что все биты, не меняя своего относительного положения, смещаются вправо или влево. В зависимости от типа команды уходящие «за край» биты могут появляться с противоположной стороны (циклические сдвиги) либо теряться (нециклические или простые сдвиги). Например, после циклического сдвига влево числа 11001001 получится результат 10010011, а после простого сдвига вправо этого же значения - ?1100100. Знак вопроса на месте 7-го бита означает, что его значение зависит от результата предыдущей операции, а точнее, от состояния флага переноса CY; если он был установлен, то в 7-м бите появится единица, в противном случае - 0.

Существуют четыре команды для смещения битов в регистре A:

1. **RLCA** - циклический сдвиг влево. После выполнения этой команды старший бит переходит в младший и дублируется во флаге переноса CY (то есть он будет установлен, если 7-й бит перед выполнением команды был в 1 и сброшен, если 7-й бит имел нулевое значение).
2. **RRCA** - циклический сдвиг аккумулятора вправо. Эта команда в точности противоположна предыдущей: младший бит переходит в старший и повторяется во флаге CY.
3. **RLA** - это тоже циклический сдвиг, но не совсем обычный. В этой команде флаг переноса рассматривается как еще один дополнительный бит аккумулятора: 7-й бит перемещается в CY, а предыдущее значение флага переноса переносится в младший бит.
4. **RRA** - эта команда аналогична предыдущей с тем отличием, что движение битов происходит в обратном направлении, то есть слева направо.

На рис. 5.2 показаны схемы направлений перемещения битов во всех четырех перечисленных командах. После их выполнения все основные флаги (кроме CY, конечно) остаются без изменений, а во флагах N и H появляются нули.

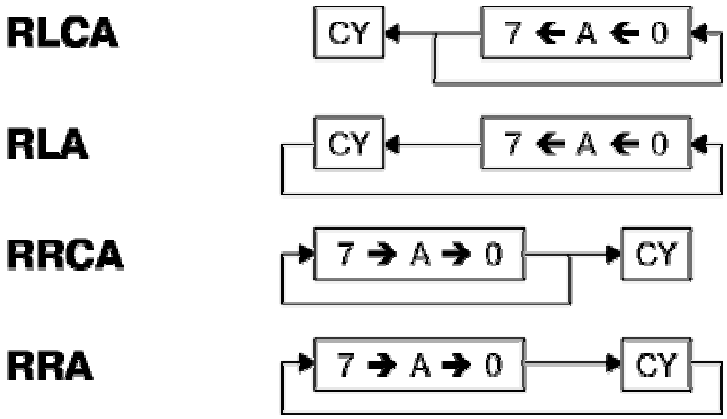


Рис. 5.2. Выполнение команд сдвига аккумулятора

Похожие сдвиги могут быть получены и с другими регистрами (в том числе и с аккумулятором), либо в ячейке памяти, адресованной регистровой парой HL. Вот эти команды:

Команда RLC S выполняет действие, аналогичное RLCA. В качестве операнда S могут использоваться регистры A, B, C, D, E, H или L, а также адрес (HL), если записать инструкцию RLC (HL). Во всех остальных командах можно применять тот же набор операндов (немного опережая события, добавим, что во всех этих командах могут участвовать также и ячейки памяти, адресованные индексными регистрами, но поскольку мы их еще не рассматривали, применять такие инструкции пока не будем).

Инструкция RRC S выполняется аналогично команде RRCA, но помимо аккумулятора применима к любым перечисленным операндам.

Команды RR S и RL S подобны описанным командам RRA и RLA соответственно.

Весьма широко применяются команды SLA S (сдвиг влево) и SRL S (сдвиг вправо). Они отличаются от прочих команд сдвигов тем, что освобождающийся бит при любых условиях заполняется нулевым значением, в результате чего их вполне можно рассматривать как операции умножения или деления на 2 соответственно. «Вытесняемый» бит при этом, как и при других сдвигах переходит во флаг переноса.

Еще одна операция «деления на 2 со знаком» выполняется командой SRA S, которая смещает вправо только 7 младших битов, а старший оставляет без изменения. Младший бит, как и положено, вытесняется во флаг CY. Все команды этой группы воздействуют уже не только на флаг переноса, но и на все прочие. Флаги H и N, так же, как и при сдвигах аккумулятора, сбрасываются в 0.

На рис. 5.3 приведены схемы всех перечисленных сдвигов. Рассмотрите этот рисунок внимательно, и вам, наверное, уже станет понятно в общих чертах, как можно получить эффект плавно движущейся строки. Теперь остается, пожалуй, только одна сложность: научиться быстро и безошибочно определять адреса в экранной области, соответствующие началу любой строки. Конечно, вы можете воспользоваться рис. 2.3 из второй главы, но тогда не сможете написать универсальной подпрограммы «на все случаи жизни».

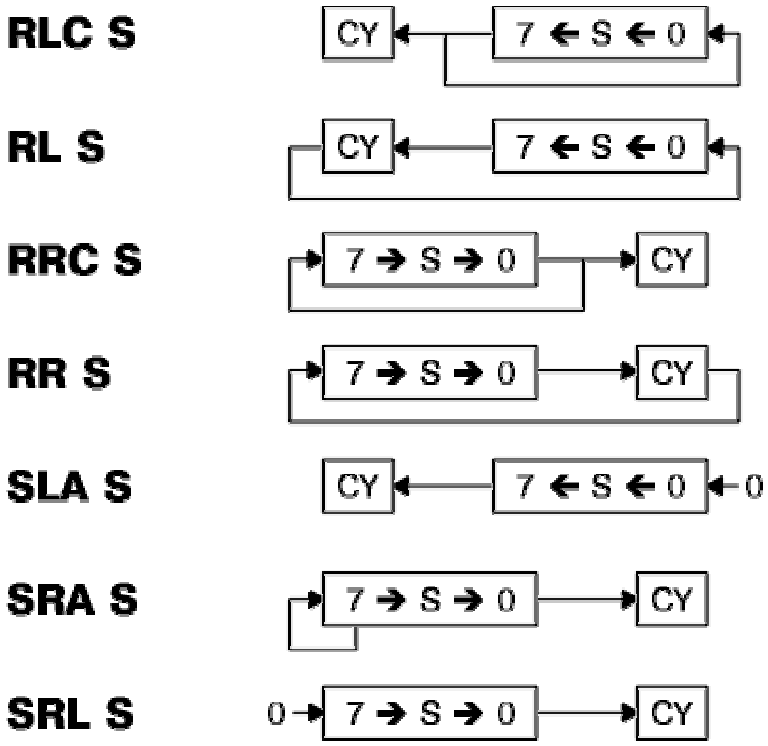


Рис. 5.3. Выполнение команд сдвига регистров

Алгоритм расчета адресов экрана достаточно сложен, поэтому, как и прежде, обратимся за помощью к ПЗУ, благо в «прошивке» Speccy имеются необходимые процедуры. Для получения начального адреса любой строки экрана можно обратиться к подпрограмме, расположенной по адресу 3742. Перед обращением к ней в аккумулятор необходимо поместить номер строки экрана. На выходе в регистровой паре HL получится искомый адрес, зная который, уже несложно рассчитать и любой другой адрес в пределах данной строки. Каждая строка имеет длину 32 байта, которые расположены последовательно, и вмещает 8 рядов пикселей. При переходе к следующему ряду адрес видеобуфера увеличивается на 256, то есть увеличивается только старший байт адреса, а младший остается без изменений.

Для примера покажем, как рассчитать адрес второго байта сверху в 5-й строке и 11-й позиции экрана (иначе, в позиции печати, определяемой директивой Бейсика AT 5,11):

```
        LD    A,5         ;номер строки
        CALL  3742        ;получаем в HL начальный адрес
        LD    A,L         ;берем значение младшего байта адреса
        OR    11          ;добавляем смещение в 11 байт (знакомест)
        LD    L,A         ;возвращаем в младший байт
        INC   H           ;увеличиваем адрес на 256 и тем самым
                          ; получаем адрес второго байта в
                          ; знакоместе сверху
```

Теперь можно написать программу, дающую эффект бегущей строки. Для определенности будем скроллировать 21-ю строку экрана:

```
        ORG   60000
        LD    A,21        ;21-я строка экрана
SCRLIN CALL  3742        ;получаем ее адрес в HL
; Так как строка должна бежать слева направо, то раньше нужно сдвигать
;  последние байты, поэтому определяем адрес конца строки
        LD    A,L
        OR    31
```

```
        LD    L,A
        LD    C,8          ;высота строки 8 пикселей
SCRL1   LD    B,32         ;длина строки 32 байта
        AND   A            ;очистка флага CY
        PUSH  HL           ;сохраняем адрес
SCRL2   RL    (HL)         ;последовательно сдвигаем все байты
        DEC   HL
        DJNZ  SCRL2
        POP   HL           ;восстанавливаем адрес
        INC   H            ;переходим к следующему ряду пикселей
        DEC   C            ;повторяем
        JR    NZ,SCRL1
        RET
```

Но это еще не все, ведь данная процедура сдвинет строку только на один пиксель влево, а для перемещения ее на знакоместо потребуется выполнить приведенную подпрограмму 8 раз. Однако прежде чем мы продолжим создание полноценного эффекта, поясним смысл некоторых использованных команд.

Возможно, вам не совсем ясно, что в данной подпрограмме делает инструкция AND A. Как сказано в комментарии к этой строке, она очищает флаг переноса. Собственно, это и все, что нам от нее требуется, но для чего это нужно? Посмотрите на схему перемещения битов командой RL S и увидите, что при ее выполнении бит из CY переходит в младший бит операнда, в то время как старший сохраняется во флаге переноса. Этим и обусловлен выбор именно команды RL (HL), ведь нам нужно скроллировать не отдельный байт, а целую цепочку байтов, значит, вытесняемый бит должен быть сохранен для следующей команды сдвига. Но сдвигая самый первый байт в цепочке, мы должны убедиться, что в младшем бите появится 0, поэтому и нужно сбросить флаг CY. Если этого не сделать, то в конце концов в скроллируемой строке может появиться какой-то нежелательный «мусор» в виде «включенных» пикселей.

Надеемся, что этих объяснений достаточно, а если нет, то попытайтесь мысленно проследить, что происходит в результате выполнения команды RL (HL) на каждом «витке» цикла, обозначенного меткой SCRL2, какие биты и куда при этом сдвигаются.

А сейчас напишем небольшую тестовую программку на Бейсике, проверяющую работоспособность нашей процедуры. Постарайтесь после этого самостоятельно переписать ее на ассемблере, а когда справитесь с задачей, перелистните несколько страниц и проверьте себя, сравнив полученный результат с ответом, данным в конце этого раздела.

```
10 INK 6: PAPER 0: BORDER 0: CLS
20 LET a$="Examine yourself how you know the assembler!"
30 FOR i=1 TO LEN a$
40 PRINT AT 21,31; INK 0; a$(i)
50 FOR j=1 TO 8
60 RANDOMIZE USR 60000
70 NEXT j
80 NEXT i
90 FOR i=1 TO 256: RANDOMIZE USR 60000: NEXT i
```

Используя команды сдвигов, можно придумать великое множество интересных эффектов. Приведем маленький пример наиболее простого из них - циклического скроллинга отдельного знакоместа, и предоставим вам возможность пофантазировать и развить эту идею.

```
        ORG   60000
        LD    B,8
        LD    HL,16384
ROL     RRC   (HL)
        INC    H
```

```
        DJNZ    ROL
        RET
```

Чтобы посмотреть, как эта программка работает, напечатайте в Бейсике в левом верхнем углу экрана какой-нибудь символ и в цикле вызывайте процедуру. Изображение должно многократно «провернуться» вокруг вертикальной оси, причем уходящие вправо точки будут вновь появляться с левого края. Если заменить команду RRC (HL) на RLC (HL), то скроллинг будет выполняться в обратную сторону.

«Волна»

Предлагаем вам еще один интересный эффект, иногда встречающийся в игровых программах: по строке текста от левого к правому краю экрана как бы пробегает волна, буквы приподнимаются, затем опускаются и наконец занимают свое первоначальное положение.

Для осуществления этого эффекта прежде всего потребуется написать две подпрограммы вертикального скроллинга отдельных знакомест экрана: одну для перемещения символа вверх, а другую - для движения изображения вниз. Принцип такого рода скроллингов исключительно прост. Например, для сдвига изображения на один пиксель вверх нужно байт из второго ряда пикселей перенести на место байта первого ряда, затем байт из третьего ряда поместить во второй и так далее, а самый нижний ряд пикселей заполнить нулями. Правда, при этом потеряется содержимое самого верхнего байта, но для текстовой строки это не особенно важно, так как обычно буквы сверху и снизу имеют по пустому ряду точек (ведь между строками текста должен быть какой-то зазор). Скроллинг букв вниз аналогичен, только перемещать байты знакоместа нужно, начиная с нижнего края.

Подпрограмма вертикального скроллинга знакоместа может выглядеть так:

```
UP      CP    32          ;проверка позиции перемещаемого знакоместа
        RET   NC          ;выход, если больше или равна 32
        LD    HL,(AD_LIN) ;получаем адрес экрана начала строки
        PUSH  AF
        OR    L
        LD    L,A
        PUSH  HL
        LD    D,H         ;копируем адрес в DE
        LD    E,L
        LD    B,7         ;повторяем 7 раз
UP1     INC   H           ;в HL - адрес байта следующего ряда
        LD    A,(HL)      ;переносим из (HL) в (DE)
        LD    (DE),A
        INC   D           ;переходим к следующему ряду
        DJNZ  UP1
        LD    (HL),0      ;обнуляем самый нижний ряд
        POP   HL
        POP   AF
        RET
```

Перед обращением к этой (а также и к следующей) подпрограмме в аккумулятор нужно занести горизонтальную позицию знакоместа в строке. Если заданная позиция выходит за пределы экрана, то есть не попадает в диапазон от 0 до 31, подпрограмма не должна выполняться. Для этого в самом ее начале проверяется обозначенное условие и в случае его невыполнения происходит условный выход из подпрограммы (команда RET NC). Как видите, условными могут быть не только переходы. Сразу заметим, что по условию можно также вызывать процедуры, используя команды типа

```
        CALL  S,ADDR
```

где S - любое из возможных условий, а ADDR - абсолютный адрес или метка.

После проверки возможности выполнения подпрограммы в регистровую пару HL загружается адрес начала строки экрана, по которой будет пробегать «волна». Этот адрес рассчитывается заранее и помещается в двухбайтовую переменную AD_LIN, которая в программе будет задаваться инструкцией ассемблера DEFW. В остальных строках не встретилось ничего нового, поэтому, надеемся, вполне достаточно кратких комментариев.

Подпрограмма скроллинга вниз похожа на первую, но все же есть и некоторые отличия. Мы говорили, что начинать такой скроллинг нужно с нижнего края знакоместа, однако можно поступить и несколько иначе, например, так:

```
DOWN    CP    32          ;начало такое же, как и в
        RET   NC          ; предыдущей подпрограмме
        LD    HL,(AD_LIN)
        PUSH  AF
        OR    L
        LD    L,A
        PUSH  HL
        XOR   A           ;в аккумуляторе 0
        EX    AF,AF'      ;отправляем его в альтернативный AF'
        LD    B,7
DOWN1   LD    A,(HL)      ;считываем байт текущего ряда
        EX    AF,AF'      ;меняем аккумулятор на альтернативный
        LD    (HL),A      ;записываем в текущий ряд
        INC   H           ;переходим к следующему ряду
        DJNZ  DOWN1
        EX    AF,AF'      ;запись последнего байта
        LD    (HL),A      ; в самый нижний ряд
        POP   HL
        POP   AF
        RET
```

Раньше мы уже упоминали команду EX AF,AF', сейчас же показали ее практическое применение. Напомним, что она выполняет действие, аналогичное команде EXX, только меняет на альтернативный аккумулятор, а заодно и флаговый регистр (этим иногда тоже можно пользоваться, сохраняя флаги для последующих операций). Советуем внимательно изучить подпрограмму DOWN и проследить за «эволюциями» аккумулятора в данном примере. Это поможет вам лучше понять принцип работы многих других подпрограмм, с которыми вы еще встретитесь.

После того, как основные процедуры скроллингов созданы, можно приступить к написанию подпрограммы, формирующей саму «волну». Алгоритм создания этого эффекта совсем несложен: первый символ, находящийся в «голове» синусоиды нужно сдвинуть, например, вверх, тогда следующие два пойдут вниз и последний - снова вверх. Получив таким образом синусоиду, смещаем ее начало на одну позицию и повторяем все с самого начала до тех пор, пока «волна» не пройдет по всему экрану и не скроется за его пределами. Вот программа, создающая на экране описываемый эффект:

```
        ORG   60000
        XOR   A           ;инициализация переменных:
        LD    (HEAD),A    ; начальной позиции «волны»
        CALL  3742
        LD    (AD_LIN),HL ; и адреса строки экрана
WAVE    LD    HL,HEAD
        LD    A,(HL)
        INC   (HL)        ;увеличивать или уменьшать можно не
```

```
                        ; только содержимое регистров или
                        ; регистровых пар, но и значение в
                        ; ячейке памяти, адресованной парой HL
        CP    35        ;ушла ли «волна» за пределы экрана?
        RET   Z
        CALL  UP        ;первый символ вверх
        DEC   A
        CALL  DOWN      ;второй - вниз
        DEC   A
        CALL  DOWN      ;третий тоже вниз
        DEC   A
        CALL  UP        ;последний - вверх
        LD    BC,5
        CALL  7997      ;небольшая задержка (PAUSE 5)
        JR    WAVE
HEAD    DEFB  0         ;позиция «головы» синусоиды
AD_LIN  DEFW  0         ;адрес экрана начала строки
UP      .........
DOWN    .........
```

Данная подпрограмма создает эффект «волны» в самой верхней строке экрана, а чтобы получить то же самое в другой строке, нужно перед командой

```
        CALL  3742
```

добавить

```
        LD    A,N
```

где N - номер требуемой строки.

Подпрограмма в таком виде пригодна для вызова из Бейсика. Напечатайте в верхней строке какой-нибудь текст, желательно большими буквами, а затем выполните оператор

```
RANDOMIZE USR 60000
```

Если же вы захотите все необходимые приготовления выполнить в ассемблере, то у вас уже достаточно знаний, чтобы справиться с этой задачей самостоятельно и без особых трудностей.

«Растворение» символов

Продолжим изучение новых команд и приемов программирования, а заодно рассмотрим еще один интересный эффект, который можно назвать «растворение» символов. Он используется, например, в таких играх, как LODE RUNNER, THE DAM BUSTER и других. Возможно, он заинтересует и вас. Суть его состоит в том, что при переходе от одной картинки, формируемой программой, к другой происходит не мгновенная очистка экрана, как оператором CLS, а изображение постепенно как бы растворяется. Это очень напоминает таяние снега. Подпрограмма, создающая такой эффект удивительно проста и коротка:

```
        ORG   60000
        ENT   $
THAW    LD    B,8         ;экран очищается за 8 циклов
        LD    DE,0        ;адрес начала кодов ПЗУ
THAW1   LD    HL,#4000    ;адрес начала экранной области
        PUSH  DE
THAW2   LD    A,(DE)      ;берем «случайный» байт из ПЗУ
        AND   (HL)        ;объединяем с байтом из видеобуфера
        LD    (HL),A      ;помещаем обратно в видеобуфер
        INC   HL          ;переходим к следующим адресам
        INC   DE
```

```
LD    A,H         ;проверяем, нужно ли повторять цикл
CP    #58         ;если прошли еще не весь видеобуфер
                  ; (#5800 - адрес начала области атрибутов)
JR    NZ,THAW2    ; то повторяем
PUSH  BC
LD    BC,1
CALL  7997        ;PAUSE 1
POP   BC
POP   DE
LD    HL,100
ADD   HL,DE       ;увеличиваем адрес в ПЗУ на 100
EX    DE,HL       ;меняем HL на DE
DJNZ  THAW1       ;повторяем цикл
JP    3435        ;окончательно очищаем экран
```

Сначала объясним смысл вновь встретившихся команд, а затем более подробно расскажем о принципе работы программы.

Как мы говорили в самом начале книги, микропроцессор способен выполнять элементарные арифметические операции над числами. Сложению соответствует мнемоника ADD (Addition - сложение) а вычитанию - SUB (Subtraction). В этих операциях может участвовать регистр A (арифметические операции над однобайтовыми числами) или пара HL (при сложении или вычитании двухбайтовых чисел). К содержимому аккумулятора можно прибавлять (или вычитать) значение другого регистра или непосредственную числовую величину, а к паре HL можно только прибавлять и только содержимое другой регистровой пары (кроме AF, конечно). Результат арифметического действия получается в аккумуляторе или в регистровой паре HL соответственно.

Помимо обычных операций сложения и вычитания существуют их разновидности - сложение и вычитание с переносом (или, как еще говорят, с заемом). Они отличаются тем, что в операции принимает участие еще и флаг переноса: при сложении он прибавляется к результату, а при вычитании - отнимается. Записываются такие команды с мнемоникой ADC или SBC. В отличие от обычного вычитания в операции вычитания с переносом регистровая пара HL вполне может участвовать.

Есть небольшая особенность в записи этих команд: операция вычитания с участием аккумулятора выглядит не SUB A,S или SBC A,S, как это можно было ожидать, а просто SUB S или SBC S. То есть имя регистра A не пишется. Во всех остальных командах нужно указывать и имя. Вот некоторые примеры:

```
ADD  A,7     ;прибавить к содержимому аккумулятора 7
ADC  A,C     ;прибавить к аккумулятору регистр C и флаг CY
SUB  B       ;вычесть из аккумулятора значение регистра B
SBC  127     ;вычесть из аккумулятора число 127 и флаг CY
ADD  HL,HL   ;удвоить значение регистровой пары HL
ADC  HL,DE   ;сложить содержимое пар HL и DE и добавить
             ; значение флага CY
SBC  HL,BC   ;вычесть с учетом флага переноса BC из HL
```

Все перечисленные команды изменяют основные флаги за исключением команды ADD HL,SS, которая влияет лишь на флаг переноса.

Другая новая команда, встретившаяся в подпрограмме «растворения» экрана - это команда EX DE,HL. Она выполняет самое элементарное действие: обменивает содержимым регистровые пары HL и DE. То, что раньше было в HL, переходит в DE и наоборот. Правда, в данном случае она использована просто для пересылки полученного в HL результата в пару DE. Такой, вроде бы, нелепый ход объясняется отсутствием в системе команд

микропроцессора Z80 инструкций для пересылки значений между регистровыми парами (типа LD DE,HL), поэтому команда EX DE,HL иногда может заменять последовательность

```
LD    D,H
LD    E,L
```

что не только сокращает запись, но и несколько ускоряет работу программы.

Теперь вернемся к нашему примеру и посмотрим, как он работает. Основную роль здесь выполняет команда AND (HL), объединяющая байт из ПЗУ с байтом из экранной области памяти. В данном случае коды ПЗУ можно рассматривать как некоторую последовательность «случайных» чисел, поэтому в результате операции AND мы получим в аккумуляторе байт из видеобуфера, но некоторые биты в нем окажутся «выключенными», а какие именно - предсказать довольно трудно. Каждый следующий байт экрана объединяется с другим байтом из ПЗУ, отчего уже после первого прохождения цикла часть изображения пропадет. После второго прохода на экране останется еще меньше «включенных» пикселей. Но для этого нужно изменить последовательность «случайных» чисел, чего легче всего добиться изменением начального адреса в ПЗУ. В нашем примере он просто увеличивается на 100 байт. Поскольку нет гарантии, что после установленных восьми циклов все изображение окончательно исчезнет, в самом конце программа переходит на процедуру полной очистки экрана. Кстати, обратите внимание, что вместо последовательности

```
CALL  3435
RET
```

стоит единственная команда

```
JP    3435
```

Это позволительно делать практически всегда, когда за командой безусловного вызова следует безусловный же выход из подпрограммы. Исключение составляют лишь некоторые особые случаи, о которых мы поговорим, когда в этом возникнет необходимость.

В подпрограмме «растворения» экрана есть еще один довольно интересный момент - это способ организации внутреннего цикла. Зная длину области данных видеобуфера, которая равна 6144 байт, можно было бы задать количество повторений в явном виде, но в данном случае, поскольку адрес экранной области начинается с ровного шестнадцатеричного значения (то есть младший байт адреса равен нулю) и размер обрабатываемого блока также кратен 256, достаточно проверять только старший байт адреса на достижение определенного значения, а именно, #58, так как с адреса #5800 начинается область хранения атрибутов для каждого знакоместа экрана.

Ответ на задачу

А сейчас, как мы и обещали, даем ответ на задачу, поставленную в параграфе «Бегущая строка» и приводим текст программы на ассемблере, соответствующий предложенному фрагменту на Бейсике. Не стоит очень расстраиваться, если вы обнаружите в чем-то расхождения, ведь любую, даже очень небольшую программу, можно написать тысячью и одним способом. Поэтому, если ваша программа работает, можете считать, что с поставленной задачей вы справились.

```
ORG   60000
ENT   $
LD    A,6         ;подготовка экрана
```

```
        LD    (23693),A
        XOR   A
        CALL  8859
        CALL  3435
        LD    A,2
        CALL  5633
        LD    HL,TEXT     ;адрес текстовой строки
MAIN1   LD    DE,PR_AT    ;позиционирование курсора, черный
        LD    BC,5        ; (совпадающий с фоном) цвет символов
        CALL  8252
        LD    A,(HL)      ;чтение очередного символа строки
        AND   A
        JR    Z,MAIN3     ;если 0, закончить вывод
        RST   16
        INC   HL
        PUSH  HL
; Восьмикратное (по ширине символов в пикселях) скроллирование строки влево
        LD    B,8
MAIN2   PUSH  BC
        LD    A,21
        CALL  SCRLIN
        CALL  PAUSE       ;задержка для получения более
                          ; плавного смещения строки
        POP   BC
        DJNZ  MAIN2
        POP   HL
        JR    MAIN1
; Скроллинг, пока вся строка не исчезнет за левым краем экрана (0 = 265 раз)
MAIN3   LD    B,0
MAIN4   PUSH  BC
        LD    A,21
        CALL  SCRLIN
        CALL  PAUSE
        POP   BC
        DJNZ  MAIN4
        RET
PAUSE   LD    BC,1
        JP    7997
SCRLIN .........
PR_AT   DEFB  22,21,31,16,0
TEXT    DEFM  "Examine yourself how you know the assembler!"
        DEFB  0
```

ОКНА

Пожалуй, найдется не так уж много игровых программ, в которых в той или иной форме не использовались бы окна. Вы, наверное, уже хорошо знакомы с этим термином, но тем не менее поясним, что именно мы будем под ним подразумевать. Окно - это некоторая прямоугольная часть рабочего экрана, внутри которой можно производить различные преобразования независимо от внешней области. Например, можно изменить цвет окна, вывести в него какой-нибудь текст, переместить внутри него изображения и многое другое. Некоторые игры буквально напичканы окнами: в одном происходит сражение, в другом сообщается, сколько времени вам осталось «жить», в третьем... Да что перечислять, вы и сами это знаете не хуже нас. Поговорим лучше о том, как создаются окна и как получаются всевозможные эффекты, связанные с ними.

Формирование окна

Прежде всего нам нужно разобраться, каким образом задаются окна. Для этого необходимо знать, какие параметры требуются для их описания.

В простейшем случае (и если вам доводилось работать с Laser Basic'ом, то это должно быть хорошо известно) достаточно определить четыре переменные: ROW - позиция по вертикали верхнего края окна, COL - горизонтальная координата левого края окна, LEN - ширина окна и HGT - его высота. Чаще все четыре параметра задаются в знакоместах, но иногда (как, например, в редакторе Art Studio) - в пикселях, правда, реализовать такие окна несравненно сложнее.

Если вы хотите иметь возможность заключать окна в рамки, причем с индивидуальным рисунком для каждого из них, то потребуется еще один параметр - тип рамки. Он может задаваться символом, содержащим рисунок рамки, просто ее порядковым номером или каким-то другим способом, а ноль, например, будет говорить об отсутствии окаймления.

Большее количество переменных может понадобиться для определения окон со сложной внутренней структурой. Например, окно может иметь собственное название, заголовок, выделяемый цветом и постоянно присутствующий в окне. Такие типы окон часто можно встретить в меню игровых (MICRONAUT ONE) или прикладных (Art Studio) программ (рис. 5.4). Кроме того, окно может быть как бы приподнятым над общим фоном экрана и отбрасывать на него «тень».

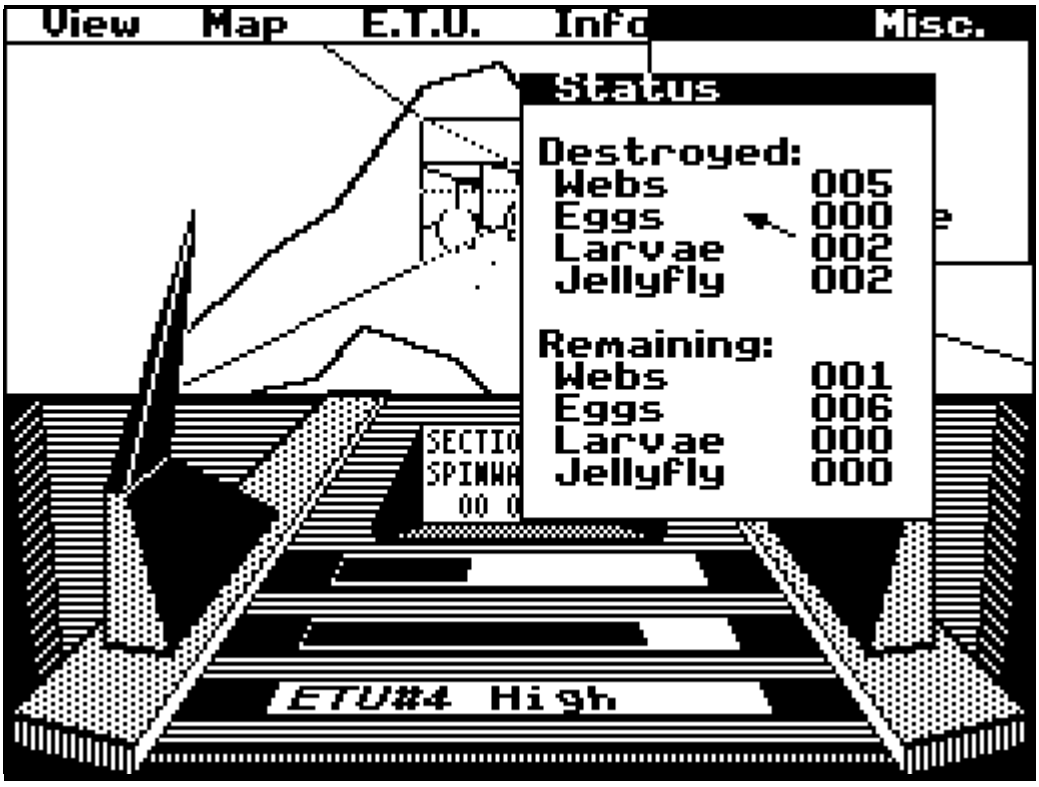


Рис. 5.4. Окна в игре MICRONAUT ONE

Как видите, можно придумать множество типов окон, но давайте сначала посмотрим, как получить наиболее простые из них и напишем подпрограммы, выполняющие наиболее распространенные преобразования в окнах. Первую из них, которая выполняет очистку заданной области экрана, назовем в соответствии с аналогичной процедурой Laser Basic'a

CLSV. Перед обращением к ней необходимо заполнить 4 переменные, под которые нужно зарезервировать память инструкцией ассемблера DEFB:

```
COL     DEFB  0
ROW     DEFB  0
LEN     DEFB  0
HGT     DEFB  0
```

Обратите внимание, что переменные должны располагаться именно в том порядке, в котором они указаны; менять их местами или разбрасывать по тексту программы недопустимо. Это условие введено для упрощения всех последующих процедур, работающих с окнами, о чем мы еще скажем при детальном разборе подпрограмм. Другое упрощение касается допустимых значений переменных: во-первых, окно не может иметь нулевые размеры ни по ширине, ни по высоте, а во-вторых, оно не должно выходить за пределы экрана.

Такие допущения сделаны не только для упрощения подпрограмм и сокращения их размеров. Это позволяет также и несколько увеличить их быстродействие, так как все необходимые проверки возлагаются на программиста. Кстати, в большинстве игровых программ операции с окнами (да и многие другие) производятся как раз безо всяких проверок корректности задаваемых параметров. Ведь подобные действия нужны, в основном, лишь на стадии отладки программы, а когда работа закончена, они становятся «мертвым грузом», только замедляющим выполнение процедур.

Итак, приводим подпрограмму очистки окна экрана:

```
CLSV    LD    BC,(LEN)     ;чтение сразу двух переменных:
                           ; C = LEN, B = HGT
        LD    A,(ROW)
CLSV1   PUSH  AF
        PUSH  BC
        CALL  3742         ;адрес начала строки экрана
        LD    A,(COL)      ;прибавляем смещение
        ADD   A,L          ; COL по горизонтали
        LD    L,A
        LD    B,8          ;в каждой строке 8 рядов пикселей
CLSV2   PUSH  HL
        LD    E,C          ;счетчик циклов в E, равный ширине окна
        XOR   A            ;в аккумуляторе 0
CLSV3   LD    (HL),A       ;обнуляем очередной байт видеобуфера
        INC   HL           ;переходим к следующему
        DEC   E            ;пока не дойдем до правого края окна
        JR    NZ,CLSV3
        POP   HL
        INC   H            ;переходим к следующему ряду пикселей
        DJNZ  CLSV2
        POP   BC
        POP   AF
        INC   A            ;переходим к следующей строке экрана
        DJNZ  CLSV1        ;повторяем, пока не дойдем
                           ; до нижнего края окна
        RET
```

Эта подпрограмма только очищает окно, но никак не влияет на его цвет. Для изменения атрибутов нужна другая процедура, которую назовем SETV. Здесь нам потребуется дополнительная переменная для хранения байта атрибутов, которую нужно определить в программе строкой

```
ATTR    DEFB  0
```

и перед обращением к подпрограмме SETV занести в нее необходимое значение.

Прежде чем привести текст подпрограммы, скажем несколько слов о способе расчета требуемого адреса в области атрибутов экрана. Эта задача намного проще определения адреса в области данных, так как атрибуты имеют линейную организацию. Поэтому достаточно номер строки экрана умножить на 32, то есть на длину полной строки экрана, добавить величину смещения от левого края (горизонтальную позицию интересующего знакоместа) и полученную величину сложить с адресом начала области атрибутов. Наиболее сложным моментом расчетов может показаться операция умножения и это было бы действительно так, не будь один из сомножителей числом, равным степени 2, что позволяет свести задачу к простому сложению.

Вот текст подпрограммы, выполняющей установку атрибутов в окне экрана:

```
SETV   LD    DE,#5800    ;адрес начала области атрибутов экрана
       LD    BC,(LEN)    ;C = LEN, B = HGT
       LD    A,(ROW)
       LD    L,A         ;расчет адреса левого верхнего угла окна
       LD    H,0         ; в области атрибутов экрана
       ADD   HL,HL       ;умножаем на 32 (2 в 5-ой степени)
       ADD   HL,HL
       ADD   HL,HL
       ADD   HL,HL
       ADD   HL,HL
       ADD   HL,DE       ;полученное смещение складываем
                         ; с началом области атрибутов
       LD    A,(COL)     ;добавляем горизонтальное смещение окна
       ADD   A,L
       LD    L,A
       LD    A,(ATTR)    ;в аккумуляторе байт атрибутов
SETV1  PUSH  BC
       PUSH  HL
SETV2  LD    (HL),A      ;помещаем в видеобуфер
       INC   HL
       DEC   C           ;до правого края окна
       JR    NZ,SETV2
       POP   HL
       POP   BC
       LD    DE,32       ;переходим к следующей строке
       ADD   HL,DE       ; (длина строки 32 знакоместа)
       DJNZ  SETV1       ;повторяем, пока не дойдем до нижнего
                         ; края окна
       RET
```

Еще одним довольно распространенным видом преобразований, выполняемых с окнами, является инвертирование изображения. Подпрограмма, реализующая это действие очень похожа на процедуру очистки окна, только в этом случае байт данных из видеобуфера инвертируется в аккумуляторе и затем помещается обратно на свое место. Как вы знаете, инверсия - это изменение состояния битов на противоположное: если бит установлен, он сбрасывается и наоборот. Такую операцию можно выполнить, например, с помощью команды

```
       XOR   255
```

однако микропроцессор имеет для этих целей специальную инструкцию CPL, которая и выполняется быстрее и занимает в памяти всего один байт. Ею мы и воспользуемся в подпрограмме INVV:

```
INVV   LD    BC,(LEN)
       LD    A,(ROW)
INVV1  PUSH  AF
       PUSH  BC
       CALL  3742
```

```
        LD    A,(COL)
        ADD   A,L
        LD    L,A
        LD    B,8
INVV2   PUSH  HL
        LD    E,C
INVV3   LD    A,(HL)       ;читаем байт из видеобуфера
        CPL                ;инвертируем байт в аккумуляторе
        LD    (HL),A       ;возвращаем обратно в видеобуфер
        INC   HL
        DEC   E
        JR    NZ,INVV3
        POP   HL
        INC   H
        DJNZ  INVV2
        POP   BC
        POP   AF
        INC   A
        DJNZ  INVV1
        RET
```

Поскольку здесь говорится об окнах, задаваемых с точностью до знакоместа, есть возможность несколько ускорить процедуру инвертирования изображения, что может оказаться существенным при работе с большой площадью экрана. Ведь вместо того, чтобы инвертировать каждый байт данных, можно просто поменять местами цвета INK и PAPER в области атрибутов и вместо восьми байт для каждого знакоместа обрабатывать только один. Для успешного решения этой задачи еще раз напомним значения битов в байте атрибутов: биты 0, 1 и 2 отвечают за цвет «чернил» INK, биты 3, 4 и 5 определяют цвет «бумаги» PAPER, 6-й бит задает уровень яркости BRIGHT, а старший 7-й бит включает или выключает мерцание FLASH.

Вот эта подпрограмма:

```
INVA    LD    DE,#5800     ;начало как в процедуре SETV
        LD    BC,(LEN)
        LD    A,(ROW)
        LD    L,A
        LD    H,0
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,DE
        LD    A,(COL)
        ADD   A,L
        LD    L,A
INVA1   PUSH  BC
        PUSH  HL
INVA2   LD    A,%11000000  ;маскируем 2 старших бита атрибутов,
                           ; которые не будут изменяться
        AND   (HL)         ;выделяем их из суммарных атрибутов
        LD    B,A          ;запоминаем их в регистре B
        LD    A,%00111000  ;маскируем биты для цвета PAPER
        AND   (HL)         ;выделяем их
        RRCA               ;сдвигаем на место битов цвета INK
        RRCA
        RRCA
        OR    B            ;объединяем с выделенными ранее битами
        LD    B,A          ;снова запоминаем
        LD    A,%00000111  ;маскируем биты для цвета INK
        AND   (HL)         ;выделяем их
```

```
        RLCA            ;сдвигаем на место битов цвета PAPER
        RLCA
        RLCA
        OR    B         ;объединяем все атрибуты
        LD    (HL),A    ;возвращаем их в видеобуфер
        INC   HL
        DEC   C
        JR    NZ,INVA2
        POP   HL
        POP   BC
        LD    DE,32
        ADD   HL,DE
        DJNZ  INVA1
        RET
```

Довольно часто в игровых программах применяется зеркальное отображение окон. Подобная процедура имеется и в Laser Basic'e. Ее выполняет оператор .MIRV, поэтому и нашу подпрограмму мы назовем так же. Перед обращением к ней, как и во всех предшествующих процедурах необходимо определить переменные ROW, COL, HGT и LEN с теми же ограничениями, о которых мы уже говорили.

```
MIRV    LD    BC,(LEN)
        LD    A,(ROW)
MIRV1   PUSH  AF
        PUSH  BC
        CALL  3742        ;в HL - адрес экрана
        LD    A,(COL)
        OR    L
        LD    L,A         ;начальный адрес левого края окна
; В DE получаем соответствующий адрес противоположного края окна
        LD    D,H
        LD    A,C
        ADD   A,L
        DEC   A
        LD    E,A
        LD    B,8         ;8 рядов пикселей
MIRV2   PUSH  DE
        PUSH  HL
        PUSH  BC
        SRL   C           ;делим ширину окна на 2
        JR    NC,MIRV3    ;продолжаем, если разделилось без остатка
        INC   C           ;иначе увеличиваем счетчик на 1
MIRV3   LD    A,(HL)      ;получаем байт с левой стороны
        CALL  MIRV0       ;зеркально отображаем его
        PUSH  BC          ;запоминаем его
        LD    A,(DE)      ;берем байт с правой стороны
        CALL  MIRV0       ;отображаем
        POP   AF          ;восстанавливаем предыдущий байт
                          ; в аккумуляторе
        LD    (HL),B      ;записываем «правый» байт
                          ; на левую сторону окна
        LD    (DE),A      ; и наоборот
        INC   HL          ;приближаемся с двух сторон
        DEC   DE          ; к середине окна
        DEC   C
        JR    NZ,MIRV3    ;повторяем, если еще не дошли до середины
        POP   BC
        POP   HL
        POP   DE
        INC   H           ;переходим к следующему ряду пикселей
        INC   D
        DJNZ  MIRV2
        POP   BC
        POP   AF
        INC   A           ;переходим к следующей строке экрана
```

```
        DJNZ  MIRV1
        RET
; Подпрограмма зеркального отображения байта в аккумуляторе
MIRV0   RLA
        RR    B           ;отображенный байт получится в B
        RLA
        RR    B
        RLA
        RR    B
        RLA
        RR    B
        RLA
        RR    B
        RLA
        RR    B
        RLA
        RR    B
        RLA
        RR    B
        RET
```

В этой процедуре применены уже достаточно сложные средства, о которых стоит поговорить особо. Ядром ее является внутренняя подпрограмма MIRV0, которая переворачивает байт: 7-й бит переходит в 0-й, 6-й - в 1-й и т. д. Исходный байт на входе в нее помещается в аккумулятор, а «перевернутый» получаем на выходе в регистре B. Поясним, как это происходит. После выполнения команды RLA биты аккумулятора сдвигаются влево и вытесняемый бит переходит во флаг переноса, который используется в качестве пересылочного буфера. При выполнении же команды RR B бит из флага переноса попадает в младший бит регистра B и на следующих этапах постепенно перемещается к левому краю, то есть в сторону старшего бита. После выполнения восьми пар команд сдвигов все биты из аккумулятора перейдут в регистр B, но окажутся записанными в обратном, «зеркальном» порядке. Добавим, что вместо команд

```
        RLA
        RR    B
```

с тем же результатом можно использовать команды

```
        RRA
        RL    B
```

На первый взгляд кажется, что подпрограмма MIRV0 написана не слишком экономично. Раньше мы призывали вас использовать все доступные методы оптимизации программы, а теперь вдруг размахнулись вместо того, чтобы заключить одни и те же повторяющиеся команды в цикл. Однако, мы должны заметить, что существует оптимизация не только размеров программы; иногда бывает гораздо важнее оптимизировать ее по времени исполнения. И в данном случае важность достижения наивысшего быстродействия процедуры зеркального отображения окна значительно «перевешивает» стремление к сокращению ее объема. Попробуйте переписать подпрограмму MIRV0, использовав один из возможных способов организации циклов, и результат будет заметен даже при отображении сравнительно небольших окон.

Желанием ускорить работу программы обусловлен и выбор команды сдвига аккумулятора. Ведь с тем же успехом можно было написать любую другую инструкцию, например, RL A или SLA A, но если вы загляните в Приложение I, то заметите, что команда RLA выполняется в два раза быстрее других команд сдвига, всего за 4 такта (столько же времени требует и команда RLCA, которую также можно использовать в данной подпрограмме).

Другим интересным моментом подпрограммы является сохранение в стеке с последующим восстановлением отображенного байта внутри цикла MIRV3. Как мы уже сказали, требуемый байт получается в регистре B и применение команды PUSH BC поэтому должно быть

понятно. Но затем почему-то использована инструкция POP AF. Это не ошибка, так и должно быть. Дело в том, что после второго вызова подпрограммы MIRV0 регистр B оказывается занят значением другого «перевернутого» байта, пары HL и DE также содержат нужную информацию. Свободными остаются только регистры C и A, но C связан в пару с занятым B, к тому же это младший регистр, а нам нужно восстановить из стека значение старшего. По счастью, аккумулятор в регистровой паре AF как раз занимает «старшее» место, а состояние флагов в данном случае нас не интересует. Именно это и делает возможным применение единственной инструкции POP AF вместо ряда пересылок между регистрами. Как видите, не всегда обязательно восстанавливать из стека ту же регистровую пару, которая была до этого сохранена командой PUSH.

Зеркальное отображение окон не будет полноценным, если «переворачивать» только пиксели, нужно также отображать и атрибуты. Процедура, выполняющая такую операцию, в общем должна быть похожа на только что описанную, она будет даже несколько проще благодаря тому, что сами байты в этом случае отображать уже не требуется. Если вам понятен принцип работы предыдущей подпрограммы, то с этой вы разберетесь без труда, посему приводим ее исходный текст без лишних предисловий и комментариев:

```
MARV    LD    BC,(LEN)
        LD    A,(ROW)
        LD    L,A
        LD    H,0
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        LD    DE,#5800
        ADD   HL,DE
        LD    A,(COL)
        ADD   A,L
        LD    L,A
        LD    D,H
        LD    A,C
        ADD   A,L
        DEC   A
        LD    E,A
MARV1   PUSH  BC
        PUSH  DE
        PUSH  HL
        SRL   C
        JR    NC,MARV2
        INC   C
MARV2   LD    A,(HL)
        EX    AF,AF'
        LD    A,(DE)
        LD    (HL),A
        EX    AF,AF'
        LD    (DE),A
        INC   HL
        DEC   DE
        DEC   C
        JR    NZ,MARV2
        POP   DE
        POP   HL
        LD    BC,32
        ADD   HL,BC
        EX    DE,HL
        ADD   HL,BC
        POP   BC
```

```
        DJNZ  MARV1
        RET
```

Динамическое окно

Продемонстрируем использование подпрограмм CLSV и SETV на примере довольно часто используемого способа вывода окна: оно появляется не внезапно, а как бы выплывает на поверхность экрана или приближается издалека. При этом изменяются не только размеры окна, но и его цвет.

Универсальная программа, создающая таким способом окно любого размера и в произвольном месте экрана, потребует достаточно сложных вычислений с привлечением не только операций сложения и умножения, но и деления. Поэтому, чтобы упростить задачу, напишем программу для конкретного окна. А чтобы вы могли лучше понять, как она работает, сначала сделаем ее в Laser Basic'e и только потом перепишем на ассемблере:

```
10 INK 5: PAPER 0: BORDER 0: CLS
20 FOR n=7 TO 1 STEP -1
30 LET m=7-n
40 .ROW=5+n:.COL=2+2*n
50 .HGT=2+2*m:.LEN=1+4*m
60 LET attr=8*n+64: POKE 23693,attr
70 .CLSV:.SETV
80 NEXT n
```

После этого можно обвести окно рамкой, а внутри что-нибудь написать, но это мы уже сделаем в ассемблерной программе.

Поскольку рамки вокруг окон рисуются довольно часто, прежде чем привести текст программы, напишем универсальную процедуру, выводящую в выбранном месте экрана прямоугольник произвольного размера. Этой процедурой мы еще воспользуемся впоследствии, поэтому ее также желательно иметь в отдельном библиотечном файле. Перед обращением к ней в регистрах B и C нужно задать координаты верхнего левого угла прямоугольника соответственно по вертикали и горизонтали, а в H и L - его высоту и ширину. Поскольку позже мы предложим более быстродействующую процедуру, то чтобы не возникло путаницы с именами, назовем ее BOX_0, а в имени другой процедуры изменим индекс.

```
BOX_0   PUSH  HL
        CALL  8933        ;PLOT - верхний левый угол
        POP   BC
        PUSH  BC
        LD    DE,#101     ;верхняя линия
        LD    B,0
        CALL  9402        ;DRAW
        POP   BC
        PUSH  BC
        LD    D,-1        ;правая линия
        LD    C,0
        CALL  9402
        POP   BC
        PUSH  BC
        LD    E,-1        ;нижняя линия
        LD    B,0
        CALL  9402
```

```
        POP   BC
        LD    DE,#101      ;левая линия
        LD    C,0
        CALL  9402
        LD    HL,10072
        EXX
        RET
```

Если в приведенной процедуре вам что-нибудь не совсем ясно, вернитесь к четвертой главе и еще раз просмотрите раздел «Рисование графических примитивов».

Используя подготовленные процедуры, можно написать программу «всплывающего» окна:

```
        ORG   60000
        ENT   $
        CALL  SETSCR       ;подготовка экрана
DYNW    LD    B,7
DYNW1   LD    A,7          ;вычисление промежуточной переменной
                           ; для расчета размеров окна
        SUB   B
        LD    C,A
        LD    A,B          ;расчет переменной ROW
        ADD   A,5
        LD    (ROW),A
        LD    A,B          ;расчет переменной COL
        ADD   A,A
        ADD   A,2
        LD    (COL),A
        LD    A,C          ;расчет переменной HGT
        ADD   A,A
        ADD   A,2
        LD    (HGT),A
        LD    A,C          ;расчет переменной LEN
        ADD   A,A
        ADD   A,A
        INC   A
        LD    (LEN),A
        LD    A,B          ;расчет байта атрибутов
        RLCA
        RLCA
        RLCA
        OR    %01000000    ;64 - повышенная яркость
        LD    (ATTR),A
        PUSH  BC
        LD    BC,3
        CALL  7997         ;небольшая задержка перед выводом окна
        CALL  CLSV         ;очистка окна
        CALL  SETV         ;установка атрибутов окна
        POP   BC
        DJNZ  DYNW1
        LD    DE,D_ATTR    ;установка атрибутов линий рамки
        LD    BC,6
        CALL  8252
        LD    BC,#7625     ;B = 118, C = 37
        LD    HL,#5EBE     ;H = 94, L = 190
        CALL  BOX_0        ;рамка вокруг окна
        LD    DE,TEXT
        LD    BC,END-TEXT
        JP    8252         ;печать текста в окне
SETSCR  .........
BOX_0   .........
CLSV    .........
SETV    .........
COL     DEFB  0
```

```
ROW     DEFB  0
LEN     DEFB  0
HGT     DEFB  0
ATTR    DEFB  0
D_ATTR DEFB  16,0,17,1,19,1
TEXT    DEFB  22,9,12,16,6,17,1,19,1
        DEFM  "Program"
        DEFB  22,11,8,16,4
        DEFM  "DYNAMIC··WINDOW"
        DEFB  22,15,8,16,5
        DEFM  "Saint-Petersburg"
        DEFB  22,17,14,16,3
        DEFM  "1994"
END
```

«Размножающиеся» окна

Рассмотрим программу, которая последовательно выводит на экран цепочку разноцветных окон различных размеров и которую при желании можно использовать как основу для создания оригинальной заставки. Самое последнее окно, появляющееся на экране, постепенно закрашивается черным цветом, оставляя только желтый контур, а затем в нем выводится заданный текст в виде «бегущей строки» (рис. 5.5).

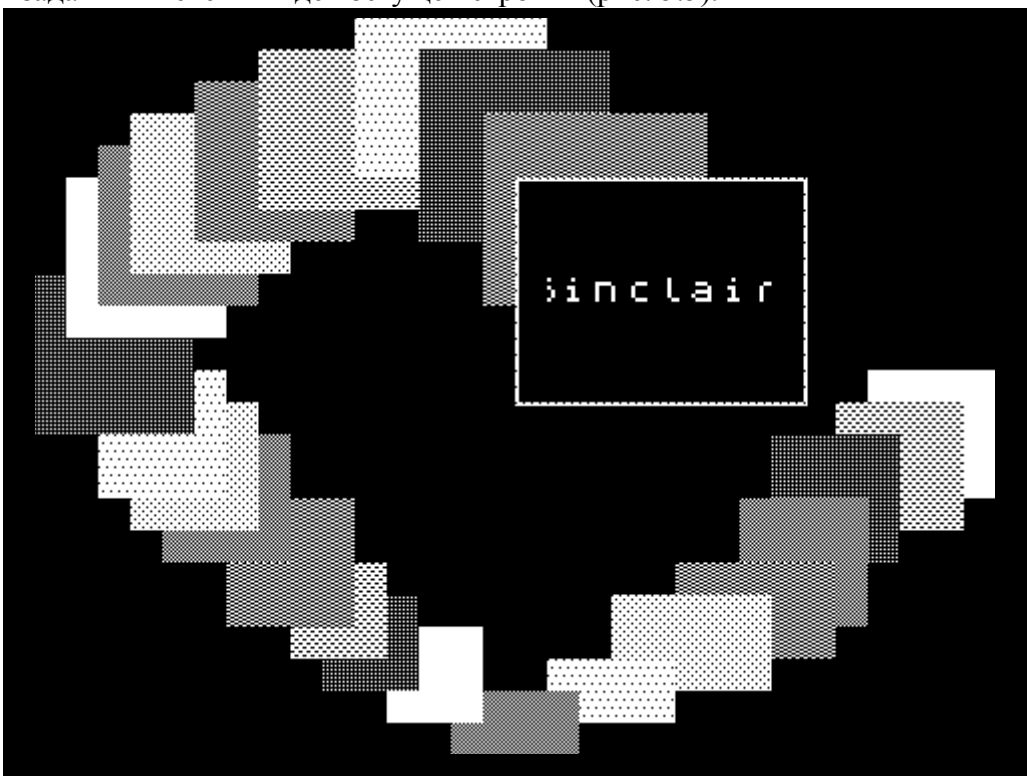


Рис. 5.5. Цепочка окон

Программа состоит из двух основных частей. На первом этапе из блока данных считываются параметры окон и выполняются уже известные процедуры CLSV и SETV. В конце блока данных установлен байт со значением -1 (255), при считывании которого программа переходит ко второму этапу - формированию «бегущей строки». Такое решение позволяет легко изменять не только размеры, цвет и местоположение окон, но и добавлять другие или убирать лишние, что особенно важно при отладке программы. Текст «бегущей строки» дан в формате ASCIIZ, и это также дает возможность как угодно изменять его без коррекций в самой программе.

Несмотря на довольно большой размер программы, в ней не встретится никаких неизвестных команд, поэтому приводим ее текст лишь с краткими комментариями:

```
        ORG   60000
        ENT   $
        CALL  SETSCR

; Вывод последовательности окон
        LD    HL,COORD    ;адрес блока данных параметров окон
PW1     LD    A,(HL)      ;последовательное считывание параметров
        CP    -1          ;проверка достижения конца блока данных
        JR    Z,PW2       ;если да, переходим ко второму этапу
        INC   HL
        LD    (COL),A
        LD    A,(HL)
        INC   HL
        LD    (ROW),A
        LD    A,(HL)
        INC   HL
        LD    (LEN),A
        LD    A,(HL)
        INC   HL
        LD    (HGT),A
        LD    A,(HL)      ;последний параметр - цвет окна
        INC   HL
        RLCA              ;сдвигаем на место атрибута PAPER
        RLCA
        RLCA
        OR    7           ;добавляем цвет INK 7
        LD    (ATTR),A
        PUSH  HL
        PUSH  BC
        XOR   A
        OUT   (254),A     ;получаем «щелчок»
        LD    BC,5
        CALL  7997        ;PAUSE 5
        LD    A,16
        OUT   (254),A
        CALL  CLSV        ;вывод окна
        CALL  SETV
        POP   BC
        POP   HL
        JR    PW1         ;переход к следующему окну
PW2     LD    A,6
        LD    (23695),A   ;устанавливаем временные атрибуты
        LD    BC,#8780    ;B = 135, C = 128
        LD    HL,#3848    ;H = 56, L = 72
        CALL  BOX_0       ;прямоугольник вокруг последнего окна
; «Бегущая строка» в последнем окне
        LD    HL,TEXT     ;адрес текста «бегущей строки»
PW3     LD    A,22        ;AT 8,24
        RST   16
        LD    A,8
        RST   16
        LD    A,24
        RST   16
        LD    A,16        ;INK 0
        RST   16
        XOR   A
        RST   16
        LD    A,(HL)      ;считываем очередной символ
        AND   A           ;дошли до конца?
        RET   Z           ;если да, то завершаем программу
```

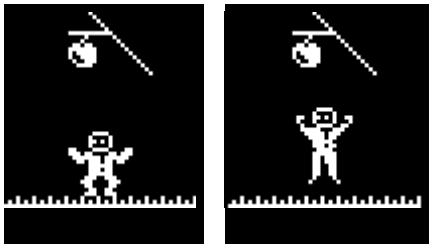
```
        RST   16          ;выводим считанный символ на экран
        INC   HL
        LD    B,8         ;сдвигаем строку на 8 пикселей влево
PW4     PUSH  BC
        PUSH  HL
        CALL  SCROL       ;скроллинг строки на 1 пиксель влево
        LD    BC,1
        CALL  7997        ;PAUSE 1
        POP   HL
        POP   BC
        DJNZ  PW4
        JR    PW3         ;переходим к выводу следующего символа
SCROL   LD    HL,18448+8  ;заранее рассчитанный адрес экрана
                          ; конца «бегущей строки»
        LD    C,8
SCROL1  LD    B,8
        AND   A
        PUSH  HL
SCROL2  RL    (HL)
        DEC   HL
        DJNZ  SCROL2
        POP   HL
        INC   H
        DEC   C
        JR    NZ,SCROL1
        RET
SETSCR  .........
SETV    .........
CLSV    .........
BOX_0   .........
COL     DEFB  0
ROW     DEFB  0
LEN     DEFB  0
HGT     DEFB  0
ATTR    DEFB  0
; Данные для всех окон. Параметры записаны в таком порядке:
; COL, ROW, LEN, HGT и последнее число - код цвета окна (PAPER)
COORD   DEFB  27,11,4,4,7
        DEFB  26,12,4,4,4
        DEFB  24,13,4,4,1
        DEFB  23,15,4,4,2
        DEFB  21,17,5,3,3
        DEFB  19,18,5,3,5
        DEFB  17,20,4,2,6
        DEFB  14,21,4,2,2
        DEFB  12,19,3,3,7
        DEFB  10,18,3,3,1
        DEFB  9,17,3,3,4
        DEFB  7,15,4,4,3
        DEFB  5,13,4,4,2
        DEFB  4,12,4,4,5
        DEFB  3,11,4,4,6
        DEFB  1,8,5,5,1
        DEFB  2,5,5,5,7
        DEFB  3,4,5,5,2
        DEFB  4,3,5,5,5
        DEFB  6,2,5,5,3
        DEFB  8,1,8,5,4
        DEFB  11,0,6,5,6
        DEFB  13,1,6,6,1
        DEFB  15,3,7,6,3
        DEFB  16,5,9,7,6
; «Закрашивание» последнего окна
        DEFB  20,8,1,1,0
        DEFB  19,7,3,3,0
```

```
        DEFB  18,6,5,5,0
        DEFB  17,6,7,5,0
        DEFB  16,5,9,7,0
        DEFB  255
;-------------------
TEXT    DEFM "Sinclair Research Ltd. 1982"
        DEFM  "·····Program··W·I·N·D·O·W"
        DEFM  "··*·Saint-Petersburg··1994··*"
        DEFM  "·········"
        DEFB  0
```

СПРАЙТЫ (ПРОГРАММА «ПРЫГАЮЩИЙ ЧЕЛОВЕЧЕК»)

Прояснив в какой-то степени вопрос о перемещении по экрану символов неплохо теперь заняться спрайтами и решить некоторые общие проблемы. Среди них: создание специальных блоков данных, удобных для кодирования и последующего вывода на экран спрайтов, разработка эффективной подпрограммы вывода этих спрайтов и, наконец, формирование самого подвижного изображения.

Рассмотрим один из вариантов решения этих вопросов (так как позже будут изложены и другие). В начале коротко о том, что мы увидим на экране. После запуска программы появится простенький пейзаж, состоящий из зеленой травы и ветки дерева, на которой висит спелое яблоко. От земли отталкивается человечек и в прыжке пытается достать это яблоко. Для получения более или менее правдоподобного эффекта движения оказалось достаточным выводить человечка всего в двух фазах (рис. 5.6 *а*, *б*). Разберем теперь программу, которая все это делает.



а б

Рис. 5.6. Фазы движения человечка

Прежде всего закодируем отдельные элементы двух изображений человечка, воспользовавшись, как и в предыдущей главе, определяемыми пользователем символами UDG. В результате кодирования получится блок данных, первому байту которого присвоим метку UDG:

```
UDG     DEFB  0,0,3,4,102,68,35,62          ;A (144)
        DEFB  0,0,192,32,166,34,204,56      ;B (145)
        DEFB  31,15,7,7,3,7,6,6             ;C (146)
        DEFB  112,96,192,96,192,224,224,224 ;D (147)
        DEFB  6,6,6,4,8,0,0,0               ;E (148)
        DEFB  96,96,96,32,16,0,0,0          ;F (149)
        DEFB  3,4,6,4,195,236,127,55        ;G (150)
        DEFB  192,32,160,32,195,55,126,108  ;H (151)
        DEFB  7,7,7,14,28,12,6,14           ;I (152)
        DEFB  224,96,224,112,56,48,96,112   ;J (153)
        DEFB  8,16,108,254,190,158,78,60    ;Яблоко (154)
```

```
       DEFB  17,85,255,0,0,0,0,0           ;Трава (155)
```

Справа от каждой строки указаны коды, которые понадобятся нам для составления трех других блоков данных. Первый из них с меткой PEJZ мы используем для создания примитивного пейзажа, состоящего, как мы уже говорили, из зеленой травы и яблока (маленькую ветку проще получить программным путем).

```
PEJZ   DEFB  22,13,13,16,4
       DEFB  155,155,155,155,155,155
       DEFB  22,8,15,16,2,17,0
       DEFB  154
```

Можно заметить, что кроме кодов UDG в блок включены также и управляющие символы, устанавливающие позицию печати и изменяющие цвета спрайтов.

Второй и третий блоки - это как раз те два спрайта, которые должны попеременно появляться на вашем экране, создавая иллюзию движения:

```
SPR1   DEFB  6
       DEFB  0,0,7,144,0,1,7,145
       DEFB  1,0,7,146,1,1,7,147
       DEFB  2,0,7,148,2,1,7,149
SPR2   DEFB  6
       DEFB  0,0,0,32,0,1,0,32
       DEFB  1,0,7,150,1,1,7,151
       DEFB  2,0,7,152,2,1,7,153
```

Для вывода этих спрайтов на экран можно было бы, как и раньше, воспользоваться подпрограммой 8252, однако основная идея применения спрайтов заключается в том, чтобы их можно было легко перемещать по экрану. Но поскольку спрайт обычно состоит из нескольких знакомест, которые имеют строго фиксированное положение внутри него, то при использовании процедуры 8252 пришлось бы перекодировать координаты всех составляющих его фрагментов. Сами понимаете, что это не слишком удобно, особенно если спрайт состоит из десятка-другого знакомест. Это означает, что для вывода спрайтов на экран необходимо создать самостоятельную процедуру, которую мы назовем PUT.

Теперь следует сказать несколько слов о построении блоков данных SPR1 и SPR2, поскольку они тесно связаны с подпрограммой PUT и их формат, естественно, должен быть согласован с ее работой. Опишем их структуру. Первый байт соответствует количеству фрагментов, входящих в спрайт (в нашем примере как для первого, так и для второго спрайта - это число 6). Далее следуют параметры для каждого фрагмента, занимающие по 4 байта:

```
      1-й байт - относительная вертикальная координата данного
                 знакоместа в спрайте;
      2-й байт - относительная горизонтальная координата данного
                 знакоместа в спрайте;
      3-й байт - байт суммарных атрибутов знакоместа;
      4-й байт - код символа соответствующего фрагмента спрайта.
```

Поскольку подпрограмма PUT нам потребуется не только в данном примере, сохраните ее в виде отдельного библиотечного файла. Она имеет следующий вид:

```
PUT    LD    E,(HL)       ;считываем количество фрагментов спрайта
PUT1   INC   HL
       LD    A,22
       RST   16
       LD    A,B          ;регистр B содержит координату ROW
       ADD   A,(HL)       ;прибавляем к ней относительную
                          ; координату внутри спрайта
       RST   16
       INC   HL
```

```
        LD    A,C           ;в C - горизонтальная координата COL
        ADD   A,(HL)        ;складываем ее с относительной
                            ; координатой, взятой из блока данных
        RST   16
        INC   HL
        LD    A,(HL)        ;считываем байт атрибутов знакоместа
        LD    (23695),A
        INC   HL
        LD    A,(HL)        ;берем код выводимого символа
        RST   16
        DEC   E
        JR    NZ,PUT1       ;переходим к выводу следующего
                            ; фрагмента
        RET
```

Перед вызовом этой процедуры необходимо в регистровой паре HL указать начальный адрес блока данных, соответствующего выводимому спрайту, а в регистрах B и C задать координаты экрана по вертикали и горизонтали.

Наконец соберем все части программы в единое целое, добавив несколько уже знакомых процедур, и напишем небольшую часть, управляющую движением человечка:

```
        ORG   60000
        ENT   $
        CALL  SETSCR
        LD    HL,UDG
        LD    (23675),HL
        CALL  FON
CYCLE   LD    HL,SPR1       ;вывод спрайта 1
        LD    BC,#A0F       ;B = 10, C = 15
        CALL  PUT
        CALL  PAUSE         ;небольшая задержка
        LD    HL,SPR2       ;вывод спрайта 2
        LD    BC,#A0F
        CALL  PUT
        CALL  PAUSE         ;снова задержка
; Выход из цикла при нажатии клавиши Space
        LD    A,(23560)     ;системная переменная LAST_K, в которой
                            ; хранится код последней нажатой клавиши
        CP    " "
        JR    NZ,CYCLE      ;повтор
        RET
; Задержка (PAUSE 12)
PAUSE   LD    BC,12
        JP    7997
; Рисование пейзажа
FON     LD    DE,PEJZ
        LD    BC,19
        CALL  8252
        EXX
        PUSH  HL
        LD    A,6
        LD    (23695),A
        LD    BC,#7078      ;B = 112, C = 120
        CALL  8933
        LD    DE,#101       ;D = 1, E = 1
        LD    BC,10
        CALL  9402
        LD    BC,#757D      ;B = 117, C = 125
        CALL  8933
        LD    E,#FF01       ;D = -1, E = 1
        LD    BC,#F0F       ;B = 15, C = 15
        CALL  9402
```

```
        POP    HL
        EXX
        RET
SETSCR  .........
PUT     .........
PEJZ    .........
SPR1    .........
SPR2    .........
UDG     .........
```

Эффект движения здесь создается с помощью простого цикла, начинающегося с метки CYCLE, в теле которого по очереди выводятся то первый спрайт, изображающий стоящего человечка, то второй, показывающий человечка в прыжке. Скорость смены фаз движения регулируется подпрограммой PAUSE.

МУЛЬТИПЛИКАЦИОННАЯ ЗАСТАВКА

Разговор о подвижных изображениях хотелось бы завершить чем-то законченным, что можно прямо или с вашими дополнениями использовать в игровых программах. Но при этом текст будущей ассемблерной программы не должен быть уж слишком длинным. Учитывая все это, предлагаем один из вариантов динамической заставки, в которой создается настоящая мультипликационная картинка (рис. 5.7). По двум лестницам вверх и вниз передвигаются два маленьких человечка, поворачиваясь к вам то лицом, то спиной. Вся заставка окружена рамкой из шариков, переливающихся всеми цветами радуги. Когда смотришь на эту подвижную картинку, то не верится, что создана она довольно простыми средствами, уже рассмотренными нами в предыдущих разделах. Разберем теперь текст программы и подробно прокомментируем некоторые ее части, представляющие особый интерес.

Рис. 5.7. Мультипликационная заставка

Начнем, как и раньше, с составления блоков данных. Самое простое - это формирование блока, содержащего тексты, без которых не обходится ни одна заставка, будь она статическая или динамическая:

```
TEXT   DEFB  22,2,9,16,6
       DEFM  "Welcome to the"
       DEFB  22,5,7,16,7
       DEFM  "L·I·T·T·L·E··M·A·N"
       DEFB  22,8,9,16,4
       DEFM  "0. START GAME"
       DEFB  22,10,9
       DEFM  "1. KEYBOARD"
       DEFB  22,12,9
       DEFM  "2. KEMPSTON"
       DEFB  22,14,9
       DEFM  "3. INSTRUCTIONS"
       DEFB  22,16,9
       DEFM  "4. DEFINE KEYS"
       DEFB  22,19,8,16,3
       DEFM  "Press key 0 to 4"
```

Поскольку с выводом текстов мы уже неоднократно сталкивались и подробно их комментировали, то не будем лишний раз заострять на этом внимание и сразу перейдем к следующему блоку. Для кодирования отдельных элементов подвижных изображений вновь воспользуемся символами UDG, определяющими отдельные фрагменты будущих спрайтов.

```
UDG    DEFB  3,3,15,3,6,7,2,1              ; A (144)
       DEFB  224,224,248,224,176,240,32,19 ; B (145)
       DEFB  13,29,53,37,7,6,14,0          ; C (146)
       DEFB  208,216,200,208,240,112,48,56 ; D (147)
       DEFB  7,7,31,7,13,15,4,3            ; E (148)
       DEFB  192,192,240,192,96,224,64,128 ; F (149)
       DEFB  11,27,19,11,15,14,12,28       ; G (150)
       DEFB  176,184,172,164,224,96,112,0  ; H (151)
       DEFB  3,3,15,0,4,4,0,1              ; I (152)
       DEFB  224,224,248,0,16,16,0,192     ; J (153)
       DEFB  6,27,27,27,7,6,14,0           ; K (154)
       DEFB  176,120,248,240,240,112,48,56 ; L (155)
       DEFB  7,7,31,0,8,8,0,3              ; M (156)
       DEFB  192,192,240,0,32,32,0,128     ; N (157)
       DEFB  13,30,31,15,15,14,12,28       ; O (158)
       DEFB  96,216,216,216,224,96,112,0   ; P (159)
       DEFB  0,28,38,79,95,127,62,28       ; Q (160)
       DEFB  16,16,16,127,16,16,16,16      ; R (161)
       DEFB  8,8,8,254,8,8,8,8             ; S (162)
```

Следующий блок данных - это то, чем будет манипулировать программа, создавая на экране непрерывное движение маленьких симпатичных человечков:

```
SPR1   DEFB  4,0,0,7,144,0,1,7,145,1,0,4,146,1,1,4,147
SPR2   DEFB  4,0,0,7,148,0,1,7,149,1,0,4,150,1,1,4,151
SPR3   DEFB  4,0,0,7,152,0,1,7,153,1,0,4,154,1,1,4,155
SPR4   DEFB  4,0,0,7,156,0,1,7,157,1,0,4,158,1,1,4,159
SPR5   DEFB  2,0,0,6,161,0,1,6,162
```

Каждая из строк SPR1...SPR4 соответствует одной из фаз движения человечка, а SPR5 - фрагменту лестницы. Последний используется не только для рисования лестниц в статической картинке, но и для восстановления изображения позади бегущего человечка. Формат данных в этих блоках вам уже знаком. Он выбран точно таким же, как и в программе ПРЫГАЮЩИЙ ЧЕЛОВЕЧЕК, следовательно, для вывода всех спрайтов на экран можно воспользоваться подпрограммой PUT, которую мы уже описали. Кроме блоков данных нам понадобятся еще несколько переменных, необходимых для управления перемещением человечков по экрану:

```
ROW    DEFB  0          ;позиция по вертикали левого человечка
DIRECT DEFB  0          ;направление движения
POSIT  DEFB  0          ;фаза движения
```

Позже мы объясним смысл этих переменных более подробно.

После того, как мы разобрались с надписями и спрайтами, можно переходить к основной программе, которая, как и в большинстве игр, содержит две части: статическую и динамическую. Первая устанавливает атрибуты экрана и формирует на нем все неподвижные элементы заставки: текст, лестницы и рамки вокруг них (рис. 5.7). Во многих играх вместо загрузки дополнительных фонтов новый набор символов формируется программой. Например, можно предложить следующий способ получения шрифта Bold, буквы в котором имеют утолщенное начертание:

```
BOLD   LD    HL,15616    ;адрес стандартного набора символов
       LD    DE,END      ;метка конца программы; по этому
                         ; адресу расположится новый фонт
       LD    BC,#300     ;счетчики циклов: B = 3, C = 0 (256)
       PUSH  DE
BOLD1  LD    A,(HL)      ;байт из стандартного набора
       SRL   A           ; смещаем на 1 бит (пиксель) влево
       OR    (HL)        ; и объединяем с его прежним значением
       LD    (DE),A      ;помещаем в новый фонт
       INC   HL          ;переходим к следующему байту
       INC   DE
       DEC   C
       JR    NZ,BOLD1    ;в общей сложности повторяем цикл
       DJNZ  BOLD1       ; 768 раз
       POP   HL
       DEC   H           ;уменьшаем адрес нового шрифта на 256
       LD    (23606),HL  ; и заносим в системную
                         ; переменную CHARS
       RET
```

После получения нового шрифта можно сформировать на экране все неподвижные части заставки:

```
SCREEN LD    B,3
; Рисование окон
WIND   LD    C,3
       CALL  STAIRS
       LD    C,27
       CALL  STAIRS
       INC   B
       LD    A,B
       CP    19
       JR    C,WIND
; Рисование рамок вокруг лестниц
       LD    DE,ATRBOX
       LD    BC,4
       CALL  8252
       LD    BC,#1414    ;B = 20, C = 20
       CALL  8933
       CALL  BOX
       LD    BC,#14D4    ;B = 20, C = 212
       CALL  8933
       CALL  BOX
; Печать текста
       LD    DE,TEXT
       LD    BC,UDG-TEXT
       JP    8252
; Данные атрибутов рамок
ATRBOX DEFB  16,7,17,1
; Рисование фрагмента лестницы
STAIRS LD    A,17
       RST   16
```

```
       XOR   A
       RST   16
       LD    HL,SPR5
       JP    PUT
; Рисование рамок
BOX    EXX
       PUSH  HL
       LD    BC,#8700    ;B = 135, C = 0
       LD    DE,#101     ;D = 1, E = 1
       CALL  9402
       LD    BC,23       ;B = 0, C = 23
       CALL  9402
       LD    BC,#8700
       LD    D,-1
       CALL  9402
       LD    BC,23
       LD    E,-1
       CALL  9402
       POP   HL
       EXX
       RET
```

Подпрограмма SCREEN не содержит ничего такого, что бы нами еще не рассматривалось, поэтому отдельно мы ее комментировать не будем, а ограничимся уже приведенными пояснениями и перейдем к динамической части заставки.

В первую очередь напишем подпрограмму, печатающую в заданной позиции экрана элемент, из которого составлена рамка вокруг всего изображения:

```
; Вывод на экран шарика цвета E в позицию BC
PRINT  LD    A,22
       RST   16
       LD    A,B
       DEC   A
       RST   16
       LD    A,C
       DEC   A
       RST   16
       LD    A,16
       RST   16
       LD    A,E
       RST   16
       LD    A,160
       RST   16
       RET
```

Значительно сложнее заставить двигаться человечков, да чтобы они при этом еще переступали с ноги на ногу и поворачивались к вам то лицом, то спиной. Для этих целей введены переменные:

ROW - определяет вертикальную позицию левого человечка. Вертикальная координата правого легко вычисляется, так как они движутся в противофазе: когда один идет вверх, другой движется вниз и наоборот. Горизонтальные же координаты обоих человечков во время движения не изменяются.

DIRECT - задает направление движения также для левого человечка. Если она имеет значение 1, левый человечек идет вниз, а правый - вверх. При изменении направлений движения она меняет знак и становится равной минус 1 (255).

POSIT - сообщает программе, в какой из двух фаз выводить изображения человечков. При ее значении, равном 0, выводятся спрайты SPR2 и SPR4, а если она равна 1 - SPR1 и SPR3.

Иными словами, эта переменная отвечает за то, на какую ногу в данный момент должен наступить каждый человечек.

Стоит сказать несколько слов о том, какими программными средствами изменяются эти переменные. Наиболее просто получается переключение переменной POSIT. Для этого достаточно использовать команду XOR 1, инвертирующую значение младшего бита. Чтобы изменить знак в переменной DIRECT, можно поступить следующим образом: инвертировать все ее биты, например, командой CPL, а затем увеличить ее на 1. Можно именно так и поступить, но лучше воспользоваться специальной командой, как раз предназначенной для изменения знака в аккумуляторе. Эта команда записывается мнемоникой NEG (Negative - отрицательный). Что касается ROW, то она изменяется простым сложением с переменной DIRECT.

Теперь приводим тексты соответствующих подпрограмм:

```
; Перемещение человечков
MAN     LD    A,(ROW)
        LD    B,A
        LD    A,(DIRECT)
        LD    D,A
        CP    1           ;проверка направления движения
        JR    NZ,MAN1
        CALL  PRMAN1
        LD    E,17        ;нижняя граница координаты ROW
        JR    MAN2
MAN1    CALL  PRMAN2
        LD    E,3         ;верхняя граница координаты ROW
MAN2    LD    HL,POSIT    ;изменение переменной POSIT
        LD    A,(HL)
        XOR   1
        LD    (HL),A
        LD    HL,ROW
        LD    A,(HL)
        CP    E           ;проверка необходимости смены
        LD    A,D         ; направления движения
        JR    NZ,MAN3
        NEG               ;изменение знака в аккумуляторе
        LD    (DIRECT),A  ;изменение переменной DIRECT
MAN3    ADD   A,(HL)      ;изменение переменной ROW
                          ; (адресуемой парой HL)
        LD    (HL),A
        RET

; Левый человечек идет вниз, правый - вверх
PRMAN1 DEC    B           ;координата ROW
        LD    C,3         ;горизонтальная координата левого
                          ; человечка
        LD    HL,SPR5     ;восстановление фона
        CALL  PUT
        INC   B
        LD    A,(POSIT)
        AND   A
        LD    HL,SPR1
        JR    NZ,PRM11
        LD    HL,SPR2
PRM11   CALL  PUT
        LD    A,22        ;вычисление вертикальной координаты
        SUB   B           ; правого человечка
        LD    B,A
        LD    C,27        ;горизонтальная координата правого
```

```
                        ; человечка
        LD    HL,SPR5     ;восстановление фона
        CALL  PUT
        DEC   B
        DEC   B
        LD    A,(POSIT)
        AND   A
        LD    HL,SPR4
        JR    Z,PRM12
        LD    HL,SPR3
PRM12   JP    PUT

; Левый человечек идет вверх, правый - вниз
PRMAN2  INC   B
        INC   B
        LD    C,3
        LD    HL,SPR5
        CALL  PUT
        DEC   B
        DEC   B
        LD    A,(POSIT)
        AND   A
        LD    HL,SPR3
        JR    NZ,PRM21
        LD    HL,SPR4
PRM21   CALL  PUT
        LD    A,19
        SUB   B
        LD    B,A
        LD    C,27
        LD    HL,SPR5
        CALL  PUT
        INC   B
        LD    A,(POSIT)
        AND   A
        LD    HL,SPR2
        JR    Z,PRM22
        LD    HL,SPR1
PRM22   JP    PUT
```

Теперь объединим рассмотренные подпрограммы и напишем управляющую часть:

```
        ORG   60000
        ENT   $
        LD    HL,UDG
        LD    (23675),HL
        CALL  BOLD
        LD    A,15
        LD    (23693),A
        XOR   A
        CALL  8859
        CALL  3435
        LD    A,2
        CALL  5633
; Инициализация переменных
        LD    A,4
        LD    (ROW),A
        LD    A,1
        LD    (DIRECT),A
        LD    (POSIT),A
        CALL  SCREEN
        LD    A,17
        RST   16
        XOR   A
        RST   16
; Начало динамической части заставки
```

```
MAIN   LD    E,7          ;код цвета шариков, окаймляющих
                          ; рабочий экран
; Вывод рамки, состоящей из разноцветных шариков
MAIN1  LD    B,22         ;в регистрах BC - координаты экрана
MAIN2  LD    C,1
       CALL  PRINT
       LD    C,32
       CALL  PRINT
       DJNZ  MAIN2
       LD    C,31
MAIN3  LD    B,1
       CALL  PRINT
       LD    B,22
       CALL  PRINT
       DEC   C
       JR    NZ,MAIN3
       PUSH  BC
       PUSH  DE
       CALL  MAN          ;вывод очередной фазы движения
                          ; человечков
       POP   DE
       POP   BC
       PUSH  BC
       LD    B,9          ;счетчик цикла задержки перемещения
                          ; человечков
MAIN4  PUSH  BC
       LD    BC,1
       CALL  7997
       POP   BC
; Проверка нажатия клавиш от 0 до 4. Если нажаты - выход на EXIT
       LD    A,(23560)
       CP    "4"+1
       JR    NC,MAIN5     ;проверка >"4" (>="4"+1)
       CP    "0"
       JR    NC,EXIT
MAIN5  DJNZ  MAIN4
       POP   BC
       DEC   E
       JR    NZ,MAIN1
       JR    MAIN
EXIT   LD    (KEY),A      ;сохраняем код клавиши для других
                          ; частей многокадровой заставки,
                          ; принцип построения которой будет
                          ; рассмотрен в следующей главе
       POP   BC
       RET
; Подпрограммы
BOLD   .........
SCREEN .........
MAN    .........
PRMAN1 .........
PRMAN2 .........
PRINT  .........
PUT    .........
; Блоки данных
TEXT   .........
UDG    .........
SPR1...SPR5
; Переменные
ROW    DEFB  0
DIRECT DEFB  0
POSIT  DEFB  0
KEY    DEFB  0
; Адрес размещения нового фонта (обязательно в самом конце,
;  чтобы коды набора не перекрыли коды программы)
```

```
END
```

ГЛАВА ШЕСТАЯ,

в которой демонстрируются возможности ассемблера на примере создания многокадровых заставок

Когда в первой главе мы рассматривали основные части игровой программы, то в общих чертах показали, как строится многокадровая заставка и какие функции должны выполнять отдельные ее элементы. Теперь настало время всерьез поговорить о том, как реализовать эти элементы в виде законченных подпрограмм, а затем собрать их в единое целое для получения готовой заставки. Конечно, трудно дать ответы на все вопросы, с которыми вы можете столкнуться при решении данной задачи, поэтому мы решили ограничиться только теми проблемами, которые ранее не рассматривались совсем. Среди них - создание простейших звуковых эффектов, преобразование символов стандартного набора таким образом, чтобы надписи не стыдно было поместить на самое видное место экрана, скроллинги окон с графическими изображениями и текстами во всех направлениях, а также включение в программу элемента случайности. После того, как все это будет подробно рассмотрено, станет возможно, наконец, непосредственно заняться многокадровой заставкой.

СОЗДАНИЕ ПРОСТЕЙШИХ ЗВУКОВ

Вероятно, вы давно и с большим нетерпением ждете того момента, когда мы наконец соблаговолим поведать о том, как у компьютера «прорезать голос». До сих пор мы применяли некие акустические суррогаты, однако терпеть и дальше такое «безобразие» уже нет никакой возможности. Более детально и всесторонне вопросы создания мелодий и шумовых эффектов (в том числе и для ZX Spectrum 128) будут рассмотрены в десятой главе. Пока же мы предлагаем не очень сложный способ получения звуков, основанный на использовании подпрограммы ПЗУ, к которой в конечном итоге обращается интерпретатор при выполнении оператора BEEP. Эта подпрограмма располагается по адресу 949 и требует указания в регистровых парах DE и HL соответственно длительности и высоты звука. Методику расчета этих параметров для получения конкретных музыкальных звуков мы объясним позже, а сейчас нас интересуют простые звуковые эффекты, не имеющие конкретной высоты, поэтому данные значения можно подбирать просто на слух. Пожалуй, самая простая программа, которую можно сравнить с резонатором, настроенным на определенную частоту, выглядит следующим образом:

```
LD    DE,40
LD    HL,500
CALL  949
RET
```

Несмотря на некоторую трудность подбора параметров, все же попытаемся создать с помощью приведенной подпрограммы что-либо интересное, причем так, чтобы время звучания и высоту тона можно было свободно регулировать. Поскольку при этом возможны самые разные варианты, то получится множество звуковых эффектов, что, собственно говоря, нам и требуется.

Рассмотрим программу, основанную на приведенном выше примере, генерирующую звуки длительностью порядка 1-3 секунд. Она отличается тем, что высота тона изменяется в цикле, а продолжительность звучания можно регулировать не только изменением DE, но и занося различные значения в регистр B. Мы предполагаем использовать ее в дальнейшем, для чего присвоим ей собственное имя:

```
SND     LD    B,10        ;количество циклов
        LD    HL,300      ;начальная частота звучания
        LD    DE,8        ;длительность звука
SND1    PUSH  BC
        PUSH  DE
        PUSH  HL
        CALL  949
        POP   HL
        POP   DE
        POP   BC
        DEC   HL          ;или INC HL
        DJNZ  SND1
        RET
```

После запуска вы услышите звук, увеличивающийся по высоте, но если где-то в игре потребуется, чтобы с течением времени частота уменьшалась, достаточно команду DEC HL заменить на INC HL. Для извлечения более коротких звуков (0.1-0.3 сек.) достаточно задать в SND другие значения регистров:

```
SND     LD    B,90
        LD    DE,2
        LD    HL,150
SND1    .........
```

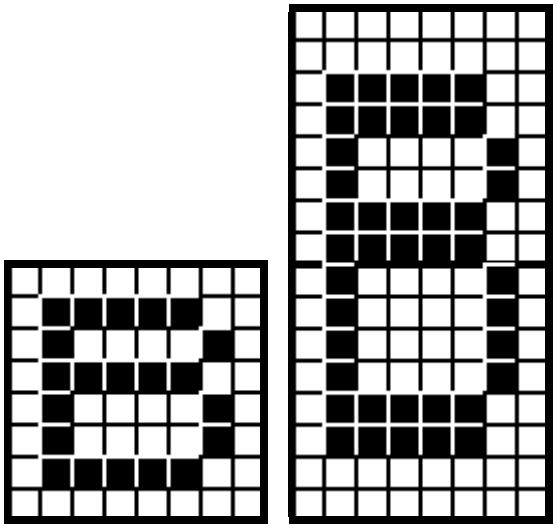
Располагая этими программками, попробуйте поэкспериментировать, встроив, например, два резонатора в один цикл, либо, организовав несколько (в том числе и вложенных) циклов с одинаковыми или разными резонаторами. Уверены, что это занятие приведет вас ко многим «открытиям» и доставит немало приятных минут.

МАНИПУЛЯЦИИ С БУКВАМИ

Иметь в своем распоряжении множество наборов символов различных начертаний, подобных тем, которые мы привели в четвертой главе, и использовать их для печати текстов заставки и других частей программы - это замечательно, но не всегда целесообразно. Любой фонт занимает, во-первых, какую-то часть памяти компьютера, а во-вторых - определенное место на ленте и требует дополнительного времени при загрузке игровой программы. Но ведь у вас под рукой всегда имеется стандартный набор символов, и возникает вопрос, а нельзя ли его как-то преобразовать программным путем, чтобы получить видоизмененные буквы и цифры. В предыдущей главе мы описали процедуру BOLD, применение которой приводит к утолщению символов, и этот прием довольно часто используется даже в фирменных играх. Рассмотрим теперь два способа преобразования символов, первый из которых увеличивает их высоту в два раза, а второй делает их в два раза шире.

Печать символов высотой в два знакоместа

Чтобы лучше понять, как работает ассемблерная программа, удваивающая высоту символов, приведем сначала два рисунка. На первом из них (рис. 6.1, *а*) показан исходный символ стандартного набора, а на другом (рис. 6.1, *б*) - то, что должно получиться после преобразования. Нетрудно заметить, что все байты (начиная с верхнего) левого символа, повторяются дважды на правом рисунке. Таким образом, заставив программу проделать эту операцию с каждым из восьми байтов, мы получим увеличение высоты символа ровно в два раза. Если вам очень захочется, то можно таким способом «вырастить» буквы в три, четыре и более раз, но над программой тогда придется немного поработать. Заметим, что символы высотой, скажем, в шесть знакомест выглядят на экране весьма эффектно.



а б

Рис. 6.1. Действие процедуры BIGSYM

Приведем теперь текст программы BIGSYM, которая удваивает высоту одного символа, причем его код должен быть предварительно помещен в ячейку памяти по адресу 23296.

```
        ORG   60000
        ENT   $
        LD    A,(23296)    ;загружаем код печатаемого символа
BIGSYM  LD    L,A
        LD    H,0          ;переписываем этот код в HL
        ADD   HL,HL        ;умножаем код на 8
        ADD   HL,HL
        ADD   HL,HL
        LD    DE,(23606)   ;в DE загружаем адрес начала
                           ; текущего фонта
        ADD   HL,DE
        EX    DE,HL
        LD    HL,(23684)   ;в HL помещаем адрес в видеобуфере,
                           ; по которому будет выводиться первый
                           ; байт измененного символа
        LD    B,4
        PUSH  HL           ;делаем две копии HL, чтобы
        PUSH  HL           ; использовать их во втором цикле
BIGS1   LD    A,(DE)       ;считываем байт из фонта
        LD    (HL),A       ;переписываем в видеобуфер
        INC   H
        LD    (HL),A       ;еще раз - ниже
        INC   H
        INC   DE           ;переходим к следующему байту
        DJNZ  BIGS1
        POP   HL           ;восстанавливаем HL
        LD    BC,32        ;вычисляем адрес первого байта
        ADD   HL,BC        ; второго знакоместа
```

```
        LD    B,4
BIGS2   LD    A,(DE)       ;аналогично циклу BIGS1
        LD    (HL),A
        INC   H
        LD    (HL),A
        INC   H
        INC   DE
        DJNZ  BIGS2
        POP   HL           ;восстанавливаем HL
        INC   HL           ;увеличиваем HL на 1 для подготовки
                           ; печати следующего символа
        LD    (23684),HL   ;записываем в системную переменную
                           ; адрес следующего знакоместа экрана
        RET
```

Обращаем ваше внимание на то, что процедура BIGSYM получилась не совсем универсальной. Сделано это с единственной целью упростить и сократить ее исходный текст. Применяя ее, нужно следить, чтобы символы при печати не выходили за пределы экрана ни по вертикали, ни по горизонтали. Кроме того, верхняя и нижняя половинки выводимых знаков не должны попадать в разные трети экрана, то есть не допускается позиционирование курсора на 7-ю и 15-ю строки экрана.

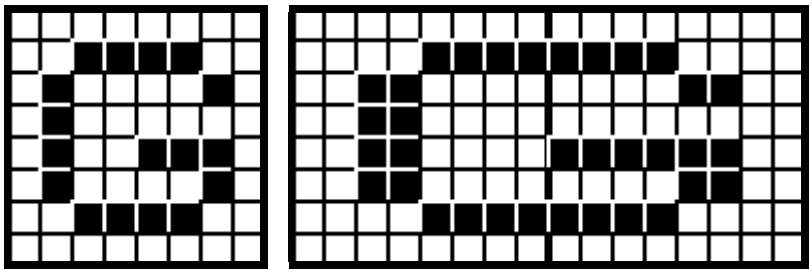
Чтобы посмотреть, как выглядит на экране целая строка удлиненных символов, введите небольшую программку на Бейсике. Само собой разумеется, что перед ее запуском ассемблерная программа должна быть оттранслирована.

```
10 PRINT AT 5,5;
20 LET a$="Starting program BIGSYM!"
30 FOR n=1 TO LEN a$
40 POKE 23296,CODE a$(n)
50 RANDOMIZE USR 60000
60 NEXT n
```

Насладившись созерцанием новых букв, возможно, вы захотите воспользоваться процедурой BIGSYM в собственной программе. В последнем разделе этой главы, где приводится программа и описание многокадровой заставки, можно посмотреть, как это лучше сделать. Пока же можем сказать следующее: перед вызовом процедуры из ассемблера необходимо установить позицию печати в нужное место экрана, воспользовавшись командой RST 16, а в аккумулятор занести код выводимого символа. В этом случае надобность в команде LD A,(23296), предваряющей в приведенном примере процедуру BIGSYM, отпадает.

Печать символов шириной в два знакоместа

По аналогии с предыдущим параграфом, начнем с двух небольших рисунков, из которых можно легко понять, как осуществляется требуемое преобразование. Слева (рис. 6.2, *а*) приведен исходный символ, а справа (рис. 6.2, *б*) - то, что получится после работы программы DBLSYM. Сравнивая два этих рисунка, можно заметить, что правый символ получается в результате повторения каждого вертикального ряда битов, взятого из левого символа (напомним, что действие программы BIGSYM приводит к повтору горизонтальных рядов битов исходного символа).



а б

Рис. 6.2. Действие процедуры DBLSYM

Поскольку работа этой процедуры не столь очевидна, как может показаться с первого взгляда, то для лучшего ее понимания мы приводим небольшую схемку (рис. 6.3), отражающую пути перемещений битов при многократном выполнении команд сдвигов. Напоминаем, что эти команды вместе с аналогичными схемами приведены в разделе «Организация циклов в ассемблере» пятой главы.

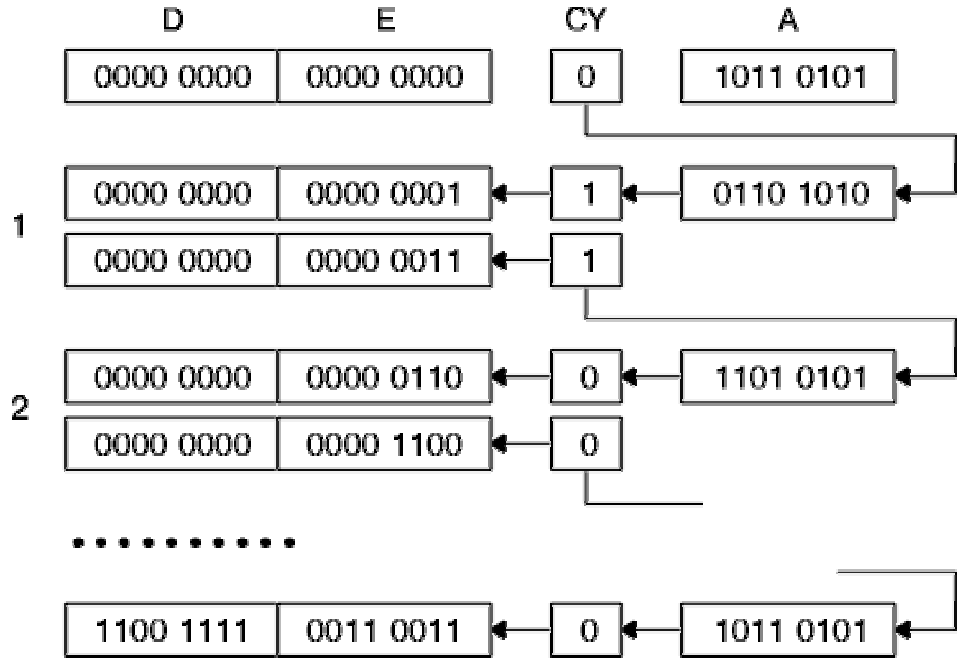


Рис. 6.3. Перемещения битов

```
        ORG   60000
        ENT   $
        LD    A,(23296)    ;начало - как в процедуре BIGSYM
DBLSYM  LD    L,A
        LD    H,0
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        LD    DE,(23606)
        ADD   HL,DE
        EX    DE,HL
        LD    HL,(23684)
        LD    B,8          ;количество повторений внешнего цикла
        PUSH  HL
; Во внешнем цикле по очереди берутся 8 байтов исходного символа
;  для преобразования во внутреннем цикле
DBLS1   PUSH  BC           ;сохраняем BC для внешнего цикла
        PUSH  DE           ;сохраняем текущий адрес
```

```
                        ; в символьном наборе
        LD    A,(DE)
        LD    DE,0          ;подготавливаем пару DE для последующего
                            ; формирования в ней расширенного
                            ; байта символа
        LD    B,8           ;количество повторений внутреннего цикла
                            ; (равное числу бит в одном байте)
; Во внутреннем цикле каждый бит исходного символа дважды копируется
; в регистровую пару DE, где в конце концов получаем расширенный
; в два раза байт (рис. 6.3)
DBLS2   RLCA
        PUSH  AF
        RL    E
        RL    D
        POP   AF
        RL    E
        RL    D
        DJNZ  DBLS2
        LD    (HL),D        ;вывод преобразованных байтов из
        INC   HL            ; пары DE на экран
        LD    (HL),E
        DEC   HL
        POP   DE            ;восстанавливаем значение DE
        INC   H
        INC   DE            ;переходим к обработке следующего
                            ; байта из набора символов
        POP   BC            ;восстанавливаем BC для внешнего цикла
        DJNZ  DBLS1
        POP   HL
        INC   HL            ;переходим к следующему
        INC   HL            ; знакоместу экрана
        LD    (23684),HL    ;записываем в системную переменную
                            ; адрес следующего символа
        RET
```

Чтобы напечатать на экране строку с каким-то текстом, можно воспользоваться той же программкой на Бейсике, что и для демонстрации процедуры BIGSYM.

Теперь в вашем распоряжении имеются три программы, в той или иной мере изменяющие форму стандартных символов и, чтобы лучше понять их работу, можно порекомендовать слегка «поиздеваться» над символами, создавая комбинированные программы. Например, часть символа увеличить по высоте, а оставшуюся часть - по ширине, или BIGSYM и DBLSYM соединить с программой BOLD. Надо сказать, что последний вариант дает совсем недурные результаты, в чем вы сможете убедиться, дойдя до последнего раздела этой главы.

КАК СДЕЛАТЬ НАЗВАНИЕ ИГРЫ

Думается, вы согласитесь, какую важную роль играет в заставке изображение названия игры и все, что его окружает. Для доказательства достаточно загрузить какую-нибудь фирменную игру и убедиться, что так оно и есть. Если вы достаточно внимательно изучили предыдущие страницы, то уже должны быть в состоянии изменить стандартный набор символов и сделать вполне приличное название. Но все же хотелось бы научиться создавать что-то оригинальное и, кроме того, независимое от стандартного фонта. Можно, конечно, загрузить Art Studio или The Artist II и попробовать нарисовать название там, но кто-то после первых же штрихов сделает простой вывод: пожалуй, это не для меня. Учитывая все сказанное, мы хотим предложить способ получения названий (и не только их), в основе которого лежит хорошо знакомый вам оператор DRAW.

Чтобы не утомлять вас долгими рассуждениями о том, каким образом формируется та или иная буква, лучше всего приведем в качестве своеобразного комментария программу на Бейсике с краткими пояснениями к ней, которые предлагаем внимательно изучить, после чего будет совсем нетрудно понять принцип работы соответствующей ей ассемблерной программы.

```
  10 READ y: IF y=-1 THEN GO TO 100
  20 READ x,hgt,len,ink,dx
  30 LET y=113-y: LET x=x+40
  40 INK ink
  50 FOR i=1 TO hgt
  60 PLOT x,y: DRAW len,0
  70 PLOT x,y+1: DRAW len,0
  80 LET y=y-4: LET x=x+dx: NEXT i
  90 GO TO 10
 100 STOP
1000 REM --- Буква «Н» ---
1010 DATA 0,0,12,6,2,0
1020 DATA 20,6,2,14,2,0
1030 DATA 0,20,12,10,2,0
1040 REM --- Буква «А» ---
1050 DATA 0,46,12,4,1,-1
1060 DATA 0,46,12,10,1,1
1070 DATA 24,42,2,16,1,0
1080 REM --- Буква «Р» ---
1090 DATA 0,72,12,10,3,0
1100 DATA 0,82,2,13,3,0
1110 DATA 24,82,2,13,3,0
1120 DATA 4,96,1,2,3,0
1130 DATA 24,96,1,2,3,0
1140 DATA 8,92,4,8,3,0
1150 REM --- Буква «Д» ---
1160 DATA 0,112,10,4,6,-1
1170 DATA 0,112,10,10,6,1
1180 DATA 40,100,2,34,6,0
1190 REM --- Буква «Ы» ---
1200 DATA 0,139,12,10,4,0
1210 DATA 16,150,2,13,4,0
1220 DATA 40,150,2,13,4,0
1230 DATA 20,163,1,2,4,0
1240 DATA 40,163,1,2,4,0
1250 DATA 24,160,4,8,4,0
1260 DATA 0,167,4,8,4,0
1270 DATA 16,169,1,6,4,0
1280 DATA 20,171,6,4,4,0
1290 DATA 44,169,1,6,4,0
1300 DATA -1
```

Теперь скажем несколько слов об этой программе, поскольку все основные идеи, заложенные в ней, затем будут реализованы в программе на ассемблере, но в начале покажем, что вы увидите на экране после ее исполнения (рис. 6.4).

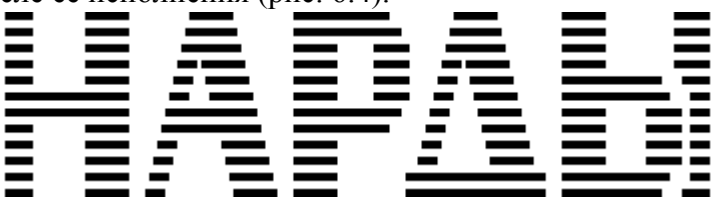


Рис. 6.4. Название для игры НАРДЫ

Вот такая симпатичная надпись, в которой каждая буква составлена из отрезков параллельных двойных линий, нарисованных DRAW'ами. В высоту укладывается двенадцать таких линий плюс промежутки между ними. Точно такие любой из вас сможет легко начертить по линейке на листе бумаги, измерить их длины и координаты, после чего ввести в программу свои вычисления в виде блока данных. Посмотрим, как это делается. Прежде всего разберемся со структурой строки блока данных, поскольку с подобным построением раньше мы еще не встречались, а именно оно все и определяет. Строка списка DATA содержит шесть элементов, каждому из которых в программе соответствует определенная переменная, а именно:

`y` - вертикальная координата начала линии (в пикселях);
`x` - горизонтальная координата начала линии (опять же в пикселях);
`hgt` - высота буквы (количество двойных линий плюс промежутки между ними);
`len` - длина проводимой горизонтальной линии (снова в пикселях);
`ink` - цвет линии (0...7);
`dx` - приращение линии (еще раз в пикселях).

Работу программы лучше всего показать на примере какой-нибудь буквы, скажем, «Д» (рис. 6.5, *а*), при построении которой участвуют все эти переменные (правда, некоторые величины в списках DATA равны нулю).

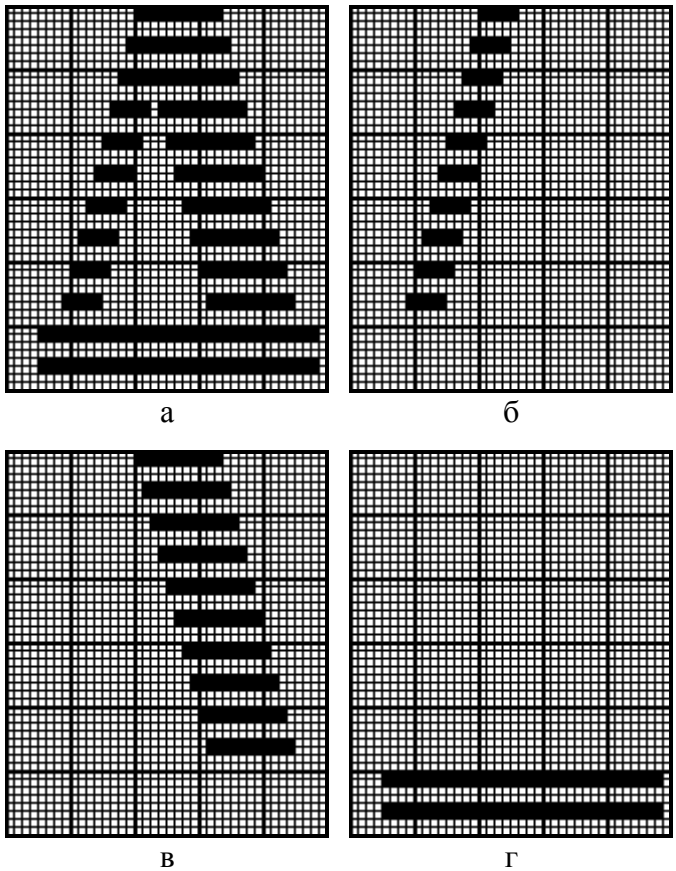


Рис. 6.5. Построение буквы «Д»

10, **20** - в этих строках порция данных переписывается из строки 1160 в переменные, после чего они принимают значения: `y=0`, `x=112`, `hgt=10`, `len=4`, `ink=6` и `dx=-1`.
30 - центровка названия.
40 - устанавливается цвет линий (желтый).
50 - начало цикла рисования фрагмента буквы.
60, **70** - рисуются подряд две горизонтальные линии (длиной по `len=4` пикселя).
80 - координаты `y` и `x` изменяются, после чего в этом же цикле выводится следующая пара линий, и так - 10 раз (`hgt`), в результате получится левый элемент буквы

(рис. 6.5, *б*).

90 - после того, как `hgt` пар линий будет напечатано, переход на начало.

На следующем этапе построения буквы «Д» прочитывается вторая порция данных из строки 1170: `y=0`, `x=112`, `hgt=10`, `len=10`, `ink=6`, `dx=1` и рисуется правая часть буквы (рис. 6.5, *в*), каждая линия которой составляет 10 пикселей. Наконец, на основании третьей порции (строка 1180): `y=40`, `x=100`, `hgt=2`, `len=34`, `ink=6` и `dx=0` ставятся две последние линии по 34 пикселя (рис. 6.5, *г*), завершающие построение буквы «Д».

После того как мы выяснили способ формирования каждой буквы средствами Бейсика, перейдем непосредственно к ассемблерной программе строка за строкой. Но прежде чем приступить к описанию текста, следует заметить, что к этому моменту мы наговорили вам довольно много всякого, поэтому небольшие по размерам программы нет смысла разбивать на отдельные части и подробно распространяться о каждой из них, как мы это делали в четвертой и пятой главах. Думается, что сейчас будет уже вполне достаточно прокомментировать отдельные команды и только в случаях, представляющих особый интерес, рассмотреть более детально некоторые группы команд.

```
        ORG   60000
        LD    HL,DATA
; Считывание параметров из блока данных
CICL1   LD    B,(HL)       ;Y-координата начала линии
        INC   B
        RET   Z            ;выход, если это конец блока данных
        LD    A,112        ;отсчитывать будем не снизу, а сверху.
                           ; Кроме того, сразу добавляем смещение
                           ; от верхнего края в 64 пикселя и
                           ; увеличиваем на 1, поскольку выполнялась
                           ; команда INC B (175-64+1=112)
        SUB   B
        LD    B,A
        INC   HL
        LD    A,(HL)       ;X-координата начала линии
        ADD   A,40         ;центрируем надпись
        LD    C,A
        INC   HL
        LD    D,(HL)       ;высота рисуемой части буквы
        INC   HL
        LD    E,(HL)       ; и ее ширина
        INC   HL
        LD    A,16         ;управляющий код для INK
        RST   16
        LD    A,(HL)       ;код цвета
        RST   16
        INC   HL
        LD    A,(HL)       ;наклон рисуемой части
        INC   HL
        PUSH  HL           ;сохраняем текущий адрес в блоке данных
; Цикл рисования части буквы, описанной очередной порцией данных
CICL2   PUSH  AF           ;сохраняем в стеке аккумулятор
        CALL  DRAW         ;чертим первую линию
        DEC   B            ;вторая линия будет на 1 пиксель ниже
        CALL  DRAW         ;чертим вторую линию
        DEC   B            ;переходим ниже
        DEC   B            ;пропускаем еще два ряда пикселей
        DEC   B
        POP   AF           ;восстанавливаем в аккумуляторе
                           ; параметр наклона
        PUSH  AF
        ADD   A,C          ;смещаем X-координату для
        LD    C,A          ; получения наклона
```

```
        POP   AF
        DEC   D           ;повторяем заданное параметром
        JR    NZ,CICL2    ; «высота» количество раз
        POP   HL          ;восстанавливаем адрес в блоке данных
        JR    CICL1       ;переходим на начало
; Подпрограмма рисования одной горизонтальной линии
DRAW    PUSH  BC          ;сохраняем нужные регистры
        PUSH  DE          ; в машинном стеке
        CALL  8933        ;ставим точку с координатами
                          ; из регистров B и C
        POP   DE          ;регистр E нам понадобится,
                          ; поэтому восстанавливаем
        PUSH  DE          ; и снова сохраняем
        EXX
        PUSH  HL          ;сохраняем регистровую пару HL'
        EXX
        LD    C,E         ;длина линии (горизонтальное смещение)
        LD    B,0         ;вертикальное смещение - 0 (линия
                          ; идет горизонтально)
        LD    DE,#101     ;положительное значение
                          ; для обоих смещений
        CALL  9402        ;рисуем линию
        EXX
        POP   HL          ;восстанавливаем регистры из стека
        EXX
        POP   DE
        POP   BC
        RET

; Блоки данных для рисования букв

; буква «Н»
DATA    DEFB  0,0,12,6,2,0
        DEFB  20,6,2,14,2,0
        DEFB  0,20,12,10,2,0
; буква «А»
        DEFB  0,46,12,4,1,-1
        DEFB  0,46,12,10,1,1
        DEFB  24,42,2,16,1,0
; буква «Р»
        DEFB  0,72,12,10,3,0
        DEFB  0,82,2,13,3,0
        DEFB  24,82,2,13,3,0
        DEFB  4,96,1,2,3,0
        DEFB  24,96,1,2,3,0
        DEFB  8,92,4,8,3,0
; буква «Д»
        DEFB  0,112,10,4,6,-1
        DEFB  0,112,10,10,6,1
        DEFB  40,100,2,34,6,0
; буква «Ы»
        DEFB  0,139,12,10,4,0
        DEFB  16,150,2,13,4,0
        DEFB  40,150,2,13,4,0
        DEFB  20,163,1,2,4,0
        DEFB  40,163,1,2,4,0
        DEFB  24,160,4,8,4,0
        DEFB  0,167,4,8,4,0
        DEFB  16,169,1,6,4,0
        DEFB  20,171,6,4,4,0
        DEFB  44,169,1,6,4,0
        DEFB  -1          ;указатель конца блока данных
```

Не решите нечаянно, что таким методом допускается рисовать исключительно буквы. В заставке и, безусловно, в других частях программы ему найдется немало областей

применения, например, для создания всевозможных орнаментов, статических рисунков, а если ее слегка дополнить, то возможно даже создавать элементы игрового пространства, скажем, лабиринты.

СКРОЛЛИНГИ ОКОН

Плавное перемещение изображений по экрану в разных направлениях можно достаточно часто увидеть в компьютерных играх. Пожалуй, легче сказать, где оно не используется, чем наоборот, поэтому решение такой задачи представляется нам достаточно важным. Мы уже показали, как формировать самые разные окна, а теперь попробуем написать несколько процедур для их плавного перемещения (скроллинга) во всех четырех направлениях. Затем на примере двух программ покажем, как применять такие процедуры для решения конкретных задач. Учтите, что каждая из приведенных ниже программ скроллинга сдвигает изображение в окне только на один пиксель. Следовательно, если вам потребуется переместить изображение в какую-то сторону, скажем, на 50 пикселей, то соответствующую процедуру следует выполнить несколько раз подряд в цикле, например:

```
        LD    B,50        ;количество сдвигов
LOOP    PUSH  BC          ;сохраняем содержимое регистра B
        ..........        ;процедура скроллинга
        POP   BC          ;восстанавливаем регистр B
        DJNZ  LOOP
        RET
```

До того, как мы приступим к детальному описанию процедуры скроллинга окна вверх, желательно рассмотреть все команды и подпрограммы ПЗУ, которые встречаются здесь впервые. Таких наберется всего две: чрезвычайно полезная команда LDIR, перемещающая блок памяти с инкрементом (т. е. с увеличением содержимого регистров, в которых записаны адреса пересылок) и подпрограмма ПЗУ, расположенная по адресу 8880. Рассмотрим их в том порядке, как они встречаются в программе.

Процедура 8880 вычисляет адрес байта в видеобуфере по координатам точки, заданным в пикселях. Началом отсчета считается левый верхний угол экрана. Таким образом, входными данными к подпрограмме являются:

- вертикальная координата, помещаемая в аккумулятор;
- горизонтальная координата, помещаемая в регистр C,

а выходными:

- в регистровой паре HL возвращается вычисленный адрес байта видеобуфера;
- в регистр A помещается значение от 0 до 7, численно равное величине смещения заданной точки в пикселях от левого края того знакоместа, для которого рассчитывается адрес.

Более подробно рассмотрим действие команды LDIR. С ее помощью группа байтов, расположенных в сторону увеличения адресов от ячейки, на которую указывает HL, пересылается в область памяти, адресуемую регистровой парой DE. Количество передаваемых байтов определяется регистровой парой BC. Чтобы почувствовать всю прелесть этой команды и оценить ее по достоинству, приведем текст цикла, выполняющего то же, что и LDIR.

```
MET     LD     A,(HL)
        LD     (DE),A
        INC    HL
        INC    DE
        DEC    BC
        LD     A,B
        OR     C
        JR     NZ,MET
        RET
```

Глядя на этот фрагмент, можно заметить очевидные достоинства команды LDIR: в цикле изменяется регистр A, а в LDIR он не затрагивается, кроме того, программа занимает 8 строк текста вместо одной и работает примерно в два с половиной раза медленнее.

Если в программу ввести исходные данные, то может получиться эффект, который мы наблюдаем в некоторых играх, например, в SOKOBAN'e. Суть его состоит в том, что из ПЗУ с адреса 0 переписываются 6144 байта в область экранной памяти, начиная с адреса 16384. Если это действие повторять в цикле, то создастся впечатление бегущей по экрану ряби:

```
        LD     BC,6144
        LD     HL,0
        LD     DE,16384
        LD     A,255
MET     PUSH   BC
        PUSH   DE
        PUSH   HL
        LDIR
        POP    HL
        POP    DE
        POP    BC
        INC    HL
        DEC    A
        JR     NZ,MET
        RET
```

Кроме рассмотренной команды LDIR в ассемблерных программах довольно широко используются и другие команды пересылок байтов:

LDDR - перемещение блока памяти с декрементом. Ее действие аналогично команде LDIR, только пересылается группа байтов, расположенных в сторону уменьшения адресов от ячейки, на которую указывает HL. Количество передаваемых байтов также определяется в BC.

LDI - пересылка содержимого одной ячейки памяти с инкрементом. Байт из ячейки, адресуемой регистровой парой HL, переносится в ячейку, адресуемую парой DE; содержимое HL и DE увеличивается на 1, а BC уменьшается на 1. Если в результате выполнения команды BC=0, то флаг P/V сбрасывается, в противном случае P/V=1.

LDD - пересылка содержимого одной ячейки памяти с декрементом. Действие аналогично команде LDI, только содержимое регистровых пар HL и DE уменьшается на 1.

Покончив с теорией, можно заняться более приятным делом и написать процедуры для скроллинга окон для всех направлений. Начнем со смещения окна вверх. Перед вызовом этой подпрограммы нужно определить уже известные по предыдущим примерам переменные ROW, COL, HGT и LEN, записав в них координаты и размеры окна. Предварительно не помешает убедиться, что окно не выходит за пределы экрана, так как в целях упрощения программы подобные проверки в ней не выполняются. Добавим, что это в равной мере относится и к другим процедурам скроллингов.

```
SCR_UP LD    A,(COL)
       LD    C,A
       LD    A,(HGT)
       LD    B,A
       LD    A,(ROW)
; Значения из переменных ROW, COL и HGT умножаем на 8,
;  то есть переводим знакоместа в пиксели
       SLA   A
       SLA   A
       SLA   A
       SLA   B
       SLA   B
       SLA   B
       DEC   B           ;потому что один ряд пикселей просто
                         ; заполняется нулями
       SLA   C
       SLA   C
       SLA   C
       PUSH  AF
       PUSH  BC
       CALL  8880        ;вычисляем адрес верхнего левого угла окна
       POP   BC
       POP   AF
SCRUP1 INC   A           ;следующий ряд пикселей
       PUSH  AF
       PUSH  BC
       PUSH  HL
       CALL  8880        ;вычисляем адрес
       POP   DE
       PUSH  HL
       LD    A,(LEN)     ;пересылаем столько байт, сколько
                         ; умещается по ширине окна
       LD    C,A
       LD    B,0
       LDIR
       POP   HL
       POP   BC
       POP   AF
       DJNZ  SCRUP1
       LD    (HL),0      ;в последний ряд пикселей записываем нули
       LD    D,H
       LD    E,L
       INC   DE
       LD    A,(LEN)     ;по ширине окна,
       DEC   A           ; минус 1
       RET   Z           ;выходим, если только одно знакоместо
       LD    C,A
       LD    B,0
       LDIR              ;иначе обнуляем и все остальные байты ряда
       RET
```

Обратите особое внимание на последние строки процедуры, где в нижний ряд пикселей записываются нулевые байты. Этот прием весьма распространен и применяется для заполнения любой области памяти произвольным значением. Чтобы понять идею, нужно хорошо представлять, как работает команда LDIR. Сначала в первый байт массива, адресуемый парой HL, заносится какой-то определенный байт, в DE переписывается значение из HL и увеличивается на 1, в BC задается уменьшенный на единицу размер заполняемого массива, а дальше с выполнением команды LDIR происходит следующее. Байт из первого адреса (HL) переписывается во второй (DE), затем DE и HL увеличиваются, то есть HL будет указывать на второй байт, а DE - на третий. На следующем круге число из второго адреса пересылается в третий, но после первого выполнения цикла второй байт уже содержит ту же величину, что и первый, поэтому второй и третий байты к этому моменту станут равны первому. И так далее, до заполнения всего массива.

Практическое применение такому методу найти нетрудно. Кроме процедур вертикального скроллинга окон он может использоваться, например, для очистки экрана, инициализации блоков данных и других нужд. Попробуйте сами написать процедуру, аналогичную оператору CLS, в которой участвовала бы команда LDIR.

Вернемся к рассмотрению подпрограмм скроллингов окон. Процедура сдвига окна вниз во многом похожа на предыдущую:

```
SCR_DN LD    A,(COL)
       LD    C,A
       LD    A,(HGT)
       LD    B,A
       LD    A,(ROW)
       ADD   A,B          ;начинаем перемещать изображение не
                          ; сверху, как в SCR_UP, а снизу
       SLA   A
       SLA   A
       SLA   A
       DEC   A
       SLA   B
       SLA   B
       SLA   B
       DEC   B
       SLA   C
       SLA   C
       SLA   C
       PUSH  AF
       PUSH  BC
       CALL  8880
       POP   BC
       POP   AF
SCRDN1 DEC   A            ;следующий ряд пикселей (идем вверх)
       PUSH  AF
       PUSH  BC
       PUSH  HL
       CALL  8880
       POP   DE
       PUSH  HL
       LD    A,(LEN)
       LD    C,A
       LD    B,0
       LDIR
       POP   HL
       POP   BC
       POP   AF
       DJNZ  SCRDN1
       LD    (HL),0
       LD    D,H
       LD    E,L
       INC   DE
       LD    A,(LEN)
       DEC   A
       RET   Z
       LD    C,A
       LD    B,0
       LDIR
       RET
```

Теперь приведем подпрограмму, выполняющую скроллинг окна влево. Она уже совсем не похожа на предшествующие, но в принципе повторяет известную вам процедуру SCRLIN, описанную в разделе «Бегущая строка» предыдущей главы. Но это и понятно: ведь там мы также скроллировали окно, только оно имело фиксированные размеры в одно знакоместо

высотой и 32 - шириной. Здесь же мы приводим универсальную процедуру, но и с ее помощью можно получить тот же эффект.

```
SCR_LF LD    A,(HGT)      ;количество повторений такое же,
       LD    B,A          ; сколько строк занимает окно
       LD    A,(ROW)      ;номер верхней строки
SCRLF1 PUSH  AF           ;дальше все очень похоже на процедуру SCRLIN
       PUSH  BC
       CALL  3742
       LD    A,(COL)
       LD    B,A
       LD    A,(LEN)
       DEC   A
       ADD   A,B
       ADD   A,L
       LD    L,A
       LD    B,8
SCRLF2 PUSH  HL
       LD    A,(LEN)
       AND   A
SCRLF3 RL    (HL)
       DEC   HL
       DEC   A
       JR    NZ,SCRLF3
       POP   HL
       INC   H
       DJNZ  SCRLF2
       POP   BC
       POP   AF
       INC   A
       DJNZ  SCRLF1
       RET
```

И наконец, для полного комплекта, напишем соответствующую подпрограмму, выполняющую скроллинг окна вправо:

```
SCR_RT LD    A,(HGT)
       LD    B,A
       LD    A,(ROW)
SCRRT1 PUSH  AF
       PUSH  BC
       CALL  3742
       LD    A,(COL)
       ADD   A,L
       LD    L,A
       LD    B,8
SCRRT2 PUSH  HL
       LD    A,(LEN)
       AND   A
SCRRT3 RR    (HL)
       INC   HL
       DEC   A
       JR    NZ,SCRRT3
       POP   HL
       INC   H
       DJNZ  SCRRT2
       POP   BC
       POP   AF
       INC   A
       DJNZ  SCRRT1
       RET
```

А сейчас рассмотрим программу, в которой демонстрируются возможности вертикального скроллинга вверх. После запуска вы увидите две быстро сменяющие друг друга картинки: сначала весь экран заполняется красивым орнаментом (фактурой), а затем средняя его часть

стирается процедурой очистки окна. После этого строка за строкой снизу вверх начнет медленно перемещаться текст с правилами игры (рис. 6.6), который обычно вызывается из меню как один из кадров заставки.

Рис. 6.6. Правила игры

Перед запуском ассемблерной программы следует с адреса 64768 загрузить набор русских букв, однако при желании вы можете использовать и аналогичные латинские, выполнив некоторые изменения в программе. О том, как это сделать, вполне достаточно сказано в четвертой главе.

```
       ORG   60000
       ENT   $
; Задание адресов новых фонтов и постоянных атрибутов экрана
LATF   EQU   64000-256
RUSF   EQU   LATF+768
       LD    A,6
       LD    (23693),A
       LD    A,0
       CALL  8859
       CALL  3435
       LD    A,2
       CALL  5633
; Ввод символа UDG для рисования фактуры
       LD    HL,UDG
       LD    (23675),HL
; Заполнение всего экрана фактурой
       CALL  DESK
; Начало основной программы
       LD    HL,RUSF
       LD    (23606),HL
NEW    LD    HL,TEXT
CYCLE1 CALL  PRINT
       LD    B,10
CYCLE2 PUSH  HL
       PUSH  BC
       CALL  SCR_UP
       CALL  PAUSE
```

```
        POP   BC
        POP   HL
        JR    Z,EXIT
        DJNZ  CYCLE2
        LD    A,(HL)
        AND   A
        JR    NZ,CYCLE1
        LD    B,80          ;расстояние между текстами
CYCLE3  PUSH  BC
        CALL  SCR_UP
        CALL  PAUSE
        POP   BC
        JR    Z,EXIT
        DJNZ  CYCLE3
        JR    NEW
; Восстановление стандартного набора символов и выход
EXIT    LD    HL,15360
        LD    (23606),HL
        RET
; Для того, чтобы строки выводимого текста можно было успеть прочитать,
;  в цикл вставлена подпрограмма задержки, а нажав клавишу Space,
;  можно в любой момент завершить работу программы.
PAUSE   LD    BC,6
        CALL  7997
        LD    A,(23560)
        CP    " "
        RET
; Подпрограмма печати строки текста
PRINT   PUSH  BC
        LD    DE,PAR
        LD    BC,5
        CALL  8252
        LD    B,24          ;число символов в строке
PR_CI   LD    A,(HL)
        RST   16
        INC   HL
        DJNZ  PR_CI
        POP   BC
        RET
PAR     DEFB  22,15,4,16,0
; Подпрограмма заполнения экрана фактурой
DESK    LD    BC,704
        LD    A,#14
        RST   16
        LD    A,1
        RST   16
DESK1   LD    A,144
        RST   16
        DEC   BC
        LD    A,B
        OR    C
        JR    NZ,DESK1
        LD    A,#14
        RST   16
        LD    A,0
        RST   16
; Создание в фактуре окна для вывода текста
        CALL  CLSV
        CALL  SETV
        RET
; Подпрограммы
CLSV    .........
SETV    .........
SCR_UP  .........
; Данные для окна с текстом
```

```
COL1    DEFB  4
ROW1    DEFB  4
LEN1    DEFB  24
HGT1    DEFB  12
ATTR    DEFB  %01000010
; Данные для окна в фактуре
COL     DEFB  3
ROW     DEFB  3
LEN     DEFB  26
HGT     DEFB  13
; Данные для выводимого в окно текста
TEXT    DEFM  "····P·I·R·A·M·I·D·A·····"
        DEFM  "·························"
        DEFM  "Celx igry sostoit w tom,"
        DEFM  "~toby··perwym··postawitx"
        DEFM  "swoi··fi{ki··na··wer{inu"
        DEFM  "piramidy. Dlq |togo nuv-"
        DEFM  "no ispolxzowatx gorizon-"
        DEFM  "talxnye··i··wertikalxnye"
        DEFM  "peredwiveniq fi{ek,krome"
        DEFM  "togo,velatelxno wospolx-"
        DEFM  "zowatxsq··dopolnitelxny-"
        DEFM  "mi prodwiveniqmi,···sutx"
        DEFM  "kotoryh··sostoit··w tom,"
        DEFM  "~toby sostawitx fi{ki··w"
        DEFM  "wide treugolxnika, posle"
        DEFM  "~ego sdelatx hod werhnej"
; При желании текст можно продолжить
        DEFB  0
; Данные для фактуры вокруг окна
UDG     DEFB  248,116,34,71,143,7,34,113
```

Два вида горизонтального скроллинга продемонстрируем на примере еще одной программы, где эти процедуры действуют одновременно. Сразу после запуска программа создает на экране фрагмент средневекового замка из красного кирпича с зубчатыми стенами и бойницами, а в средней его части - ажурные ворота. До тех пор, пока никакая клавиша не нажата, изображение неподвижно. После нажатия любой из них створки ворот начинают медленно расходиться в разные стороны (рис. 6.7), создавая совершенно бесподобный эффект.

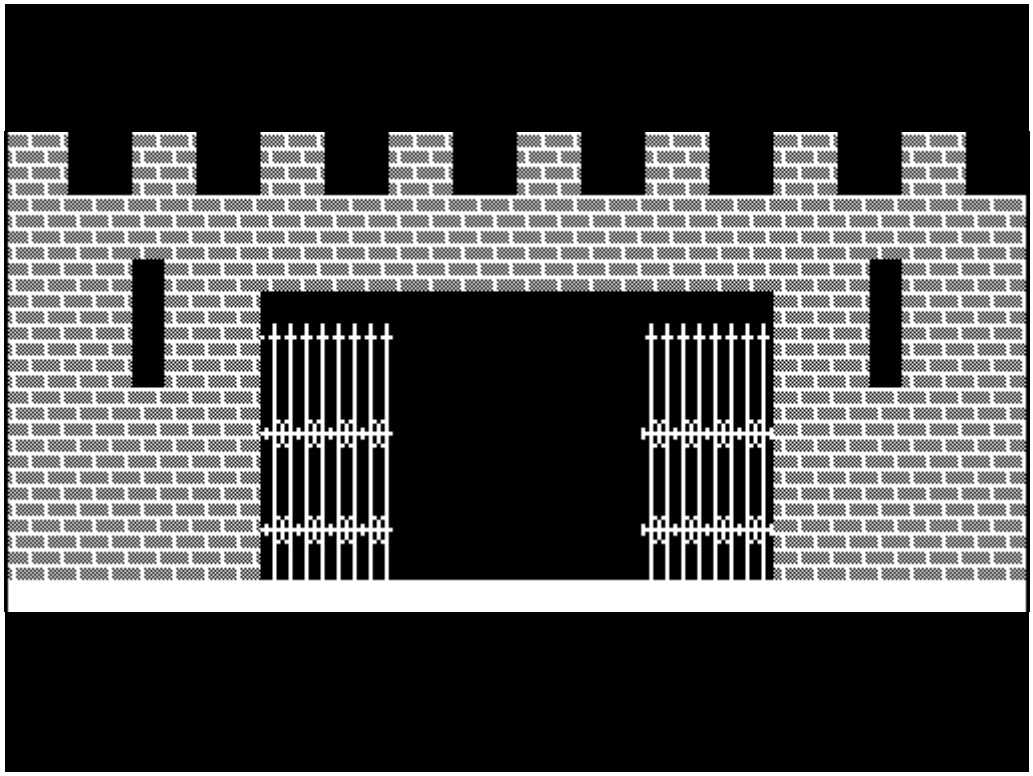


Рис. 6.7. Раскрытие ворот замка скроллингом окон

А теперь перейдем непосредственно к программе.

```
       ORG   60000
       ENT   $
; Подготовка экрана
       LD    A,7
       LD    (23693),A
       XOR   A
       CALL  8859
       CALL  3435
       LD    A,2
       CALL  5633
; Назначение нового адреса символов UDG
       LD    HL,UDG
       LD    (23675),HL
; Вывод на экран изображения крепостной стены с воротами
       CALL  SCRN
       LD    BC,0
       CALL  7997
; Основная часть программы, в которой створки ворот
;  со стуком разъезжаются в разные стороны
       LD    B,64
MAIN   PUSH  BC
       CALL  SCR_LF
       LD    A,0
       OUT   (254),A
       CALL  SCR_RT
       LD    A,16
       OUT   (254),A
       LD    BC,6
       CALL  7997
       POP   BC
       DJNZ  MAIN
       RET
; Формирование изображения крепостной стены с воротами:
; Изображение стены
SCRN   LD    DE,D_WALL
       LD    BC,7
       CALL  8252
       LD    BC,384
SCRN1  LD    A,147
       RST   16
       DEC   BC
       LD    A,B
       OR    C
       JR    NZ,SCRN1
; Зеленая трава
       LD    DE,D_GRAS
       LD    BC,5
       CALL  8252
       LD    B,32
SCRN2  LD    A," "
       RST   16
       DJNZ  SCRN2
; Зубцы на стене
       LD    BC,#400     ;AT 4,0
       CALL  PR_AT
       LD    B,16
SCRN3  LD    DE,D_BATT
       PUSH  BC
       LD    BC,10
```

```
       CALL  8252
       POP   BC
       DJNZ  SCRN3
; Ворота
       LD    BC,#908     ;AT 9,8
       CALL  PR_AT
       LD    B,16
SCRN4  LD    A," "
       RST   16
       DJNZ  SCRN4
; Бойницы
       LD    H,4
SCRN5  LD    A,H
       ADD   A,7
       LD    B,A
       LD    C,4
       CALL  PR_AT
       LD    A," "
       RST   16
       LD    C,27
       CALL  PR_AT
       LD    A," "
       RST   16
       DEC   H
       JR    NZ,SCRN5
; Штыри решетки
       LD    A,16
       RST   16
       LD    A,5
       RST   16
       LD    B,10        ;Y
       LD    H,8
SCRN6  LD    L,16
       LD    C,8         ;X
SCRN7  CALL  PR_AT
       LD    A,145
       RST   16
       INC   C
       DEC   L
       JR    NZ,SCRN7
       INC   B
       DEC   H
       JR    NZ,SCRN6
; Пики решетки
       LD    BC,#A08     ;AT 10,8
       CALL  PR_AT
       LD    B,16
SCRN8  LD    A,144
       RST   16
       DJNZ  SCRN8
; Узор решетки
       LD    L,2
       LD    B,13
SCRN9  LD    C,8
       CALL  PR_AT
       LD    B,16
SCRN10 LD    A,146
       RST   16
       DJNZ  SCRN10
       LD    B,16
       DEC   L
       JR    NZ,SCRN9
       RET
; Подпрограмма позиционирования вывода спрайтов
PR_AT  LD    A,22
```

```
        RST   16
        LD    A,B
        RST   16
        LD    A,C
        RST   16
        RET
; Подпрограммы скроллинга окон
SCR_LF ........
SCR_RT ........
; Данные для левого окна
COL     DEFB  8
ROW     DEFB  10
HGT     DEFB  8
LEN     DEFB  8
; Данные для правого окна
COL1    DEFB  16
ROW1    DEFB  10
LEN1    DEFB  8
HGT1    DEFB  8
; Данные для рисования крепостной стены и решетки
UDG     DEFB  34,34,34,119,34,34,34,34   ;пики   (144)
        DEFB  34,34,34,34,34,34,34,34    ;штыри  (145)
        DEFB  54,42,170,255,170,42,54,34 ;узор   (146)
        DEFB  255,2,2,2,255,32,32,32     ;кирпич (147)
; Данные позиционирования печати
D_BATT DEFB  17,2,16,7,147,147,17,0,32,32
D_WALL DEFB  22,6,0,17,2,16,7
D_GRAS DEFB  22,18,0,17,4
```

ВВОД ЭЛЕМЕНТА СЛУЧАЙНОСТИ

Любая игра потеряет всякий смысл, если действия компьютера можно будет предугадать на всех этапах развития сюжета. Чтобы придать персонажам видимость самостоятельности и непредсказуемости поведения, в игровых программах довольно широко используются так называемые случайные числа. Строго говоря, получить действительно случайные значения программным путем нет никакой возможности, вы можете лишь заставить компьютер вырабатывать более или менее длинную последовательность неповторяющихся величин, но в конце концов она все же начнет повторяться. Поэтому такие числа обычно называют псевдослучайными. В Бейсике для их получения используется функция RND, которая вырабатывает по определенному закону числа от 0 до 1 и далее они обычно преобразуются программными средствами в числа из заданного диапазона. В ассемблере работать с дробными величинами значительно сложнее, отчего программисты с этой целью редко прибегают к использованию подпрограмм ПЗУ, а пишут, как правило, свои аналогичные процедуры.

В качестве «случайных» чисел довольно часто используют последовательность кодов ПЗУ. Такой метод крайне прост и дает неплохую степень случайности в циклах. Если вы не забыли, именно такой метод мы применили в программе «растворения» символов, описанной в предыдущей главе. Сейчас же мы расскажем и о некоторых других способах получения псевдослучайных чисел.

Иногда «случайные» числа извлекают из системного регистра регенерации R. Поскольку его значение постоянно увеличивается после выполнения каждой команды микропроцессора, предугадать, что же он содержит в какой-то момент времени практически невозможно. Таким образом, простейший генератор случайных чисел может выглядеть так:

```
        LD    A,R
```

Но помните, что это справедливо только для достаточно разветвленных программ, особенно если их работа зависит от внешних воздействий (например, при управлении с помощью клавиатуры или джойстика). В коротких же циклах ни о какой непредсказуемости говорить не приходится.

Кроме того, есть и еще один недостаток использования регистра регенерации. Значение его никогда не превышает 127. Иными словами, седьмой бит этого регистра обычно «сброшен» в 0, и, дойдя до значения 127, он вновь обнуляется.

Однако, справедливости ради, стоит заметить, что это относится лишь к тем программам, в которых регистр R не изменяется принудительным образом. При желании вы можете установить его 7-й бит и тогда постоянно будете получать из него значения от 128 до 255. Правда, делается это обычно в целях защиты программ (например, такой метод применен в игре NIGHT SHADE), но это уже совсем другая тема.

Когда требуется получить наибольшую степень случайности, прибегают к математическим расчетам. Разберем одну из таких «математических» подпрограмм. Несмотря на ее простоту, она вырабатывает все же достаточно длинную последовательность неповторяющихся значений, чтобы их можно было рассматривать в качестве случайных.

```
RND255 PUSH  BC
       PUSH  DE
       PUSH  HL
; Регистровая пара HL загружается значением из счетчика «случайных» чисел
;  (это может быть, например, системная переменная 23670/23671,
;  которая используется Бейсиком для тех же целей)
       LD    HL,(ADDR)
       LD    DE,7          ;дальше следует расчет очередного
                           ; значения счетчика
       ADD   HL,DE
       LD    E,L
       LD    D,H
       ADD   HL,HL
       ADD   HL,HL
       LD    C,L
       LD    B,H
       ADD   HL,HL
       ADD   HL,BC
       ADD   HL,DE
       LD    (ADDR),HL     ;сохранение значения счетчика «случайных»
                           ; чисел для последующих расчетов
       LD    A,H           ;регистр A загружается значением
                           ; старшего байта счетчика
       POP   HL
       POP   DE
       POP   BC
       RET
ADDR   DEFW  0
```

Эта процедура возвращает в аккумуляторе «случайные» числа от 0 до 255. Однако в подавляющем большинстве случаев нужно иметь возможность получать значения из произвольного диапазона. С этой целью дополним подпрограмму RND255 расчетами по ограничению максимального значения и назовем новую процедуру просто RND. Перед обращением к ней в регистре E задается верхняя граница вырабатываемых «случайных» чисел. Например, для получения в аккумуляторе числа от 0 до 50 в регистр E нужно загрузить значение 51:

```
RND     CALL  RND255
        LD    L,A
        LD    H,0
        LD    D,H
        CALL  12457
        LD    A,H
        RET
RND255 .........
```

Здесь вновь появилась еще одна подпрограмма ПЗУ, расположенная по адресу 12457. Она выполняет целочисленное умножение двух чисел, записанных в регистровых парах DE и HL. Произведение возвращается в HL. Если в результате умножения получится число, превышающее 65535, то будет установлен флаг CY, иначе выполняется условие NC. Проверка переполнения может оказаться полезной, когда перемножаются не известные заранее величины. В подпрограмме RND это условие проверять не нужно, так как оба сомножителя не превышают величины 255 (H и D предварительно обнуляются).

После того, как мы получили в свое распоряжение подпрограмму генерации случайных чисел, рассмотрим один занимательный пример ее применения. Представьте себе некое подобие мишени, состоящей, как и положено, из окружностей и ряда цифр, характеризующих заработанные вами очки при попадании в ту или иную ее часть. Вы нажимаете любую клавишу компьютера и в ту же секунду раздается звук, очень похожий на пролетающую мимо вашего уха пулю, а в мишени появляется отверстие с рваными краями. Нажимаете еще раз - снова попадание, но уже совсем в другом месте (рис. 6.8) и изображение отверстия тоже стало каким-то другим. Повторив эту процедуру много раз, вы легко можете убедиться в том, что «пули», как и при настоящей стрельбе, ложатся на мишень совершенно случайно. То же самое можно сказать и о характере отверстий. Отсюда ясно, что программа, реализующая эту игрушку, должна вырабатывать для каждого выстрела три случайных числа: координаты X и Y места попадания и номер изображения для пулевого отверстия. Теперь можно обратиться к самой программе и кратко ее прокомментировать.

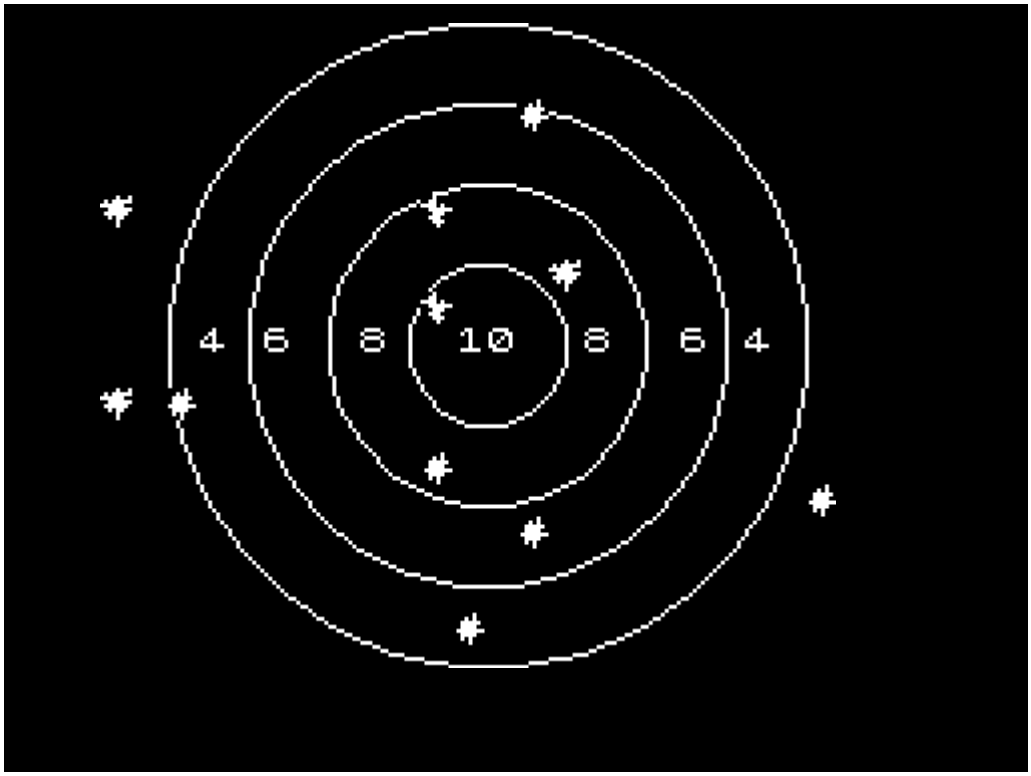


Рис. 6.8. Программа МИШЕНЬ

```
        ORG   60000
        ENT   $
; Задание постоянных атрибутов экрана
        LD    A,7
        LD    (23693),A
        XOR   A
        CALL  8859
        CALL  3435
        LD    A,2
        CALL  5633
; Ввод символов UDG - три «пулевые отверстия»
        LD    HL,UDG
        LD    (23675),HL

; Основная часть программы
        CALL  MISH          ;рисование мишени
MAIN    CALL  WAIT          ;ожидание нажатия любой клавиши
        CP    " "
        RET   Z
        LD    A,22
        RST   16
        LD    E,20          ;задание диапазона для координаты Y
        CALL  RND
        RST   16
        LD    E,30          ;задание диапазона для координаты X
        CALL  RND
        RST   16
        LD    A,16
        RST   16
        LD    A,6
        RST   16
        LD    E,3           ;задание номера «пулевого отверстия»
        CALL  RND
        ADD   A,144         ;вычисление кода спрайта
        RST   16
        CALL  SND           ;звуковой сигнал
        JR    MAIN

; Подпрограмма вывода на экран мишени
MISH    LD    C,20
        CALL  CIRC
        LD    C,40
        CALL  CIRC
        LD    C,60
        CALL  CIRC
        LD    C,80
        CALL  CIRC
        LD    DE,TEXT
        LD    BC,LENTXT
        JP    8252

; Подпрограмма рисования окружностей
CIRC    EXX
        PUSH  HL
        EXX
        PUSH  BC
        LD    A,120
        CALL  11560
        LD    A,90
        CALL  11560
        POP   BC
        LD    B,0
        CALL  11563
        CALL  9005
        EXX
```

```
        POP   HL
        EXX
        RET

; Подпрограмма остановки счета
WAIT    XOR   A
        LD    (23560),A
WAIT1   LD    A,(23560)
        AND   A
        JR    Z,WAIT1
        RET
; Подпрограммы
RND     .........
SND     LD    B,80
        LD    HL,150
        LD    DE,1
SND1    .........
; Данные для мишени
TEXT    DEFB  22,10,14
        DEFM  "10"
        DEFB  22,10,18
        DEFM  "8"
        DEFB  22,10,21
        DEFM  "6"
        DEFB  22,10,23
        DEFM  "4"
        DEFB  22,10,11
        DEFM  "8"
        DEFB  22,10,8
        DEFM  "6"
        DEFB  22,10,6
        DEFM  "4"
LENTXT  EQU   $-TEXT
; Данные для «пулевых отверстий»
UDG     DEFB  4,20,62,60,127,60,40,8
        DEFB  9,95,252,63,126,44,8,8
        DEFB  16,48,244,63,28,56,28,8
```

МНОГОКАДРОВАЯ ЗАСТАВКА

После того, как игра загрузится в память компьютера, скорее всего на экране появится основной кадр многокадровой заставки, который носит название «Меню» и где обычно перечисляются действия, связанные с настройкой игры. Поскольку в первой главе мы об этом уже говорили, то не будем повторяться и сразу перейдем к задачам, которые следует решать при создании такой заставки. На наш взгляд, таких задач будет три: формирование окон в заданных частях экрана, создание блоков данных с текстами для меню и всех кадров и, наконец, то, с чем мы раньше еще не встречались - составление программы управления всеми частями заставки.

Что касается окон, то в принципе этот вопрос нами уже решен и можно было бы к нему вновь не возвращаться. Однако использование подпрограмм, работающих с окнами в том виде, в котором мы их предложили ранее, не всегда удобно. Если все параметры окон фиксированы, то нелепо каждый раз переопределять переменные ROW, COL и иже с ними. Лучше составить блоки данных, содержащие сведения о размерах, местоположении и всех прочих характеристиках, а затем выполнять любые преобразования окон, передавая подпрограммам единственное значение - адрес таблицы параметров (то бишь блока данных).

В начале книги мы говорили о существовании такой группы регистров, как индексные. Группа эта немногочисленна и включает в себя лишь два регистра: IX и IY. Как мы уже сообщали, оба они состоят из 16 бит и разделить их на половинки «законными» методами невозможно, поэтому они обычно рассматриваются не как регистровые пары, а как отдельные регистры.

Но для каких целей они существуют, в каких ситуациях их удобно применять и в каких группах команд они могут встречаться? Обычно эти регистры используются при обработке массивов, блоков данных или таблиц, а употребляются они практически во всех типах команд, в которых может принимать участие регистровая пара HL. За более подробной информацией на этот счет можете обратиться к Приложению I, где в алфавитном порядке приведены все команды микропроцессора.

Известно, что в Бейсике к любому элементу массива можно обратиться по индексу, например, оператор `LET b=a(8)` присвоит переменной `b` значение 8-го элемента массива `a()`. В ассемблере подобная запись может выглядеть так:

```
        LD    B,(IX+7)
```

Обратите внимание, что первый элемент массива имеет индекс 0, а не 1. В отличие от массивов Бейсика, максимальный индекс у регистров IX и IY не может превышать 127, но зато допускаются отрицательные значения номера элемента массива. Таким образом, общий размер адресуемой области составляет 256 байтовых элементов, а индексный регистр указывает на его «середину».

Реальным примером большой структуры данных могут служить системные переменные Бейсика. Обычно они адресуются регистром IY, который указывает на переменную `ERR_NR`, находящуюся по адресу 23610. Кстати, именно в связи с этим регистр IY лучше оставить в покое и никак не изменять его в своих программах, по крайней мере, до тех пор, пока вы так или иначе используете операционную систему компьютера. Что касается регистра IX, то им вы можете смело пользоваться при любых обстоятельствах.

Наверное, лучше всего объяснить идею применения индексных регистров на конкретном примере. В качестве такого примера приведем уже знакомые вам процедуры очистки окон и установки в них постоянных атрибутов (тем более, что они понадобятся при составлении программы многокадровой заставки), но графические переменные заменим единой структурой, первый элемент которой адресуем через IX. Саму же структуру оставим без изменения, то есть на первом месте (по смещению 0, задаваемом как IX+0 или просто IX) по-прежнему будет находиться параметр COL, на втором (задаваемом как IX+1) - ROW, дальше LEN, HGT и ATTR. А чтобы новые процедуры отличить от описанных выше, изменим их имена на SETW и CLSW:

```
SETW    LD    DE,#5800
        LD    B,(IX+3)     ;HGT
        LD    C,(IX+2)     ;LEN
        LD    A,(IX+1)     ;ROW
        LD    L,A
        LD    H,0
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,DE
        LD    A,L
```

```
        ADD   A,(IX)       ;COL
        LD    L,A
        LD    A,(IX+4)     ;ATTR
SETW1   PUSH  BC
        PUSH  HL
SETW2   LD    (HL),A
        INC   HL
        DEC   C
        JR    NZ,SETW2
        POP   HL
        POP   BC
        LD    DE,32
        ADD   HL,DE
        DJNZ  SETW1
        RET
; -----------------
CLSW    LD    B,(IX+3)     ;HGT
        LD    C,(IX+2)     ;LEN
        LD    A,(IX+1)     ;ROW
CLSW1   PUSH  AF
        PUSH  BC
        PUSH  DE
        CALL  3742
        POP   DE
        LD    A,L
        ADD   A,(IX)       ;COL
        LD    L,A
        LD    B,8
CLSW2   PUSH  HL
        PUSH  BC
        LD    B,C
CLSW3   LD    (HL),0
        INC   HL
        DJNZ  CLSW3
        POP   BC
        POP   HL
        INC   H
        DJNZ  CLSW2
        POP   BC
        POP   AF
        INC   A
        DJNZ  CLSW1
        RET
```

Сравнив эти подпрограммы с SETV и CLSV, вы найдете много общего. Собственно, отличаются они друг от друга только способом передачи параметров. И это лишний раз доказывает, насколько ассемблер гибче всех прочих языков программирования - всякий раз вы можете использовать совершенно новые или видоизмененные процедуры, наиболее пригодные для применения именно в данном конкретном случае.

Составим блоки данных, включающих в себя по пять известных вам параметров для пяти различных окон, которые будут появляться на экране:

```
WIN0    DEFB  5,0,23,3,%01001110
WIN1    DEFB  8,6,17,13,%01010111
WIN2    DEFB  13,5,13,15,%01011110
WIN3    DEFB  13,9,13,11,%01101000
WIN4    DEFB  6,5,12,15,%01110101
```

Теперь перед обращением к процедурам SETW и CLSW достаточно в регистр IX записать адрес соответствующего блока и окно появится на экране. Напишем подпрограмму формирования окна заданных размеров и черной «тени» под ним:

```
WINDOW INC   (IX)         ;смещаем окно вправо (COL)
       INC   (IX+1)       ; и вниз (ROW)
       LD    A,(IX+4)     ;ATTR
       PUSH  AF
       LD    (IX+4),1     ;байт атрибутов для «тени»
       CALL  SETW         ;изменяем только атрибуты
       DEC   (IX)         ;возвращаем окно на прежнее место
       DEC   (IX+1)
       POP   AF
       LD    (IX+4),A     ;восстанавливаем байт атрибутов
       CALL  CLSW         ;очищаем окно
       CALL  SETW         ;окрашиваем заданным цветом
       CALL  BOX          ;рисуем рамку вокруг окна
       RET
; Подпрограмма, рисующая рамку вокруг окна
BOX    LD    A,(IX+1)     ;ROW
       PUSH  AF
       CALL  3742         ;вычисляем адрес экрана
       LD    A,L
       ADD   A,(IX)       ;COL
       LD    L,A
       LD    B,(IX+2)     ;LEN
BOX1   LD    (HL),255     ;проводим верхнюю линию
       INC   HL
       DJNZ  BOX1
       LD    B,(IX+3)     ;HGT
       POP   AF
BOX2   PUSH  AF           ;по точкам рисуем боковые линии
                          ; сверху вниз
       PUSH  BC
       CALL  3742
       LD    A,L
       ADD   A,(IX)
       LD    L,A
       LD    B,8
BOX3   PUSH  HL
       LD    A,(HL)
       OR    128          ;левая точка
       LD    (HL),A
       LD    A,(IX+2)
       ADD   A,L
       DEC   A
       LD    L,A
       LD    A,(HL)
       OR    1            ;правая точка
       LD    (HL),A
       POP   HL
       INC   H
       DJNZ  BOX3
       POP   BC
       POP   AF
       INC   A
       DJNZ  BOX2
       DEC   H
       LD    B,(IX+2)     ;LEN
BOX4   LD    (HL),255     ;проводим нижнюю линию
       INC   HL
       DJNZ  BOX4
       RET
```

Начало программы традиционное - задание постоянных атрибутов экрана и окрашивание бордюра. Ну, а чтобы тексты выглядели более привлекательно, обратимся к уже известной процедуре утолщения символов - BOLD.

```
       ORG   60000
```

```
        ENT   $
        LD    A,38
        LD    (23693),A
        LD    A,4
        CALL  8859
        CALL  BOLD
```

Затем сформируем окно, в которое должно быть помещено название игры (рис. 6.9), а в нижней части заставки напечатаем принятый в таких случаях текст «Select options 0 to 4» («Выберите опции от 0 до 4»). Поскольку к метке MENU программа будет неоднократно возвращаться из кадров, то чтобы избежать хаотического наложения окон, целесообразно каждый раз очищать экран.

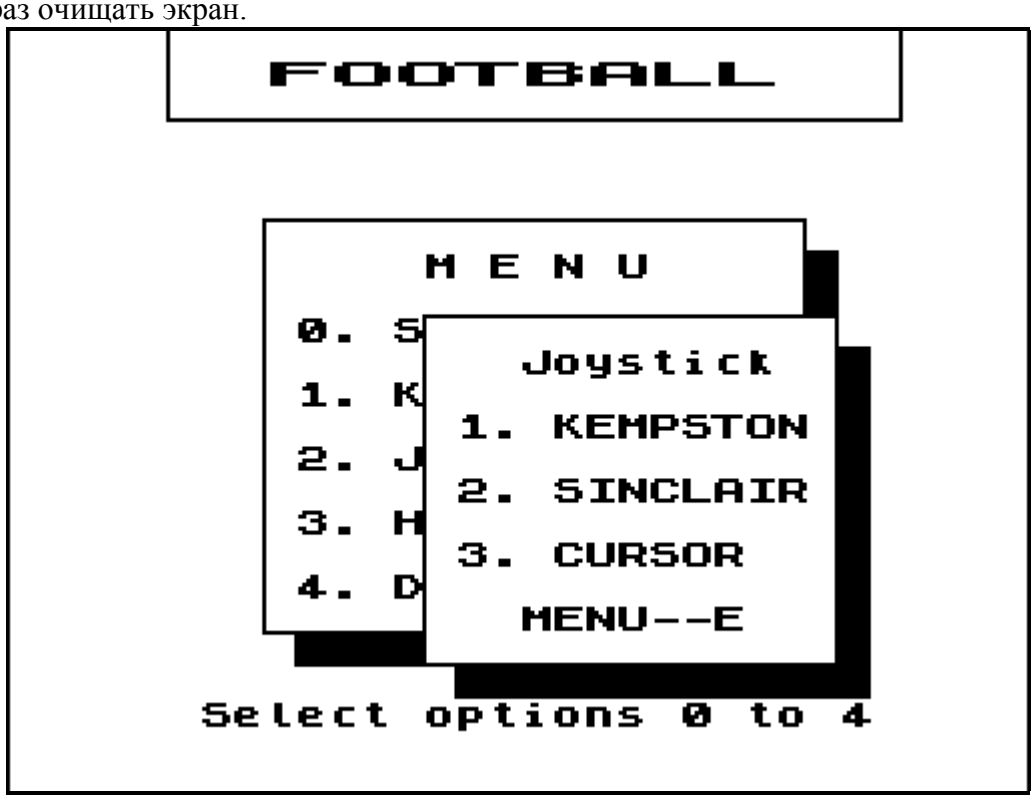


Рис. 6.9. Многокадровая заставка

```
MENU    CALL  3435
        LD    A,2
        CALL  5633
        LD    DE,TEXT5
        LD    BC,ENDTXT-TEXT5
        CALL  8252          ;«Select options 0 to 4»
        LD    IX,WIN0       ;окно для названия игры
        CALL  CLSW
        CALL  SETW
        CALL  BOX
```

В сформированном окне напечатаем название игры FOOTBALL, для чего используем расширенные подпрограммой DBLSYM буквы.

```
        LD    DE,COORD      ;позиционирование курсора
        LD    BC,TEXT-COORD
        CALL  8252
        LD    HL,TEXT       ;вывод названия игры
        LD    B,8
MET     LD    A,(HL)
        PUSH  HL
        PUSH  BC
        CALL  DBLSYM
```

```
        POP   BC
        POP   HL
        INC   HL
        DJNZ  MET
```

Сделаем для меню окно с черной тенью и поместим в него текст, в соответствии с которым вы можете обратиться к одному из четырех кадров, нажав клавиши **1-4**, или начать игру, выбрав клавишу **0** (для упрощения программы нажатие **0** или **4** возвращает вас в редактор GENS4 или в Бейсик).

```
        LD    IX,WIN1     ;окно основного меню
        CALL  WINDOW
        LD    DE,TEXT1    ;текст меню
        LD    BC,TEXT2-TEXT1
        CALL  8252
```

Теперь можно написать блок управления, который, как ни странно, выглядит довольно простым. Программка «крутится» в цикле, пока не нажата одна из указанных в кавычках клавиш. При нажатии же на клавишу, команда сравнения CP изменит биты флагового регистра (в частности флаг Z установится в ноль), после чего следующая команда JR Z,KADR? осуществит переход на выбранный вами кадр.

```
        XOR   A
        LD    (23560),A
CYCLE   LD    A,(23560)
        CP    "1"
        JR    Z,KADR1
        CP    "2"
        JR    Z,KADR2
        CP    "3"
        JR    Z,KADR3
        CP    "4"
        JR    Z,EXIT
        CP    "0"
        JR    NZ,CYCLE
; При нажатии на клавиши 0 или 4 восстанавливаются атрибуты экрана и стандартный
;  набор символов, после чего происходит выход в редактор GENS или в Бейсик.
EXIT    LD    A,7
        CALL  8859        ;белый бордюр
        LD    A,%00111000 ;стандартные атрибуты
        LD    (23693),A
        CALL  3435
        LD    A,2
        CALL  5633
        LD    HL,15360
        LD    (23606),HL
        RET
```

Части программы, формирующие окна кадров и печатающие в них тексты, исключительно похожи друг на друга. Тем не менее, имеются и некоторые отличия, которые определяются конкретными данными для окон и текстов. Каждый из кадров начинается со звукового сигнала SND, о котором мы говорили в начале этой главы и заканчивается процедурой WAIT, которая фиксирует кадр на экране, позволяя увидеть его содержание и выбрать дальнейшие действия (в этой программе чисто умозрительно, за исключением клавиши **E**, которая действительно возвращает вас в меню).

```
KADR1   CALL  SND         ;звуковой сигнал
        LD    IX,WIN2
        CALL  WINDOW      ;вывод окна
        LD    DE,TEXT2    ;текст в окне
        LD    BC,TEXT3-TEXT2
        CALL  8252
        CALL  WAIT        ;ожидание нажатия клавиши E
        JP    MENU        ;возврат в меню
```

```
; Формирование окна Кадра 2 и печать в нем текста
KADR2   CALL   SND
        LD     IX,WIN3
        CALL   WINDOW
        LD     DE,TEXT3
        LD     BC,TEXT4-TEXT3
        CALL   8252
        CALL   WAIT
        JP     MENU
; Формирование окна Кадра 3 и печать в нем текста
KADR3   CALL   SND
        LD     IX,WIN4
        CALL   WINDOW
        LD     DE,TEXT4
        LD     BC,TEXT5-TEXT4
        CALL   8252
        CALL   WAIT
        JP     MENU
; Подпрограмма ожидания нажатия клавиши E
WAIT    XOR    A
        LD     (23560),A
WAIT1   LD     A,(23560)
        CP     "E"
        RET    Z
        CP     "e"
        JR     NZ,WAIT1
        RET
SND     LD     B,10
        LD     HL,350
        LD     DE,4
SND1    .........
DBLSYM  .........
BOLD    .........
SETW    .........
CLSW    .........
WINDOW  .........
BOX     .........
; Данные для формирования окна с названием игры
WIN0    .........
; Данные для формирования всех окон с тенями
WIN1    .........
WIN2    .........
WIN3    .........
WIN4    .........
; Данные для печати названия игры
COORD   DEFB   22,1,8,16,6,19,1
TEXT    DEFM   "FOOTBALL"
; Данные для печати текста Меню
TEXT1   DEFB   22,7,13,16,7,17,2,19,1
        DEFM   "M E N U"
        DEFB   22,9,9
        DEFM   "0. START··GAME"
        DEFB   22,11,9
        DEFM   "1. KEYBOARD"
        DEFB   22,13,9
        DEFM   "2. JOYSTICK"
        DEFB   22,15,9
        DEFM   "3. HI··SCORE"
        DEFB   22,17,9
        DEFM   "4. DEMO MODE"
; Данные для печати текста Кадра 1
TEXT2   DEFB   22,6,14,17,3,16,1
        DEFM   "Define keys"
        DEFB   22,8,15
        DEFM   "LEFT....O"
```

```
       DEFB  22,10,15
       DEFM  "RIGHT...P"
       DEFB  22,12,15
       DEFM  "UP......Q"
       DEFB  22,14,15
       DEFM  "DOWN....A"
       DEFB  22,16,15
       DEFM  "FIRE....M"
       DEFB  22,18,16
       DEFM  "MENU--E"
; Данные для печати текста Кадра 2
TEXT3  DEFB  22,10,16,17,5,16,0
       DEFM  "Joystick"
       DEFB  22,12,14
       DEFM  "1. KEMPSTON"
       DEFB  22,14,14
       DEFM  "2. SINCLAIR"
       DEFB  22,16,14
       DEFM  "3. CURSOR"
       DEFB  22,18,16
       DEFM  "MENU--E"
; Данные для печати текста Кадра 3
TEXT4  DEFB  22,7,8,17,6,16,1
       DEFM  "Hi score"
       DEFB  22,8,7
       DEFM  "PETR...726"
       DEFB  22,10,7
       DEFM  "IGOR...694"
       DEFB  22,12,7
       DEFM  "ALEX...605"
       DEFB  22,14,7
       DEFM  "SERG...523"
       DEFB  22,16,7
       DEFM  "WLAD...419"
       DEFB  22,18,8
       DEFM  "MENU--E"
; Данные для текста под заставкой
TEXT5  DEFB  22,21,6,16,7,17,4,19,0
       DEFM  "Select options 0 to 4"
ENDTXT
```

ГЛАВА СЕДЬМАЯ,

в которой вы научитесь создавать все элементы игрового пространства

Как вы уже знаете, игровое пространство составляют перемещающиеся спрайты, которые появляются и исчезают на экране во время игры, и неподвижный или медленно перемещающийся пейзаж. В предыдущих главах мы частично показали, каким образом можно создавать спрайты, используя привычные символы UDG и средства ассемблера. Однако такой способ пригоден лишь для небольших изображений, да и то, если их общая площадь не превышает двух десятков знакомест. Теперь пора нам подробно познакомиться с общим случаем, когда размеры спрайтов могут быть практически любыми, а их количество ограничено лишь сюжетом игры, вашей фантазией и трудолюбием. Есть и другая, не менее важная задача - создание таких процедур вывода спрайтов на экран и рисования пейзажей, которые бы требовали на это минимального времени и, желательно, были не слишком громоздкими.

БЫСТРЫЙ ВЫВОД СПРАЙТОВ

Что бы ни появлялось на экране во время игры - спрайты или какие-либо тексты - каждое изображение состоит из отдельных символов. Таким образом, чтобы быстро выводить сложные картинки, нужно начать с самого простого - печати произвольного символа в текущую позицию экрана. Раньше мы поручали эту задачу процедуре RST 16, которая неплохо справлялась со своими обязанностями до тех пор, пока отдельные кадры изображения не слишком быстро сменяли друг друга. Безусловно, ее и дальше вполне можно использовать в подобных ситуациях. Однако, когда речь заходит о создании динамических картинок, а именно такие мы чаще всего наблюдаем после загрузки наиболее интересных игровых программ, она уже перестает нас удовлетворять. Изображения начинают временами пропадать и, конечно же, теряется естественное восприятие событий.

Во 2-й главе мы приводили пример небольшой программки на Бейсике, которая печатала букву **A**, и сказали, что по такому принципу работает любая процедура вывода символов на экран. Теперь перепишем ее на ассемблере и как основу используем для составления подпрограмм вывода спрайтов. До того, как этот фрагмент появится в программе, необходимо в регистровой паре DE задать адрес символа в наборе, в HL - рассчитанный начальный адрес знакоместа:

```
        .........
        LD    B,8
MET1    LD    A,(DE)
        LD    (HL),A
        INC   DE
        INC   H
        DJNZ  MET1
        .........
```

Ниже приводится процедура PRSYM, которая, так же как и RST 16, выводит на экран отдельные символы в текущее знакоместо экрана с учетом заданных атрибутов, но работает она приблизительно в 10 раз быстрее. Конечно же даром ничего не дается и увеличение быстродействия достигается за счет урезания выполняемых ею функций. Например, с ее помощью невозможно выводить тексты на принтер, печатать символы UDG и псевдографики, а также ключевые слова Бейсика. Не «воспринимает» она и управляющие коды. Тем не менее, PRSYM в тех или иных модификациях используется дальше в нескольких программах. Например, в одной из них показывается, как рисовать на экране лабиринты, различные орнаменты или рамки произвольной конфигурации.

Описание процедуры начнем с команды BIT, которая здесь встречается впервые и выполняет проверку состояния отдельных битов. Значение бита отражает флаг нуля - если бит установлен, выполняется условие NZ, в противном случае - Z (т. е. если проверяемый бит равен нулю, то Z=1). Команда имеет формат BIT n,s, где n - номер бита, задаваемый числом от 0 (младший) до 7 (старший), а s - операнд, которым может быть один из регистров общего назначения или (HL), (IX+d) и (IY+d). Попутно стоит сказать о еще двух командах этой группы: SET n,s - установка бита с номером n и RES n,s - сброс бита. В первом случае в бит n записывается единица, во втором - нуль. В этих командах операнды и формат такие же, как и в команде BIT, но флаги остаются без изменений.

```
PRSYM   PUSH  BC
        PUSH  DE
```

```
        PUSH  HL
        LD    L,A          ;по коду символа вычисляем его адрес
        LD    H,0          ; в текущем наборе
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        LD    DE,(23606)
        ADD   HL,DE
        LD    DE,(23684)   ;адрес текущей позиции печати
                           ; в видеобуфере
        EX    DE,HL
        PUSH  HL
        LD    B,8
PRS1    LD    A,(DE)
        BIT   3,(IY+87)    ;режим INVERSE 1
        JR    Z,PRS2
        CPL                ;если включен, инвертируем байт
PRS2    BIT   1,(IY+87)    ;режим OVER 1
        JR    Z,PRS3
        XOR   (HL)         ;если включен, объединяем
                           ; с изображением на экране
PRS3    LD    (HL),A
        INC   DE
        INC   H
        DJNZ  PRS1
        POP   HL
        PUSH  HL
        LD    A,H          ;вычисляем адрес в области атрибутов
        AND   #18
        RRCA
        RRCA
        RRCA
        ADD   A,#58
        LD    H,A
        LD    A,(23695)
        LD    (HL),A       ;записываем байт атрибутов в видеобуфер
        POP   HL
        INC   L            ;переходим к следующей позиции печати
        JR    NZ,PRS4      ;если 0, то это означает, что позиция
                           ; печати перешла в следующую треть экрана
        LD    A,H          ;в этом случае увеличиваем
        ADD   A,8          ; старший байт адреса на 8
        LD    H,A
        CP    #58
        JR    C,PRS4
        LD    H,#40        ;если выход из последней трети, то
                           ; возвращаемся в начало видеобуфера
PRS4    LD    (23684),HL
        POP   HL
        POP   DE
        POP   BC
        RET
```

Используем сначала эту процедуру для вывода буквенной или цифровой информации. Конечно, можно представить себе ситуацию, когда печатается всего один символ, например, уровень игры, но все же значительно чаще приходится иметь дело с текстовыми строками или даже целыми страницами и тут процедуры PRSYM явно недостаточно. Во-первых, необходимо уметь позиционировать курсор подобно тому, как это выполняет AT в Бейсике, во-вторых, желательно иметь возможность в любой момент переводить курсор на следующую строку, а в-третьих, - вспомним еще несколько атрибутов печати, которые применяются в команде RST 16: INK, PAPER, BRIGHT, FLASH, INVERSE и OVER. Для того чтобы вывод текстов на экран не вызывал особых проблем, необходима процедура, хорошо «понимающая» все управляющие коды, используемые при печати, и настраивающая в

соответствии с ними системные переменные, используемые подпрограммой PRSYM. Ниже приводится текст такой процедуры, которую мы назвали WRITE, с краткими комментариями к отдельным группам строк.

Прежде чем привести ее текст, объясним смысл вновь встретившейся здесь команды. Это инструкция EX (SP),HL, которая обменивает содержимое регистровой пары HL со значением, находящимся на вершине машинного стека: то, что было в HL, помещается на вершину стека, а два байта со стека перемещаются в HL. Значение регистра SP при этом не изменяется. Обратите внимание на то, как в подпрограмме WRITE2 осуществляется переход по выбранному адресу: число заносится в стек командой EX (SP),HL, а затем выполняется команда RET.

Другой момент, требующий пояснения, это процедура, расположенная в ПЗУ по адресу 8, благодаря чему ее можно вызывать командой RST. Она используется интерпретатором для выдачи различных сообщений и с ее помощью можно в любой момент остановить практически любую программу в кодах при возникновении критической ошибки. Код сообщения, уменьшенный на единицу (например, -1 для `0 OK`), записывается в директиве DEFB непосредственно за командой RST 8.

```
WRITE  LD    A,(HL)       ;берем очередной символ из строки
       INC   HL
       AND   A
       RET   Z            ;вывод символов до кода 0
       CP    " "
       JR    C,WRITE2     ;если управляющий код (< 32)
WRITE1 CALL  PRSYM
       JR    WRITE
; Выбор из таблицы адреса перехода в зависимости от кода
WRITE2 PUSH  HL
       PUSH  BC
       LD    HL,TABLE     ;адрес таблицы
       LD    C,A          ;сохраняем код в C
WRITE3 LD    A,(HL)       ;читаем код из таблицы
       INC   HL
       AND   A
       JR    Z,WRITE5     ;если встречен маркер конца таблицы
       CP    C
       JR    Z,WRITE4     ;если код найден
       INC   HL           ;пропускаем 2 байта адреса
       INC   HL
       JR    WRITE3       ;проверяем следующий код
WRITE4 POP   BC
       LD    A,(HL)       ;берем из таблицы адрес перехода
       INC   HL
       LD    H,(HL)       ;помещаем его в HL
       LD    L,A
       EX    (SP),HL      ;восстанавливаем HL, а адрес записываем
                          ; на вершину стека
       RET                ;переходим по адресу с вершины стека
WRITE5 POP   BC           ;если код отсутствует в таблице,
       POP   HL           ; то восстанавливаем регистры
       LD    A,"?"        ; и печатаем символ ?
       JR    WRITE1
; Перевод строки
PR_13  PUSH  HL
       LD    HL,(23684)   ;адрес текущей позиции печати
       LD    A,L
       AND   %11100000    ;возвращаемся к началу строки
       ADD   A,#20        ;переходим к следующей строке
       LD    L,A
```

```
       JR    NZ,PR13_1   ;если не перешли в следующую треть
       LD    A,H
       ADD   A,8         ;старший байт адреса увеличиваем на 8
       LD    H,A
       CP    #58         ;проверяем, был ли выход за пределы экрана
       JR    C,PR13_1
       LD    H,#40       ;переводим позицию печати в верхнюю
                         ; строку экрана
PR13_1 LD    (23684),HL  ;возвращаем рассчитанный адрес
                         ; позиции печати
       POP   HL
       JR    WRITE
; Цвет «чернил» INK
PR_16  LD    A,(HL)      ;читаем следующий байт из строки
       AND   %111        ;выделяем 3 младших бита (значения 0...7)
       PUSH  BC
       LD    B,%11111000 ;в регистре B - маска битов для INK
PR16_1 LD    C,A
       LD    A,(IY+85)   ;23695
       AND   B           ;освобождаем биты предыдущего атрибута
       OR    C           ; и записываем на их место новое значение
       LD    (IY+85),A   ;возвращаем байт атрибутов
PR16_2 POP   BC
       INC   HL
       JR    WRITE
; Цвет «бумаги» PAPER
PR_17  LD    A,(HL)      ;с цветом PAPER аналогично INK
       AND   %111
       RLCA              ; только предварительно сдвигаем
       RLCA              ; биты на свое место
       RLCA
       PUSH  BC
       LD    B,%11000111
       JR    PR16_1
; Мерцание FLASH
PR_18  LD    A,(HL)
       AND   1
       PUSH  BC
       LD    B,%01111111
PR18_1 RRCA
       JR    PR16_1
; Яркость BRIGHT
PR_19  LD    A,(HL)
       AND   1
       PUSH  BC
       LD    B,%10111111
       RRCA
       JR    PR18_1
; Инверсия INVERSE (3-й бит в системной переменной P_FLAG)
PR_20  LD    A,(HL)
       AND   1
       PUSH  BC
       LD    B,%11110111
       RLCA
       RLCA
PR20_1 RLCA
       LD    C,A
       LD    A,(IY+87)
       AND   B
       OR    C
       LD    (IY+87),A
       JR    PR16_2
; Режим наложения OVER (1-й бит в системной переменной P_FLAG)
PR_21  LD    A,(HL)
       AND   1
```

```
        PUSH  BC
        LD    B,%11111101
        JR    PR20_1
; Позиционирование курсора AT
PR_22   LD    A,(HL)        ;берем номер строки
        CP    24            ;если больше 24, то выход позиции
        JR    NC,OUTSCR     ; печати за пределы экрана
        INC   HL
        PUSH  DE
        PUSH  HL
        CALL  3742          ;вычисляем адрес начала строки
        POP   DE
        LD    A,(DE)        ;берем номер столбца
        CP    32            ;если больше 32, то выход позиции
        JR    NC,OUTSCR     ; печати за пределы экрана
        INC   DE
        ADD   A,L
        LD    L,A
        LD    (23684),HL    ;запоминаем адрес новой позиции
        POP   HL
        EX    DE,HL
        JP    WRITE
OUTSCR  RST   8             ;сообщение Бейсика
        DEFB  4             ; «Out of screen»
PRSYM   .........
; Таблица переходов на процедуры для управляющих кодов
TABLE   DEFB  13
        DEFW  PR_13
        DEFB  16
        DEFW  PR_16
        DEFB  17
        DEFW  PR_17
        DEFB  18
        DEFW  PR_18
        DEFB  19
        DEFW  PR_19
        DEFB  20
        DEFW  PR_20
        DEFB  21
        DEFW  PR_21
        DEFB  22
        DEFW  PR_22
        DEFB  0
```

Покончив с текстами, перейдем к описанию программы, которая выводит на экран спрайт произвольной конфигурации, но сначала стоит сказать несколько слов о том, что собой представляют спрайты с точки зрения программиста. Мы уже описывали работу со спрайтами и вам известно, что в принципе - это блоки данных, организованные определенным образом. Надо сказать, что существует множество различных форматов спрайтов. Например, формат спрайтов, принятый в Laser Basic отличается от того, который используется в Beta Basic, а тот, который хотим предложить мы, в свою очередь, не похож ни на первый, ни на второй, и все они отличаются от того, который мы продемонстрировали в предыдущих главах. Главным критерием в выборе формата блока данных является способ вывода спрайта на экран. Представляемый нами способ, быть может, не самый оптимальный в плане быстродействия, но зато программа имеет минимальные размеры при максимальном количестве возможностей. Так, например, спрайт может частично или даже полностью выходить за пределы экрана, а вывод может быть осуществлен по любому известному принципу объединения изображений (то есть с замещением либо по AND, OR или XOR). Для упрощения программы спрайт будет выводиться по символам, как и в описанной ранее процедуре PUT.

Начнем с разработки формата спрайтов, который зависит от способа вывода графики, а затем, привязываясь к формату, напишем соответствующую процедуру вывода спрайтов на экран (строго говоря, разработка формата спрайтов и процедуры их вывода должны протекать параллельно, так как одно от другого неотделимо).

Блок данных, описывающий каждый спрайт, будет состоять из двух частей: заголовка, включающего в себя относительные координаты и атрибуты для каждого знакоместа (эта часть будет напоминать формат спрайтов для процедуры PUT), и данных о состоянии пикселей (по 8 байт на знакоместо - как в символьном наборе). Заголовок будет начинаться указанием общей площади спрайта, или иначе - количества знакомест, составляющих спрайт. Для этого достаточно одного байта, что позволит создавать спрайты площадью до 255 знакомест. Затем для описания каждого символа потребуется по 3 байта, как и в процедуре PUT:

1-й байт - относительная вертикальная координата данного знакоместа в спрайте;
2-й байт - относительная горизонтальная координата данного знакоместа в спрайте;
3-й байт - суммарные атрибуты знакоместа.

Таким образом, можно составить примерно такой заголовок:

```
SPRITE DEFB  7
       DEFB  0,2,15
       DEFB  1,0,6, 1,1,6, 1,2,6, 1,3,6
       DEFB  2,1,6, 2,2,6
```

Вторую часть блока данных составляют уже знакомые вам описания пикселей знакомест (по 8 байт на каждое), причем они должны быть перечислены в том порядке, в котором указаны в заголовке, например (всего должно быть 7 строк - по количеству символов, входящих в спрайт):

```
       DEFB  33,39,62,255,0,127,127,127
       DEFB  246,73,146,255,0,11,222,251
       ..............................
       DEFB  35,216,225,228,3,16,148,35
```

Теперь разберемся с числовыми параметрами, необходимыми для вывода спрайта. Перед обращением к процедуре вывода, которую мы назвали PTBL, в регистре B нужно задать верхнюю границу описывающего прямоугольника (ROW), в C - левую границу описывающего прямоугольника (COL), в HL - адрес блока данных спрайта (метка SPRITE), а в аккумуляторе - код команды, определяющей способ вывода спрайтов, который в самом начале работы процедуры будет вставлен в основной цикл вывода и тем самым вместо составления четырех похожих друг на друга подпрограмм можно пользоваться одной. Для обычного вывода с замещением предыдущего изображения нужно задать команду NOP - отсутствие операции, которая кодируется байтом 0; для осуществления вывода по принципу OR, AND или XOR необходимо вставить в процедуру команды OR (HL), AND (HL) или XOR (HL), имеющие коды #B6, #A6 и #AE соответственно. Чтобы не держать в голове все эти коды, имеет смысл определить их как константы с помощью директивы EQU и присвоить им удобочитаемые имена:

SPRPUT - вывод с уничтожением предыдущего изображения;
SPROR - вывод по принципу OR;
SPRAND - вывод по принципу AND;
SPRXOR - вывод по принципу XOR.

В данном примере вновь применена команда EX (SP),HL, но здесь вершину стека можно рассматривать в качестве временной переменной, и такой прием может быть рассмотрен как один из способов борьбы с нехваткой регистров.

```
PTBL
SPRPUT EQU   0           ;код команды NOP
SPROR  EQU   #B6         ;код команды OR (HL)
```

```
SPRAND EQU   #A6          ;код команды AND (HL)
SPRXOR EQU   #AE          ;код команды XOR (HL)
       PUSH  HL
       LD    (MODE),A     ;устанавливаем способ объединения
                          ; изображений
       LD    A,(HL)       ;количество знакомест в спрайте
       INC   HL
       PUSH  HL           ;умножаем на 3 (результат в HL)
       LD    L,A
       LD    H,0
       LD    E,L
       LD    D,H
       ADD   HL,HL
       ADD   HL,DE
       POP   DE
       ADD   HL,DE        ;начало данных, описывающих пиксели
       EX    DE,HL
PTBL1  PUSH  AF
       PUSH  BC
       LD    A,(HL)       ;вертикальная координата в спрайте
       INC   HL
       PUSH  HL
       ADD   A,B
       CP    24
       JR    NC,PTBL4     ;если знакоместо выходит за пределы экрана
       PUSH  DE
       CALL  3742         ;получаем адрес строки экрана
       POP   DE
       EX    (SP),HL
       LD    A,(HL)       ;горизонтальная координата в спрайте
       EX    (SP),HL
       ADD   A,C
       CP    32
       JR    NC,PTBL4     ;если знакоместо выходит за пределы экрана
       ADD   A,L
       LD    L,A
       LD    B,8
       PUSH  HL           ;сохраняем адрес экрана
PTBL2  LD    A,(DE)
MODE   NOP
       LD    (HL),A
       INC   DE
       INC   H
       DJNZ  PTBL2
       POP   BC           ;восстанавливаем адрес экрана в BC
       LD    A,B          ;определяем адрес атрибутов
       AND   #18
       SRA   A
       SRA   A
       SRA   A
       ADD   A,#58
       LD    B,A
       POP   HL           ;восстанавливаем адрес данных
       INC   HL
       LD    A,(HL)       ;переносим байт атрибутов в видеобуфер
       DEC   HL
       LD    (BC),A
PTBL3  POP   BC
       POP   AF
       INC   HL
       INC   HL
       DEC   A
       JR    NZ,PTBL1
       POP   HL
       RET
```

```
PTBL4   LD      HL,8
        ADD     HL,DE
        EX      DE,HL
        POP     HL
        JR      PTBL3
```

Рис. 7.1. Спрайт из игры FIST

Рассмотрим пример вывода спрайта произвольной конфигурации (рис. 7.1). Не правда ли, многие узнали в нем одного из персонажей игры FIST. Все дело в том, что этот спрайт как нельзя лучше демонстрирует эффективность предложенной нами процедуры PTBL, поскольку его форма заметно отличается от прямоугольной. Управляющая часть программы получилась довольно короткой:

```
        ORG     60000
        ENT     $
        LD      A,48            ;INK 0: PAPER 6
        LD      (23693),A
        XOR     A               ;BORDER 0
        CALL    8859
        CALL    3435
        LD      A,2
        CALL    5633
        LD      B,10            ;ROW
        LD      C,15            ;COL
        LD      A,SPRPUT        ;вывод с уничтожением предыдущего
                                ; изображения
        LD      HL,SPR1         ;чтение адреса спрайта SPR1
        CALL    PTBL
        RET
PTBL    .........
; Заголовок спрайта
SPR1    DEFB    22
        DEFB    0,3,48, 0,4,48, 1,3,48, 1,4,48, 2,2,48
        DEFB    2,3,48, 2,4,48, 2,5,48, 2,6,48, 3,2,48
        DEFB    3,3,48, 3,4,48, 4,0,48, 4,1,48, 4,2,48
        DEFB    4,3,48, 4,4,48, 5,0,48, 5,1,48, 5,2,48
        DEFB    5,3,48, 5,4,48
; Данные спрайта
        DEFB    0,0,0,0,3,12,209,46
        DEFB    0,0,0,0,192,32,224,240
        DEFB    249,34,38,43,43,48,35,248
        DEFB    16,112,56,8,16,48,16,240
        DEFB    3,6,31,61,61,59,59,27
        DEFB    254,255,255,255,255,255,255,255
        DEFB    64,224,255,255,255,255,255,255
        DEFB    0,0,255,192,224,243,252,240
        DEFB    0,248,4,2,2,250,50,28
        DEFB    31,7,5,6,7,15,15,31
        DEFB    255,255,254,229,3,247,247,235
        DEFB    240,0,0,0,252,254,254,254
        DEFB    0,0,0,0,0,62,103,77
        DEFB    0,0,0,0,0,0,249,255
        DEFB    31,31,63,63,127,255,255,255
        DEFB    235,219,219,219,252,224,193,131
        DEFB    254,254,254,126,254,252,252,248
```

```
DEFB  65,67,71,69,34,30,0,0
DEFB  255,255,255,255,63,7,0,0
DEFB  255,254,252,248,240,224,0,0
DEFB  3,3,1,2,2,1,0,0
DEFB  248,248,152,198,1,255,0,0
```

Есть смысл немного прокомментировать приведенные выше числовые данные, которые полностью соответствуют описанному выше формату. В заголовке перечислены (например, для первой строки):

22 - общее количество знакомест в спрайте,
0 - координата Y первого выводимого на экран знакоместа, взятая относительно левого верхнего угла описывающего прямоугольника,
3 - координата X того же знакоместа,
48 - суммарные атрибуты знакоместа: PAPER 6, INK 0.

Далее идут тройки чисел, относящиеся к другим знакоместам спрайта и в последовательности, указанной выше: координата Y, координата X, суммарные атрибуты.

СПРАЙТ-ГЕНЕРАТОР

Можно по разному создавать блоки данных для спрайтов, начиная с самого простого способа, когда изображение сначала рисуется на бумаге, а затем выписываются его коды байт за байтом. Можно воспользоваться приведенной нами в четвертой главе программой, для которой сначала создается фонт (например, в Art Studio), соответствующий одному или сразу нескольким спрайтам, после чего коды все равно требуется записать и только затем уж вводить в программу. Оба варианта требуют затрат большого труда и времени и оправдывают себя лишь в случаях небольших спрайтов (порядка 1 - 6 знакомест). Учитывая все это, мы сочли необходимым предложить программу, которая полностью исключает какие-либо записи, а формируемые ею кодовые блоки можно сразу встраивать в создаваемые вами игры.

Программа состоит из двух частей - бейсиковской и кодовой. Если вы работаете с магнитофоном, то программу на Бейсике можно исполнять сразу, если же с дисководом, то три строки текста следует заменить (какие именно, сказано ниже). Затем введите и оттранслируйте ассемблерную часть, создав соответствующий кодовый файл. Теперь можно работать со спрайтами. После ввода и старта программы на вашем экране появится меню. Если вы предварительно просмотрите текст программы, то легко обнаружите, какие функции она может выполнять, тем не менее коротко прокомментируем эти опции.

Load Screen - загрузка экранного файла
Create Sprite - создание спрайта
Save Sprite - сохранение спрайт-файла
New Sprite - удаление спрайтов из памяти для начала создания нового спрайт-файла
View Table - просмотр таблицы смещений спрайтов в спрайт-файле
Quit Program - выход из программы

Нажимая клавиши **Q** и **A**, можно перемещать курсор в виде инвертированной полоски вверх или вниз по строчкам меню. Отметив курсором нужный пункт, нажмите клавишу **M** для выполнения функции.

Прежде всего необходимо загрузить экранный файл, для чего предназначен первый пункт меню Load Screen. Внизу экрана появится запрос **Screen name:**, на который нужно ввести имя загружаемой картинки со спрайтами. После загрузки экранного файла программа снова выйдет в меню.

После этого можно «вырезать» с картинки спрайты, выбрав следующий пункт Create Sprite. Окно с меню исчезнет с экрана и останется только загруженная картинка и маленький пунктирный квадратик. С помощью клавиш **Q**, **A**, **O** и **P** поместите его в верхний левый угол выбранного спрайта и нажмите клавишу **M**, чтобы зафиксировать местоположение квадратика на экране. Затем, управляя теми же клавишами, расширьте его до нужных размеров, чтобы спрайт полностью поместился внутри отмеченной пунктиром области и еще раз нажмите клавишу **M**. Возврат в меню покажет, что спрайт успешно закодирован - можно создавать следующий. Если создано уже достаточно много спрайтов и все они имеют значительные размеры, то памяти может не хватить. В этом случае программа выдаст сообщение **Out of memory!** Вы можете сохранить полученный спрайт-файл, вызвав опцию Save Sprite, и начать создание следующего, предварительно очистив память, выбрав пункт New Sprite.

Перед сохранением спрайт-файла нужно будет ввести его имя, под которым он будет записан на внешний носитель, а перед удалением спрайтов из памяти потребуется подтвердить свое намерение, нажав клавишу **Y**.

Последняя опция Quit Program в особых комментариях не нуждается, поэтому скажем только, что во избежание случайного выхода (а следовательно, и потери данных) нужно будет также подтвердить или опровергнуть выбор.

Сообщим еще общие «эксплуатационные» характеристики спрайт-генератора: каждый вновь создаваемый спрайт может занимать площадь до 255 знакомест, если вам захочется чуть больше - описывающий прямоугольник все равно не позволит, сколько бы ни старались. Максимальное количество спрайтов для одного спрайт-файла - 22, после чего его необходимо сохранить. И последнее, создаются спрайты только прямоугольной формы.

Для того чтобы вам легче было разобраться в этой сервисной программе, приведем расшифровку используемых обозначений, а также дадим краткое описание ее основной части - функции создания спрайта:

Массивы:

`m$(6,13)` - наименования опций меню
`s(22)` - смещения спрайтов относительно начала спрайт-файла

Переменные:

`spr` - количество созданных спрайтов
`addr` - адрес следующего спрайта
`col`, `row` - координаты окон и спрайтов (переменная row используется также для определения позиции курсора меню)
`len`, `hgt` - размеры окон и спрайтов
`pap` - цвет `PAPER` окон
`k$` - символ нажатой клавиши

Константы:

`scr` - адрес «теневого» экрана
`ad0` - адрес начала спрайт-файла
`ramka`, `svscr`, `restor`, `clsv`, `setv`, `gtbl` - адреса одноименных процедур

Опишем «центральную» подпрограмму ГЕНЕРАТОРА СПРАЙТОВ - подпрограмму создания спрайтов:

2010 - если создано 22 спрайта, сообщение о том, что спрайт-файл завершен. Необходимо его сохранить и начать новый.
2020 - вывод экранной картинки.
2030 - определение начальных значений переменных «вырезаемого» спрайта.

2040 - вывод по заданному размеру и в заданном месте экрана пунктирной рамки, отмечающей будущий спрайт.
2045..2090 - установка рамки в верхний левый угол «вырезаемого» спрайта.
2100 - удаление рамки и звуковой сигнал после нажатия клавиши **M**.
2110 - вывод рамки.
2130..2170 - выбор желаемого размера спрайта.
2200 - кодирование спрайта.
2210 - если процедура `gtbl` возвращает ненулевое значение, то рассчитывается величина смещения спрайта от начала спрайт-файла, а переменная `addr` указывает на конец спрайт-файла.
2220..2240 - выдается сообщение о нехватке памяти для создания спрайта заданных размеров. Можно сохранить спрайт-файл и начать новый или попытаться создать спрайт меньших размеров.

```
 10 POKE 23693,40: BORDER 5: CLS
 20 DIM m$(6,13): FOR n=1 TO 6: READ m$(n): NEXT n
 30 DIM s(22): LET spr=0
 40 LET scr=30000: LET ad0=36912: LET addr=ad0
 50 LET ramka=65000: LET svscr=65003: LET restor=65006:
    LET clsv=65009: LET setv=65012: LET gtbl=65015
 60 RANDOMIZE USR svscr
100 REM --- МЕНЮ ---
110 RANDOMIZE USR restor: LET row=6: LET col=8: LET len=15:
    LET hgt=13: LET pap=7: GO SUB 8000
120 LET row=0
130 IF row<0 THEN LET row=5
135 IF row>5 THEN LET row=0
140 FOR n=0 TO 5: PRINT PAPER 7; BRIGHT 1;AT 7+n*2,9;m$(n+1):
    NEXT n
150 PRINT INVERSE 1; BRIGHT 1;AT 7+row*2,9;m$(row+1)
160 PAUSE 0: LET k$=INKEY$
170 IF k$="a" OR k$="A" THEN BEEP .01,20: LET row=row+1:
    GO TO 130
180 IF k$="q" OR k$="Q" THEN BEEP .01,20: LET row=row-1:
    GO TO 130
190 IF k$<>"m" AND k$<>"M" THEN GO TO 150
200 BEEP .01,20: GO SUB (row+1)*1000
210 GO TO 100
1000 REM --- ЗАГРУЗКА ЭКРАННОЙ КАРТИНКИ ---
1010 INPUT "Screen name: "; LINE n$
1020 LOAD n$CODE 16384,6912
1030 RANDOMIZE USR svscr: RETURN
2000 REM --- СОЗДАНИЕ СПРАЙТОВ ---
2010 IF spr=22 THEN LET row=11: LET col=4: LET len=23: LET hgt=3:
     LET pap=3: GO SUB 8000: PRINT AT row+1,col+1; PAPER pap;
     BRIGHT 1;"Sprite-file complete!": BEEP 1,-20: PAUSE 0: RETURN
2020 RANDOMIZE USR restor
2030 LET spr=spr+1: LET row=12: LET col=15: LET len=1: LET hgt=1:
     POKE 23303,len: POKE 23304,hgt
2040 POKE 23301,col: POKE 23302,row: RANDOMIZE USR ramka
2045 PAUSE 0: LET k$=INKEY$
2050 IF (k$="q" OR k$="Q") AND row>0 THEN
     RANDOMIZE USR ramka: LET row=row-1: GO TO 2040
2060 IF (k$="a" OR k$="A") AND row<24-hgt THEN
     RANDOMIZE USR ramka: LET row=row+1: GO TO 2040
2070 IF (k$="o" OR k$="O") AND col>0 THEN
     RANDOMIZE USR ramka: LET col=col-1: GO TO 2040
2080 IF (k$="p" OR k$="P") AND col<32-len THEN
     RANDOMIZE USR ramka: LET col=col+1: GO TO 2040
2090 IF k$<>"m" AND k$<>"M" THEN GO TO 2045
2100 RANDOMIZE USR ramka: BEEP .01,20
2110 POKE 23303,len: POKE 23304,hgt: RANDOMIZE USR ramka
```

```
2120 PAUSE 0: LET k$=INKEY$
2130 IF (k$="a" OR k$="A") AND hgt<24-row AND len*(hgt+1)<256
     THEN RANDOMIZE USR ramka: LET hgt=hgt+1: GO TO 2110
2140 IF (k$="q" OR k$="Q") AND hgt>1 THEN
RANDOMIZE USR ramka: LET hgt=hgt-1: GO TO 2110
2150 IF (k$="o" OR k$="O") AND len>1 THEN RANDOMIZE USR ramka:
     LET len=len-1: GO TO 2110
2160 IF (k$="p" OR k$="P") AND len<32-col AND (len+1)*hgt<256
     THEN RANDOMIZE USR ramka: LET len=len+1: GO TO 2110
2170 IF k$<>"m" AND k$<>"M" THEN GO TO 2120
2200 BEEP .01,20: POKE 23300,hgt*len: RANDOMIZE addr:
     LET ad=USR gtbl
2210 IF ad THEN LET s(spr)=addr-ad0: LET addr=ad: RETURN
2220 LET col=6: LET row=11: LET hgt=3: LET len=20: LET pap=2
2230 GO SUB 8000: PRINT AT row+1,col+3; PAPER pap;
     BRIGHT 1;"Out of memory!"
2240 LET spr=spr-1: BEEP 1,-20: PAUSE 0: RETURN
3000 REM --- СОХРАНЕНИЕ СПРАЙТ-ФАЙЛА ---
3010 IF NOT spr THEN RETURN
3020 INPUT "Sprite name: "; LINE n$
3030 SAVE n$CODE ad0,addr-ad0
3040 RETURN
4000 REM --- УДАЛЕНИЕ СПРАЙТ-ФАЙЛА ИЗ ПАМЯТИ ---
4010 IF NOT spr THEN RETURN
4020 GO SUB 7000: IF k$<>"y" THEN RETURN
4030 LET spr=0: LET addr=ad0: RETURN
5000 REM --- ПРОСМОТР ТАБЛИЦЫ СМЕЩЕНИЙ СПРЙТОВ ---
5010 CLS : IF NOT spr THEN RETURN
5020 FOR n=1 TO spr: PRINT "Sprite No ";n,"Offset == ";s(n): NEXT n
5030 PAUSE 0: RETURN
6000 REM --- ВЫХОД ИЗ ПРОГРАММЫ ---
6010 GO SUB 7000: IF k$<>"y" THEN RETURN
6020 CLEAR : STOP
7000 REM --- ЗАПРОС ---
7010 LET row=11: LET col=5: LET len=21: LET hgt=3: LET pap=6:
     GO SUB 8000
7020 PRINT AT row+1,col+1; PAPER pap; BRIGHT 1;
     "Are you shure (Y/N)?"
7030 PAUSE 0: LET k$=INKEY$: IF k$="Y" THEN LET k$="y"
7040 BEEP .01,20: RETURN
8000 REM --- ОКНА ---
8010 POKE 23301,col: POKE 23302,row: POKE 23303,len: POKE 23304,hgt
8020 RANDOMIZE USR clsv: PRINT PAPER pap; BRIGHT 1;:
     RANDOMIZE USR setv
8030 LET l=len*8-1: LET h=hgt*8-1
8040 PLOT col*8,175-row*8: DRAW l,0: DRAW 0,-h: DRAW -l,0: DRAW 0,h
8050 RETURN
9000 REM --- ДАННЫЕ МЕНЮ ---
9010 DATA "Load Screen"
9020 DATA "Create Sprite"
9030 DATA "Save Sprite"
9040 DATA "New Sprite"
9050 DATA "View Table"
9060 DATA "Quit Program"
9900 REM --- АВТОСТАРТ ПРОГРАММЫ ---
9910 POKE 23693,40: BORDER 5: CLEAR 29999
9920 LOAD "sptgen"CODE
9930 RUN
```

Если вы работаете в системе TR-DOS, то несколько строк этой программы следует заменить на приведенные ниже:

```
1020 RANDOMIZE USR 15619: REM : LOAD n$CODE 16384,6912
3030 RANDOMIZE USR 15619: REM : SAVE n$CODE ad0,addr-ad0
9920 RANDOMIZE USR 15619: REM : LOAD "sptgen"CODE
```

и только после этого использовать.

Некоторые процедуры, как вы заметили, написаны на ассемблере и вызываются функцией USR. При использовании ряда подпрограмм в машинных кодах из Бейсика возникает проблема, как определить адреса обращения к ним. Можно, конечно, оттранслировать каждую из них отдельно, задав для каждой определенный начальный адрес или в одном исходном файле указать несколько директив ORG. Но при этом возникнут другие сложности, связанные с компоновкой программы. Можно также, оттранслировав весь пакет процедур как единое целое, просмотреть затем полученные коды с помощью дизассемблера и найти точки входа в каждую подпрограмму. Но при этом, если потребуется внести в текст какие-либо изменения (а особенно часто это придется делать на этапе отладки), то всю работу по определению адресов придется повторять с начала. В связи с этим мы предлагаем вам наиболее простой способ, часто применяемый в подобных ситуациях: в начале ассемблерного текста нужно вставить ряд команд JP, передающих управление всем процедурам пакета, к которым имеется обращение из Бейсика (либо из другого языка). Зная, что команда JP в памяти занимает 3 байта, несложно вычислить адрес любой процедуры по ее «порядковому номеру». Впоследствии мы еще не раз воспользуемся этим методом, поэтому мы и обратили на него ваше внимание.

Основная часть пакета - это подпрограмма GTBL, сохраняющая в памяти образ экрана в принятом для процедуры PTBL формате спрайтов. Подпрограмма OUT_BT также относится к ней. При вызове GTBL в переменной __SP сохраняется начальное состояние указателя стека SP. Делается это для корректного выхода в Бейсик в случае возникновения ошибки (Out of memory - нехватка памяти).

Подпрограмма RAMKA выводит на экран пунктирный прямоугольник, отмечающий границы создаваемого спрайта. Вывод производится по принципу XOR, поэтому при повторном обращении к процедуре прежний вид экрана полностью восстанавливается.

Подпрограмма SVSCR нужна для сохранения экранного изображения в памяти для последующего его восстановления процедурой RESTOR.

В пакет включены также две описанные ранее процедуры CLSV и SETV для очистки окна экрана и установки в нем постоянных атрибутов.

```
       ORG   65000
ORIGIN EQU   $           ;верхняя допустимая граница спрайт-файла
ADDR   EQU   23670       ;текущий адрес в спрайт-файле
ATTR   EQU   23695       ;значение атрибутов окна
SCREEN EQU   30000       ;адрес «теневого» экрана
N_SYM  EQU   23300       ;рассчитанная в Бейсике площадь спрайта
COL    EQU   23301       ;координаты спрайта
ROW    EQU   23302
LEN    EQU   23303       ;размеры спрайта
HGT    EQU   23304
; 65000
       JP    RAMKA
; 65003
       JP    SVSCR
; 65006
       JP    RESTOR
; 65009
       JP    CLSV
; 65012
       JP    SETV
; 65015
```

```
GTBL   CALL  RESTOR      ;восстанавливаем экранную картинку
       LD    (__SP),SP   ;запоминаем состояние стека для
                         ; возврата при возникновении ошибки
       LD    IX,(ADDR)   ;адрес конца спрайт-файла
; Формирование заголовка
       LD    A,(N_SYM)   ;количество знакомест
                         ; в создаваемом спрайте
       CALL  OUT_BT      ;записываем первый байт в спрайт-файл
       LD    A,(ROW)     ;вычисляем адрес атрибутов
       LD    L,A
       LD    H,0
       ADD   HL,HL
       ADD   HL,HL
       ADD   HL,HL
       ADD   HL,HL
       ADD   HL,HL
       LD    A,#58
       ADD   A,H
       LD    H,A
       LD    A,(COL)
       ADD   A,L
       LD    L,A
       LD    DE,(LEN)
       LD    BC,0
GTBL1  PUSH  BC
       PUSH  DE
       PUSH  HL
GTBL2  LD    A,B
       CALL  OUT_BT      ;позиция по вертикали внутри спрайта
       LD    A,C
       CALL  OUT_BT      ;позиция по горизонтали
       LD    A,(HL)
       CALL  OUT_BT      ;байт атрибутов
       INC   HL
       INC   C
       DEC   E
       JR    NZ,GTBL2
       POP   HL
       LD    DE,32       ;переходим к следующей строке
       ADD   HL,DE
       POP   DE
       POP   BC
       INC   B
       DEC   D
       JR    NZ,GTBL1
; Данные состояния пикселей
       LD    A,(HGT)
       LD    B,A
       LD    A,(ROW)
GTBL3  PUSH  AF
       PUSH  BC
       CALL  3742        ;вычисляем адрес начального знакоместа
       LD    A,(COL)
       ADD   A,L
       LD    L,A
       LD    A,(LEN)
       LD    B,A
GTBL4  PUSH  BC
       PUSH  HL
       LD    B,8         ;переписываем в спрайт-файл 8 байт
                         ; знакоместа
GTBL5  LD    A,(HL)
       CALL  OUT_BT
       INC   H
       DJNZ  GTBL5
```

```
        POP   HL
        INC   HL          ;переходим к следующему знакоместу
        POP   BC
        DJNZ  GTBL4
        POP   BC
        POP   AF
        INC   A           ;переходим к следующей строке
        DJNZ  GTBL3
        PUSH  IX          ;возвращаем в Бейсик адрес
        POP   BC          ; конца спрайт-файла
        RET
OUT_BT  PUSH  BC          ;запись в спрайт-файл байта из A
        PUSH  HL
; Проверка наличия свободной памяти
        PUSH  IX
        POP   HL
        LD    BC,ORIGIN   ;адрес конца свободной памяти
                          ; для спрайт-файла
        AND   A           ;очистка флага CY перед вычитанием
                          ; (если этого не сделать, результат будет неверен!)
        SBC   HL,BC       ;если текущий адрес достиг ORIGIN,
        JR    NC,OUTRAM   ; происходит выход в Бейсик
        POP   HL
        POP   BC
        LD    (IX),A      ;записываем байт в спрайт-файл
        INC   IX          ;увеличиваем адрес размещения кодов
        RET
OUTRAM  LD    SP,(__SP)   ;восстанавливаем значение стека
        LD    BC,0        ;возвращаем в Бейсик код ошибки
        RET
__SP    DEFW  0           ;переменная для сохранения указателя стека
; Рисование прямоугольной пунктирной рамки
RAMKA   LD    A,(ROW)
        PUSH  AF
        CALL  3742        ;вычисляем адрес экрана
        LD    A,(COL)
        ADD   A,L
        LD    L,A
        CALL  HOR         ;проводим верхнюю линию
        CALL  VERT1       ;рисуем боковые стороны в первой
                          ; строке окна
        LD    A,(HGT)
        DEC   A
        JR    Z,RAMK2     ;обходим, если единственная строка
        LD    B,A         ;иначе рисуем боковые стороны по всей
                          ; высоте окна
        POP   AF
RAMK1   PUSH  AF
        CALL  VERT        ;заканчиваем предыдущую строку
        POP   AF
        INC   A           ;переходим к следующей
        PUSH  AF
        CALL  3742        ;вычисляем адрес экрана
        LD    A,(COL)
        ADD   A,L
        LD    L,A
        CALL  VERT        ;ставим верхние точки
        CALL  VERT1       ;заканчиваем вертикальный пунктир
        POP   AF
        DJNZ  RAMK1       ;повторяем
        PUSH  AF
RAMK2   POP   AF
; Горизонтальная пунктирная линия
HOR     PUSH  BC
        PUSH  HL
```

```
        LD    A,(LEN)      ;рисуем пунктир по ширине окна
        LD    B,A
HOR1    LD    A,%10011001 ;фактура пунктирной линии
        XOR   (HL)         ;объединяем с экранным изображением
        LD    (HL),A       ;возвращаем на экран
        INC   HL
        DJNZ  HOR1
        POP   HL
        POP   BC
        RET
; Рисование двух точек для боковых сторон рамки
VERT    PUSH  HL
        LD    A,128        ;левая точка
        XOR   (HL)
        LD    (HL),A
        LD    A,(LEN)      ;ищем адрес правой стороны окна
        DEC   A
        ADD   A,L
        LD    L,A
        LD    A,1          ;правая точка
        XOR   (HL)
        LD    (HL),A
        POP   HL
        RET
; Боковые стороны рамки по высоте знакоместа
VERT1   INC   H            ;пропускаем 3 ряда пикселей
        INC   H
        INC   H
        CALL  VERT         ;ставим точки на левой и правой
                           ; сторонах прямоугольника
        INC   H
        CALL  VERT         ;повторяем для следующего ряда
        INC   H            ;делаем следующий промежуток
        INC   H
        INC   H
        RET
; Сохранение области видеобуфера в «теневом» экране
SVSCR   LD    HL,16384
        LD    DE,SCREEN
        LD    BC,6912
        LDIR
        RET
; Восстановление изображения на экране
RESTOR  LD    HL,SCREEN
        LD    DE,16384
        LD    BC,6912
        LDIR
        RET
; Подпрограмма очистки окна
CLSV    .........
; Подпрограмма установки атрибутов в окне
SETV    .........
```

МУЛЬТИПЛИКАЦИЯ

В большинстве компьютерных игр персонажи беспрерывно передвигаются по экрану, создавая неповторимые ситуации, чем собственно и привлекают к себе внимание многочисленной армии почитателей ZX Spectrum. В пятой главе мы уже слегка затронули проблему движения изображений, прояснив с помощью нескольких примеров те принципы,

которые лежат в основе любого перемещения по экрану, будь то тексты или спрайты. Теперь настало время внести в этот вопрос полную ясность, показав способы, наиболее часто используемые в игровых программах.

У вас может возникнуть вопрос, а зачем, собственно, рассматривать несколько разных способов, не проще ли ограничиться одним, только очень хорошим, и применять его во всех случаях компьютерной «жизни». Все дело в том, что возможны совершенно отличные друг от друга ситуации, определяемые конкретным замыслом. Скажем, для простых сюжетов нет смысла использовать сложные способы передвижения спрайтов, а в насыщенных играх простые способы уже могут оказаться неэффективными.

Первый способ основан на скроллинге окон, и если вы припомните программу, которая раздвигает в разные стороны створки железной решетки, то получите о нем полное представление. Тем не менее, мы вновь к нему возвращаемся, поскольку в нашем распоряжении появилась удобная процедура для вывода спрайтов. Перечислим действия, характерные для этого способа:

- поместите на экран спрайт, например, с помощью процедуры PTBL, установив режим SPRPUT;
- задайте окно, по размерам покрывающее этот спрайт и выполните скроллинг при помощи одной из четырех рассмотренных выше процедур. Спрайт довольно резво устремится в нужную сторону, так что в программе необходимо предусмотреть небольшую задержку.

Рис. 7.2. Перемещение спрайта скроллингом окна

Сразу же виден и недостаток этого способа, который состоит в том, что спрайт перемещается по экрану вместе с фоном, поскольку скроллинг захватывает все изображение в окне. Отсюда ясно, что применять такой метод можно лишь в тех случаях, когда фон как таковой отсутствует или, по крайней мере, мелкие детали не попадают в сдвигаемое окно. В качестве иллюстрации к сказанному приведем небольшую программку, которая плавно перемещает по экрану симпатичный паровозик, позаимствованный нами из спрайт-файла SPRITE2B пакета Laser Basic (рис. 7.2).

```
        ORG   60000
        ENT   $
        XOR   A
        CALL  8859
        LD    A,5
        LD    (23693),A
        CALL  3435
        LD    A,2
        CALL  5633
; Начальная установка регистров процедуры PTBL
        LD    B,10
        LD    C,0
        LD    A,SPRPUT
        LD    HL,PAROW
; Вывод на экран «паровозика»
        CALL  PTBL
; Задание параметров окна
```

```
        LD    HL,#A00      ;COL = 0, ROW = 10
        LD    (COL),HL
        LD    HL,#320      ;LEN = 32, HGT = 3
        LD    (LEN),HL
        LD    B,0          ;задание длины пробега «паровозика»
                           ; (0 = 256 пикселей)
MOVE    PUSH  BC
        CALL  SCR_RT       ;обращение к процедуре скроллинга вправо
        POP   BC
        DJNZ  MOVE
        RET
; Подпрограмма скроллинга окна вправо
SCR_RT .........
; Подпрограмма вывода спрайта
PTBL   .........
; Переменные к процедуре скроллинга
COL     DEFB  0
ROW     DEFB  0
LEN     DEFB  0
HGT     DEFB  0
; Заголовок данных для «паровозика»
PAROW   DEFB  14,0,0,5,0,1,5,0,2,5,0,3,5
        DEFB  1,0,5,1,1,5,1,2,5,1,3,5,1,4,5
        DEFB  2,0,5,2,1,5,2,2,5,2,3,5,2,4,5
; Данные
        DEFB  0,0,0,3,15,63,64,95
        DEFB  0,0,0,255,255,254,1,255
        DEFB  0,0,0,192,160,70,201,73
        DEFB  0,0,0,0,30,33,26,18
        DEFB  24,248,152,253,133,181,181,181
        DEFB  33,39,62,255,0,127,127,127
        DEFB  246,73,146,255,0,254,252,251
        DEFB  230,83,134,189,133,133,173,214
        DEFB  0,192,96,160,160,160,160,96
        DEFB  181,133,253,0,255,38,20,15
        DEFB  127,0,255,0,255,83,138,7
        DEFB  244,11,247,0,255,41,69,131
        DEFB  35,216,228,3,249,148,35,192
        DEFB  192,32,216,176,96,192,128,0
```

Второй способ не многим сложнее первого. Он основан на многократном выводе спрайта на экран. Если на каждом шаге изменять на единицу одну из координат, то спрайт будет двигаться параллельно соответствующей границе экрана, если же менять сразу обе, то он начнет перемещаться по диагонали. Основное требование к спрайту - он должен иметь по краям пустое пространство шириной в одно знакоместо, иначе изображение, помещенное на экран на предыдущем шаге, не будет полностью затираться следующим выводимым спрайтом и по экрану потянется не предусмотренный программистом след. Таким образом, если пустые места сделаны вокруг всего спрайта, то его можно спокойно передвигать в любом направлении, если же заранее известно, что он будет перемещаться вправо и никуда больше (как в примере ниже), то достаточно оставить пустую полоску шириной в одно знакоместо только слева.

К сожалению, этот способ перемещения спрайта тоже не лишен недостатка, видного невооруженным глазом: вместе с изображением, действительно подлежащим удалению, спрайт как ластик сотрет вообще весь фон позади себя. Справиться с этим можно точно так же, как мы рекомендовали выше, - выводить спрайт на сплошной фон, лишенный сложного пейзажа, что приемлемо для ограниченного числа игровых сюжетов. В качестве иллюстрации приведем программу, которая передвигает по экрану слева направо маленького динозавра, перебравшегося к нам из игры LITTLE PUFF:

```
        ORG   60000
        ENT   $
        XOR   A
        CALL  8859
        LD    A,7
        LD    (23693),A
        CALL  3435
        LD    A,2
        CALL  5633
; Основная часть программы
        LD    C,-4
MOVE    LD    B,10
        LD    A,SPRPUT
        LD    HL,DIN
        PUSH  BC
        CALL  PTBL
        LD    BC,10
        CALL  7997
        POP   BC
        INC   C
        LD    A,C
        CP    32
        JP    M,MOVE      ;если координата меньше 32 (с учетом знака)
        RET
; Подпрограмма вывода спрайта
PTBL    .........
; Заголовок данных для «динозавра»
DIN     DEFB  16,0,0,4,0,1,4,0,2,4,0,3,4
        DEFB  1,0,4,1,1,4,1,2,4,1,3,4
        DEFB  2,0,4,2,1,4,2,2,4,2,3,4
        DEFB  3,0,4,3,1,4,3,2,4,3,3,4
; Данные:
        DEFB  0,0,0,0,0,0,0,0
        DEFB  0,0,0,0,0,0,0,0
        DEFB  0,0,0,0,0,70,117,42
        DEFB  0,0,0,0,0,64,160,64
        DEFB  0,0,0,0,0,0,0,0
        DEFB  0,0,2,6,11,21,46,31
        DEFB  78,72,110,52,123,111,96,31
        DEFB  192,128,219,173,71,254,127,0
        DEFB  0,0,0,0,0,0,0,0
        DEFB  53,100,85,181,20,4,0,0
        DEFB  205,188,63,119,103,231,55,185
        DEFB  248,0,0,192,224,240,208,224
        DEFB  0,0,0,0,0,0,0,0
        DEFB  0,0,0,0,0,0,1,0
        DEFB  28,57,67,167,156,191,157,0
        DEFB  24,120,184,48,38,140,96,0
        DEFB  0,0,0,0,0,0,0,0
```

Программа содержит всего один цикл и настолько проста, что в дополнительных комментариях не нуждается.

Третий способ использует вывод спрайтов на экран по принципу XOR, который заложен в процедуре PTBL (как, впрочем, и другие). Применение принципа XOR для объединения изображений позволяет легко справиться с проблемой восстановления фона при перемещении спрайтов. Перечислим вначале все операции, которые должна выполнить программа:

- спрайт помещается на экран процедурой PTBL, в режиме XOR;
- через некоторое время (должен же спрайт немного побыть на экране) вторично выполняется процедура PTBL для того же спрайта, опять же по принципу XOR, при этом он стирается;

- координаты спрайта (или одна из них) изменяются и все повторяется сначала, при этом спрайт появляется на экране уже в другом месте.

Проделывая все это многократно, можно получить неплохой эффект мультипликации с сохранением фона.

Продемонстрируем этот способ на примере программы, которая в действии выглядит следующим образом. По дороге с ружьем на изготовку двигается солдат, медленно, но верно приближаясь к стенке из белого кирпича. Можно легко заметить, что движение состоит из двух фаз, каждой из которых, очевидно, должен соответствовать один спрайт: первый - ноги вместе и ружье чуть-чуть приподнято и второй - ноги расставлены в шаге, а ружье при этом опускается немного вниз. Когда солдат проходит мимо стенки, их картинки начинают смешиваться по принципу XOR, что мы и наблюдаем на экране - появляется какое-то хаотическое изображение, как будто человек продирается сквозь стену, а не идет мимо нее. Однако после того как стенка оказывается позади солдата, мы обнаруживаем, что оба изображения полностью восстановились.

```
        ORG   60000
        ENT   $
        LD    A,4
        LD    (23693),A
        XOR   A
        CALL  8859
        CALL  3435
        LD    A,2
        CALL  5633
; Рисование пейзажа, состоящего из дороги и стены
        CALL  GRUNT
; Вывод первой фазы спрайта «солдат» в режиме XOR
        LD    B,9         ;задаем координату Y
        LD    C,-3        ;начальное значение координаты X
LOOP    LD    A,SPRXOR    ;устанавливаем режим вывода
        LD    HL,SOLD1    ;задаем адрес спрайта 1 фазы
        PUSH  HL
        PUSH  BC
        CALL  PTBL
        LD    BC,20       ;задержка спрайта на экране
        CALL  7997
        POP   BC
        POP   HL
; Повторный вывод первой фазы спрайта в режиме XOR
        LD    A,SPRXOR
        PUSH  HL
        PUSH  BC
        CALL  PTBL
        POP   BC
        POP   HL
        INC   C           ;увеличиваем координату X
; Вывод второй фазы спрайта «солдат» в режиме XOR
        LD    A,SPRXOR
        LD    HL,SOLD2    ;задаем адрес спрайта 2 фазы
        PUSH  HL
        PUSH  BC
        CALL  PTBL
        LD    BC,20
        CALL  7997
        POP   BC
        POP   HL
; Повторный вывод второй фазы спрайта в режиме XOR
        LD    A,SPRXOR
```

```
        PUSH   HL
        PUSH   BC
        CALL   PTBL
        POP    BC
        POP    HL
        INC    C            ;увеличиваем координату X
        LD     A,C          ;количество шагов солдата
        CP     32           ;проверка условия конца «дороги»
        JP     M,LOOP
        RET
; Подпрограмма рисования пейзажа
GRUNT   EXX
        PUSH   HL
        LD     BC,#4700     ;начало горизонтальной линии (B=71, C=0)
        CALL   8933
; Рисование «дорожки»
        LD     DE,#101
        LD     BC,250
        CALL   9402
; Рисование «стены»
        LD     BC,#915      ;B = 9, C = 21
STEN    PUSH   BC
        LD     A,SPRPUT     ;устанавливаем режим вывода
        LD     HL,STENA     ;задаем адрес спрайта «стена»
        CALL   PTBL
        POP    BC
        INC    B
        LD     A,B
        CP     13
        JR     C,STEN
        POP    HL
        EXX
        RET
PTBL    .........
; Заголовок данных первой фазы спрайта «солдат»
SOLD1   DEFB   10
        DEFB   0,0,7, 0,1,7, 1,0,7, 1,1,7
        DEFB   2,0,7, 2,1,7, 2,2,7
        DEFB   3,0,7, 3,1,7, 3,2,7
; Данные первой фазы спрайта «солдат»
        DEFB   1,4,13,27,91,35,28,3
        DEFB   128,32,112,184,182,140,56,192
        DEFB   12,16,19,15,15,7,3,27
        DEFB   8,32,12,156,192,152,172,192
        DEFB   60,63,114,45,31,47,112,123
        DEFB   250,7,0,183,160,44,192,56
        DEFB   0,0,2,255,96,224,0,0
        DEFB   51,7,1,6,12,19,30,15
        DEFB   220,220,216,33,27,11,135,134
        DEFB   0,0,0,128,64,192,128,0
; Заголовок данных второй фазы спрайта «солдат»
SOLD2   DEFB   9
        DEFB   0,0,7, 0,1,7, 1,0,7, 1,1,7
        DEFB   2,0,7, 2,1,7, 2,2,7
        DEFB   3,0,7, 3,1,7
; Данные второй фазы спрайта
        DEFB   3,8,26,55,183,71,56,7
        DEFB   0,64,224,112,108,24,112,128
        DEFB   24,32,38,31,31,15,7,55
        DEFB   16,64,24,56,128,48,104,128
        DEFB   121,254,239,200,178,127,62,204
        DEFB   244,14,251,0,239,65,91,0
        DEFB   0,0,0,8,252,128,128,0
        DEFB   243,111,15,0,6,6,1,15
        DEFB   184,184,184,0,48,214,173,239
```

```
; Заголовок спрайта «стена»
STENA  DEFB  2
       DEFB  0,0,7, 0,1,7
; Данные спрайта «стена»
       DEFB  0,223,223,223,0,253,253,253
       DEFB  0,223,223,223,0,253,253,253
```

Четвертый способ основан на принципе записи части экранного изображения в буферную область памяти с последующим его возвратом на экран. Сначала поясним суть этого способа, а затем приведем небольшую программу, которая его иллюстрирует. В программе ГЕНЕРАТОР СПРАЙТОВ для сохранения образа экрана в памяти использовалась процедура GTBL и чтобы не писать еще одну подпрограмму, применим ее же для пересылки в буфер фрагмента экранного изображения. Вставляя эту процедуру в программу, следует предварительно внести в нее небольшие изменения:

- убрать первые четыре строки (до CALL OUT_BT);
- все команды CALL OUT_BT заменить на

```
LD    (IX),A
INC   IX
```

- в самом конце процедуры GTBL (непосредственно перед инструкцией RET) убрать команды PUSH IX и POP BC;
- внутренняя подпрограмма OUT_BT также не нужна, поэтому ее можно опустить.

В остальном все остается без изменений. Поскольку экранное изображение сохраняется в памяти, нужно позаботиться о выделении для этих целей некоторого рабочего буфера. Чтобы буфер не перекрыл занятую программой память, следует знать не только адрес его начала (который, кстати, понадобится для возврата полученного процедурой GTBL изображения), но и его размер. Вычислить его можно исходя из принятого нами формата спрайтов: количество знакомест (N_SYM) плюс заголовок (N_SYM×3) плюс данные (N_SYM×8). Итого получится N_SYM×11+1. Для небольших спрайтов вполне можно включить буфер в саму программу, воспользовавшись директивой ассемблера

```
BUFFER DEFS  N_SYM*11+1
```

Добавим, что при таком способе задания буфера размер его может быть больше рассчитанного, но ни в коем случае не меньше, иначе сохраняемые коды уничтожат часть программы! Кроме этого, надо сказать, что DEFS имеет смысл использовать только для сохранения небольших участков экрана, а при работе с большими изображениями (или когда одновременно сохраняются много окон) лучше выделить для этих целей некоторый участок памяти вне программы, чтобы сократить размер исполняемого модуля.

Для восстановления изображения нужно воспользоваться процедурой PTBL, задав в качестве адреса спрайта метку BUFFER или абсолютный адрес буфера, если он находится вне программы (хотя в этом случае его удобнее задать как константу).

Перечислим основные этапы реализации описываемого способа передвижения спрайта:

- процедурой GTBL забираем в буфер часть экранного изображения в форме прямоугольного окна;
- в это же место процедурой PTBL (в режиме SPRPUT) выводим спрайт;
- делаем небольшую задержку;
- ранее сохраненное окно с изображением части экрана переносим процедурой PTBL (в режиме SPRPUT) обратно на экран и в то же самое место;

- изменяем координаты спрайта (либо одну из них, как в примере) и повторяем перечисленные выше действия.

При составлении программы игры необходимо следить за тем, чтобы спрайт не выходил за пределы экрана, по крайней мере влево и вверх, так как этого не допускает процедура GTBL.

Эффективность способа продемонстрируем с помощью приведенной ниже программы, которая передвигает по экрану человечка из игры EXPRESS. Он пробегает мимо кустов, закрывая их собой по очереди, однако за его спиной кусты вновь появляются. Добежав до лежащего на дороге камня, человечек спотыкается и падает, на этом действие микромультфильма заканчивается (но вы можете попытаться его продолжить).

```
        ORG   60000
        ENT   $
N_SYM   EQU   6           ;задаем количество знакомест окна
        LD    A,6
        LD    (23693),A
        XOR   A
        CALL  8859
        CALL  3435
        LD    A,2
        CALL  5633
; Начало программы
        CALL  GRUNT       ;рисуем пейзаж
        LD    C,0         ;начальное значение координаты X для
                          ; «бегущего человечка»
; Перенос окна с частью изображения в буфер
LOOP    LD    B,10        ;COL = C, ROW = 10
        LD    (COL),BC
        LD    HL,#302     ;LEN = 2, HGT = 3
        LD    (LEN),HL
        LD    IX,BUFFER   ;в IX заносим начальный адрес буфера
        LD    A,N_SYM     ;в A заносим площадь окна
        PUSH  BC
        CALL  GTBL        ;образ окна переносим в память
        POP   BC
; Выводим человечка на то место, где было окно
        LD    HL,MAN1     ;задаем адрес спрайта «человечек»
        LD    A,SPRPUT    ;задаем режим вывода
        PUSH  BC
        CALL  PTBL
; Вводим небольшую задержку
        LD    BC,10
        CALL  7997
        POP   BC
; Ранее запомненное окно с изображением части экрана переносим
;  из буфера обратно на экран
        LD    HL,BUFFER   ;задаем начальный адрес буфера
                          ; с изображением части экрана
        LD    A,SPRPUT    ;устанавливаем режим вывода
        PUSH  BC
        CALL  PTBL        ;вывод окна на экран
        POP   BC
        INC   C           ;увеличиваем координату X человечка
        LD    A,C
        CP    20
        JR    C,LOOP      ;(или JP M,LOOP)
; Вывод человечка, споткнувшегося о камень
        LD    BC,#B16     ;B = 11, C = 22
        LD    A,SPRPUT    ;устанавливаем режим вывода
        LD    HL,MAN2     ;задаем адрес спрайта «упавший человечек»
```

```
        CALL  PTBL
        RET
PTBL    .........
; измененная процедура GTBL
GTBL    LD    (IX),A
        INC   IX
        LD    A,(ROW)
        LD    L,A
        LD    H,0
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        LD    A,#58
        ADD   A,H
        LD    H,A
        LD    A,(COL)
        ADD   A,L
        LD    L,A
        LD    DE,(LEN)
        LD    BC,0
GTBL1   PUSH  BC
        PUSH  DE
        PUSH  HL
GTBL2   LD    A,B
        LD    (IX),A
        INC   IX
        LD    A,C
        LD    (IX),A
        INC   IX
        LD    A,(HL)
        LD    (IX),A
        INC   IX
        INC   HL
        INC   C
        DEC   E
        JR    NZ,GTBL2
        POP   HL
        LD    DE,32
        ADD   HL,DE
        POP   DE
        POP   BC
        INC   B
        DEC   D
        JR    NZ,GTBL1
        LD    A,(HGT)
        LD    B,A
        LD    A,(ROW)
GTBL3   PUSH  AF
        PUSH  BC
        CALL  3742
        LD    A,(COL)
        ADD   A,L
        LD    L,A
        LD    A,(LEN)
        LD    B,A
GTBL4   PUSH  BC
        PUSH  HL
        LD    B,8
GTBL5   LD    A,(HL)
        LD    (IX),A
        INC   IX
        INC   H
        DJNZ  GTBL5
```

```
       POP   HL
       INC   HL
       POP   BC
       DJNZ  GTBL4
       POP   BC
       POP   AF
       INC   A
       DJNZ  GTBL3
       RET
; Рисование пейзажа. Еще раз хотим напомнить вам о необходимости
;  сохранения HL' при использовании подпрограммы 9402 и аналогичных
;  ей. При вызове программы из GENS это не критично, а вот в Бейсик
;  будет уже не вернуться.
GRUNT  EXX
       PUSH  HL
       LD    BC,#4700    ;B = 71, C = 0
       CALL  8933
; Рисование «дорожки»
       LD    DE,#101
       LD    BC,250
       CALL  9402
; Установка «камня»
       LD    BC,#C15     ;B = 12, C = 21
       LD    A,SPRPUT
       LD    HL,KAM
       CALL  PTBL
; Вывод двух «кустов»
       LD    BC,#B05     ;B = 11, C = 5
       LD    A,SPRPUT
       LD    HL,KUST
       CALL  PTBL
       LD    BC,#B10     ;B = 11, C = 16
       LD    A,SPRPUT
       LD    HL,KUST
       CALL  PTBL
       POP   HL
       EXX
       RET
; Графические переменные
COL    DEFB  0
ROW    DEFB  0
LEN    DEFB  0
HGT    DEFB  0
; Буфер для сохранения окна экрана
BUFFER DEFS  N_SYM*11+1
; Заголовок спрайта «бегущий человечек»
MAN1   DEFB  6,0,0, 6,0,1, 6,1,0, 7,1,1,7
       DEFB  2,0,3, 2,1,3
; Данные спрайта «бегущий человечек»
       DEFB  83,111,255,127,255,255,127,255
       DEFB  127,252,254,249,254,243,249,48
       DEFB  190,31,31,15,15,7,3,3
       DEFB  208,220,248,112,128,224,192,80
       DEFB  27,123,231,7,15,14,12,30
       DEFB  40,172,230,224,240,112,56,60
; Заголовок спрайта «упавший человечек»
MAN2   DEFB  6,0,0, 3,0,1, 7,0,2,6
       DEFB  1,0,3, 1,1,7, 1,2,6
; Данные спрайта «упавший человечек»
       DEFB  0,0,0,0,0,7,22,59
       DEFB  3,1,7,31,63,127,255,255
       DEFB  164,127,254,253,254,254,255,255
       DEFB  123,124,124,63,79,112,63,15
       DEFB  63,31,10,13,6,2,0,0
       DEFB  127,127,255,255,255,47,237,89
```

```
; Заголовок и данные спрайта «камень»
KAM    DEFB  1, 0,0,7
       DEFB  0,0,0,30,103,159,254,124
; Заголовок спрайта «куст»
KUST   DEFB  4, 0,0,6, 0,1,4, 1,0,6, 1,1,4
; Данные спрайта «куст»
       DEFB  33,12,102,242,185,13,53,121
       DEFB  56,100,192,218,160,134,157,56
       DEFB  29,204,110,38,54,182,87,91
       DEFB  50,112,119,238,236,234,90,84
```

Пятый способ лучше всего начать с описания картинки, которую формирует приведенная нами программа, а не с особенностей метода, как мы это делали раньше. Дело в том, что здесь придется ввести некоторые новые понятия, такие, например, как маска, и, как говорится в таких случаях - «лучше один раз увидеть...» После запуска программы перед вами появится «морской» пейзаж, состоящий из синей поверхности воды, черного ночного неба, на котором, тем не менее, видны белые облака (скорее всего их освещает луна), а между водой и небом - живописные острова, вблизи которых медленно проплывает военный корабль. Особенность этого способа состоит в том, что изображение корабля закрывает остров только по контуру спрайта, который не обязательно будет обозначен прямыми линиями, проходящими по границам знакомест. Достигается это благодаря использованию маски, поэтому необходимо сказать несколько слов о том, что же это такое и как ее сделать.

Представим себе, что с помощью графического редактора вы создали какой-то спрайт, скажем, изобразили корабль (рис. 7.3, *а*). Затем, воспользовавшись функцией **Cut & paste window**, перенесите полученный спрайт немного правее и обведите его по контуру, закрасив всю внешнюю область и оставив зазор в один пиксель (рис. 7.3, *б*). Наконец, удалите все, что расположено внутри контура как показано на рис. 7.3, *в*. Так вот, то, что получилось на последнем рисунке, и есть маска для исходного спрайта «корабль».

а б в

Рис. 7.3. Изготовление маски спрайта

Перейдем к реализации способа перемещения спрайтов с восстановлением фона, основанного на применении маски:

- процедурой GTBL забираем в буфер часть экранного изображения;
- процедурой PTBL по принципу AND в то же место экрана помещаем маску корабля, в результате чего будет очищена не вся прямоугольная область экрана, а только внутренняя часть, которую вы оставили незаштрихованной (вспомните, в чем заключается принцип AND);
- процедурой PTBL по принципу OR внутрь маски помещаем спрайт корабля (объединение по OR не стирает предыдущего изображения, поэтому и нужно почистить экран маской);
- ранее сохраненное окно с изображением части экрана переносим из буфера обратно на экран, используя процедуру PTBL в режиме SPRPUT;
- изменяем координаты спрайта на новые, после чего повторяем перечисленные действия в том же порядке.

```
       ORG   60000
       ENT   $
N_SYM  EQU   36          ;задаем количество знакомест окна
```

```
        LD    A,6
        LD    (23693),A
        XOR   A
        CALL  8859
        CALL  3435
        LD    A,2
        CALL  5633
; Начало программы
        CALL  MORE          ;рисуем «морской» пейзаж
        LD    C,0           ;начальная координата Y корабля
LOOP    LD    B,8           ;задаем параметры окна, в которое
                            ; поместим изображение части экрана
                            ; (COL = C, ROW = 8)
        LD    (COL),BC
        LD    HL,#409       ;LEN = 9, HGT = 4
        LD    (LEN),HL
        LD    IX,BUFFER     ;в IX заносим начальный адрес буфера
        LD    A,N_SYM       ;в регистр A заносим площадь окна
        PUSH  BC
        CALL  GTBL          ;образ окна переносим в память
        POP   BC
; По принципу AND на экран помещаем маску
        LD    B,8
        LD    HL,MASKA      ;задаем адрес маски
        LD    A,SPRAND      ;устанавливаем режим вывода
        PUSH  BC
        CALL  PTBL
        POP   BC
; По принципу OR в маску помещаем спрайт «корабль»
        LD    B,8
        INC   C             ;увеличиваем координату X «корабля»
        LD    HL,KORAB      ;задаем адрес спрайта «корабля»
        LD    A,SPROR       ;устанавливаем режим вывода
        PUSH  BC
        CALL  PTBL          ;выводим корабль
        LD    BC,20         ;вводим задержку
        CALL  7997
        POP   BC
; Ранее запомненное окно с изображением части экрана переносим
;  из буферной области памяти обратно на экран
        LD    B,8
        DEC   C
        LD    HL,BUFFER     ;устанавливаем начальный адрес буфера
        LD    A,SPRPUT      ;устанавливаем режим вывода
        PUSH  BC
        CALL  PTBL
        POP   BC
        INC   C             ;увеличиваем координату X корабля
        LD    A,C           ;конечная координата X корабля
        CP    32            ;проверяем, не вышел ли корабль
                            ; за пределы экрана
        JR    C,LOOP
        RET
; Рисование «моря» синим цветом
MORE    LD    A,14
        LD    (23693),A
        CALL  3435
        LD    A,2
        CALL  5633
        LD    BC,320
        LD    HL,#5800
; Изображение черного «неба»
NEBO    LD    (HL),7
        INC   HL
        DEC   BC
```

```
        LD    A,B
        OR    C
        JR    NZ,NEBO
        LD    BC,#606     ;B = 6, C = 6
        LD    A,SPRPUT    ;устанавливаем режим вывода
        LD    HL,OBL      ;задаем адрес спрайта «облако»
        CALL  PTBL        ;печать «облака»
        LD    BC,#414     ;B = 4, C = 20
        LD    A,SPRPUT
        LD    HL,OBL
        CALL  PTBL        ;печать еще одного «облака»
; Вывод на экран спрайта «остров»
OSTROV LD     BC,#908     ;B = 9, C = 8 - координаты
        LD    A,SPRPUT
        LD    HL,LAND     ;задаем адрес спрайта
        CALL  PTBL        ;выводим его на экран
        RET
PTBL    .........
GTBL    .........
;Графические переменные
COL     DEFB  0
ROW     DEFB  0
LNG     DEFB  0
HGT     DEFB  0
; Резервирование памяти для окна
BUFFER DEFS   N_SYM*11+1
; Заголовок данных спрайта «корабль»
KORAB   DEFB  13
        DEFB  0,3,7, 1,1,4, 1,2,7, 1,3,7, 1,4,7
        DEFB  1,5,4, 2,0,15, 2,1,15, 2,2,15, 2,3,15
        DEFB  2,4,15, 2,5,15, 2,6,15
; Данные спрайта «корабль»
        DEFB  0,0,0,16,16,16,16,24
        DEFB  0,0,0,0,0,0,0,31
        DEFB  0,0,0,0,3,4,7,197
        DEFB  60,38,30,60,149,159,179,191
        DEFB  0,0,0,0,128,128,111,229
        DEFB  0,0,0,0,0,0,240,0
        DEFB  0,1,0,127,85,43,31,0
        DEFB  2,251,84,255,85,187,255,0
        DEFB  183,254,201,255,85,255,255,0
        DEFB  238,12,191,213,127,255,255,0
        DEFB  63,217,255,85,255,255,255,0
        DEFB  127,40,255,85,186,255,255,0
        DEFB  128,0,254,20,184,240,224,0
; Заголовок маски
MASKA   DEFB  14
        DEFB  0,3,7, 1,1,4, 1,2,4, 1,3,4, 1,4,4
        DEFB  1,5,4, 1,6,4, 2,0,15, 2,1,15, 2,2,15
        DEFB  2,3,15, 2,4,15, 2,5,15, 2,6,15
; Данные маски
        DEFB  255,255,199,199,199,199,195,129
        DEFB  255,255,255,255,255,255,192,192
        DEFB  255,255,255,248,240,240,16,0
        DEFB  128,128,128,0,0,0,0,0
        DEFB  255,255,255,63,63,0,0,0
        DEFB  255,255,255,255,255,7,7,0
        DEFB  255,255,255,255,255,255,255,63
        DEFB  252,252,0,0,0,0,128,192
        DEFB  0,0,0,0,0,0,0,0
        DEFB  0,0,0,0,0,0,0,0
        DEFB  0,0,0,0,0,0,0,0
        DEFB  0,0,0,0,0,0,0,0
        DEFB  0,0,0,0,0,0,0,0
        DEFB  63,0,0,0,1,3,7,15
```

```
; Заголовок спрайта «облако»
OBL     DEFB  4
        DEFB  0,0,7, 0,1,7, 0,2,7, 0,3,7
; Данные спрайта «облако»
        DEFB  0,48,218,255,255,98,56,0
        DEFB  60,255,86,251,247,247,46,112
        DEFB  0,58,127,207,123,239,118,57
        DEFB  0,0,56,238,207,117,24,0
; Заголовок спрайта «остров»
LAND    DEFB  14
        DEFB  0,0,4, 0,1,4, 0,2,4, 0,3,4
        DEFB  0,4,4, 0,5,4, 0,6,4, 0,7,4
        DEFB  0,8,4, 0,9,4, 0,10,4, 0,11,4
        DEFB  0,12,4, 0,13,4
; Данные спрайта «остров»
        DEFB  0,0,0,0,0,0,28,255
        DEFB  0,0,0,0,0,28,255,255
        DEFB  0,0,0,0,48,255,255,255
        DEFB  0,0,0,0,0,0,252,255
        DEFB  0,0,0,0,0,3,63,255
        DEFB  0,0,48,252,255,255,253,252
        DEFB  0,0,0,0,193,243,255,255
        DEFB  0,0,0,0,193,243,255,255
        DEFB  0,0,0,0,0,3,63,255
        DEFB  0,0,48,252,255,255,253,252
        DEFB  0,0,0,0,193,243,255,255
        DEFB  0,0,48,252,255,255,253,252
        DEFB  0,0,0,0,193,243,255,255
        DEFB  0,0,0,0,0,192,252,255
```

ПОСТРОЕНИЕ ПЕЙЗАЖЕЙ

Деление изображения на пейзаж и спрайты, с точки зрения его формирования на экране, вообще-то является достаточно условным, поскольку все процедуры, используемые для рисования спрайтов, применимы и при создании пейзажей. Однако если рассматривать этот вопрос с игровой точки зрения, то здесь уже можно увидеть существенные различия, которые позволяют выделить создание пейзажа как самостоятельную задачу. Прежде всего, он в большинстве игр занимает всю площадь экрана, а спрайты, хотя и бывают довольно большими, но все же ограничены в размерах. Другое важное отличие - пейзаж, как правило, неподвижен или, если и изменяется, то незначительно.

Рассмотрим несколько приемов формирования пейзажа и проиллюстрируем их небольшими программами и рисунками. Начать нужно, по-видимому, с самого простого - с голубого неба (синей глади воды, сплошного ковра зеленой травы, желтого песка пустыни и т. д.). С этой целью достаточно в самом начале программы задать:

```
        LD    A,40        ;INK 0, PAPER 5
        LD    (23693),A
        CALL  3435
```

и требуемый фон готов. Несколько сложнее обстоят дела с космическим пространством. То есть, задать черный цвет «бумаги» никакой проблемы не составит, но вот со звездами не все так просто. Можно, конечно, нарисовать и закодировать маленькие спрайты и даже заставить их случайным образом вспыхивать (особенно хорошо выглядит задание повышенной яркости в фазе максимума). Но ведь каждый спрайт-звездочка - это целое знакоместо, поэтому много ли их можно разместить на экране. Значительно лучше смотрятся звездочки размером в

пиксель, как, например, в игре ELITE, но тут уже требуется небольшая программка, которую мы предлагаем вам написать самостоятельно.

Простые пейзажи можно сформировать из отдельных спрайтов, либо фрагментов, составленных из повторяющихся спрайтов. Здесь «трава» содержит восемь спрайтов, каждый из которых занимает четыре знакоместа, «куст» - из трех спрайтов, а «человечек» - это отдельный спрайт из шести знакомест. Ниже приводится соответствующая этому пейзажу программа:

```
        ORG   60000
        ENT   $
        LD    A,5
        LD    (23693),A
        XOR   A
        CALL  8859
        CALL  3435
; Вывод восьми спрайтов «трава»
        LD    B,8          ;задаем количество выводимых
                           ; спрайтов «трава»
        LD    HL,DATA1     ;читаем адрес блока координат
        LD    DE,TRAW
        CALL  POSIT
; Вывод трех спрайтов «куст»
        LD    B,3
        LD    HL,DATA2
        LD    DE,KUST
        CALL  POSIT
; Вывод спрайта «человечек»
        LD    B,1
        LD    HL,DATA3
        LD    DE,MAN
; Подпрограмма начальной установки регистров для процедуры PTBL
POSIT   PUSH  BC
        LD    B,(HL)       ;устанавливаем Y
        INC   HL
        LD    C,(HL)       ;устанавливаем X
        INC   HL
        PUSH  HL
        PUSH  DE
        EX    DE,HL
        LD    A,SPRPUT     ;вывод с наложением
        CALL  PTBL
        POP   DE
        POP   HL
        POP   BC
        DJNZ  POSIT
        RET
PTBL    .........
; Задание пар координат (Y,X) для вывода восьми спрайтов «трава»
DATA1   DEFB  17,0,17,4,17,8,17,12,17,16
        DEFB  17,20,17,24,17,28
; Заголовок спрайта «трава»
TRAW    DEFB  4, 0,0,4, 0,1,4, 0,2,4, 0,3,4
; Данные для спрайта «трава»
        DEFB  193,106,191,253,255,235,181,36
        DEFB  18,170,247,255,255,94,107,41
        DEFB  101,200,255,219,255,255,87,10
        DEFB  18,170,247,255,255,95,107,41
; Задание пар координат для печати трех спрайтов «куст»
DATA2   DEFB  11,14,13,14,15,14
; Заголовок спрайта «куст»
KUST    DEFB  4, 0,0,6, 0,1,4, 1,0,6, 1,1,4
```

```
; Данные для спрайта «куст»
       DEFB  33,12,102,242,185,13,53,121
       DEFB  56,100,192,218,160,134,157,56
       DEFB  29,204,110,28,54,182,87,91
       DEFB  50,112,119,238,236,234,90,84
; Пара координат для вывода «человечка»
DATA3  DEFB  14,23
; Заголовок спрайта «человечек»
MAN    DEFB  6, 0,0,5, 0,1,5, 1,0,5
       DEFB  1,1,5, 2,0,5, 2,1,5
; Данные для спрайта «человечек»
       DEFB  3,15,17,36,48,16,26,31
       DEFB  192,240,248,56,132,60,120,240
       DEFB  25,14,7,24,31,59,55,55
       DEFB  240,192,184,124,244,246,250,194
       DEFB  56,27,3,0,6,6,7,0
       DEFB  26,104,160,144,48,36,184,0
```

Задание в виде отдельных блоков данных (DATA1...DATA3) пар координат, первая из которых соответствует вертикальной позиции спрайта, а вторая - горизонтальной, позволяет легко «разбрасывать» спрайты по всему экрану, создавая любые их комбинации. Вывод спрайтов осуществляется процедурой PTBL.

Пейзажи в игровых программах довольно часто формируются из многократно повторяющихся фрагментов, причем последние могут занимать как одно знакоместо, так и состоять из нескольких. Взять, к примеру, всевозможные лабиринты, разрезы зданий и других сооружений, карты боевых действий и так далее, всего просто не перечесть. Составим программу, которая формирует картинки именно такого типа, причем для простоты будем считать, что все различающиеся фрагменты имеют размеры одного знакоместа. В этом случае для их быстрого вывода на экран можно воспользоваться способом, аналогичным тому, который использовался в рассмотренной раньше процедуре PRSYM.

Предположим, что мы хотим получить изображение, которое представляет собой внешнюю часть лабиринта к создаваемой игре. Соответствующая этой картинке программа выглядит так:

```
       ORG   60000
       LD    IX,LAB_0     ;перед вызовом в IX заносится адрес
                          ; данных лабиринта
LABS1  LD    C,(IX+1)     ;позиция начального элемента по горизонтали
       LD    B,(IX+2)     ;позиция начального элемента по вертикали
       LD    A,(IX+3)     ;количество повторений и направление
       AND   31
       JR    Z,LABS5      ;если выводится одиночный элемент
       LD    E,A
LABS2  LD    A,(IX)       ;код символа (0...5)
       CALL  PRINT        ;вывод символа на экран
       BIT   7,(IX+3)     ;проверка направления вывода
       JR    NZ,LABS3
       INC   C            ;слева направо
       JR    LABS4
LABS3  INC   B            ;сверху вниз
LABS4  DEC   E            ;следующий элемент
       JR    NZ,LABS2
       JR    LABS6
LABS5  LD    A,(IX)       ;вывод одиночного элемента
       CALL  PRINT
LABS6  LD    DE,4         ;увеличиваем адрес в блоке данных
       ADD   IX,DE        ; на 4 байта
       LD    A,(IX)       ;проверка на достижение конца блока данных
       INC   A            ;если -1 (255)
```

```
        JR    NZ,LABS1
        RET
PRINT   PUSH  BC
        PUSH  DE
        PUSH  HL
        LD    L,A         ;по коду определяем адрес символа
        LD    H,0
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        LD    DE,D_SYMB
        ADD   HL,DE
        PUSH  HL
        LD    A,B         ;вычисляем адрес видеобуфера
        CALL  3742
        LD    A,L
        ADD   A,C
        LD    L,A
        POP   DE
        LD    B,8
        PUSH  HL
PRINT1  LD    A,(DE)      ;переносим на экран 8 байт
        LD    (HL),A
        INC   DE
        INC   H
        DJNZ  PRINT1
        POP   HL          ;восстанавливаем начальный адрес экрана
        LD    A,H         ;рассчитываем адрес атрибутов
                          ; соответствующего знакоместа
        AND   #18
        RRCA
        RRCA
        RRCA
        ADD   A,#58
        LD    H,A
        LD    A,(23695)   ;записываем байт атрибутов в видеобуфер
        LD    (HL),A
        POP   HL
        POP   DE
        POP   BC
        RET
; Данные для построения лабиринта (рамки)
; IX+0 - номер символа в таблице D_SPR (0..5)
; IX+1 - начальная позиция экрана по горизонтали
; IX+2 - начальная позиция экрана по вертикали
; IX+3 - младшие 5 битов - количество повторений вывода символа,
;        7-й бит определяет направление вывода:
;            установлен - сверху вниз (задается символами @#80),
;            сброшен - слева направо
LAB_0   DEFB  0,2,8,0, 1,3,8,2
        DEFB  2,5,8,0, 3,5,9,0
        DEFB  5,5,10,0, 1,6,10,20
        DEFB  3,2,9,2@#80, 5,2,11,0
        DEFB  1,3,11,0, 2,4,11,0
        DEFB  3,4,12,8@#80, 4,4,20,0
        DEFB  1,3,20,0, 0,2,20,0
        DEFB  3,2,21,2@#80, 5,2,23,0
        DEFB  1,3,23,2, 4,5,23,0
        DEFB  3,5,22,0, 0,5,21,0
        DEFB  1,6,21,20, 4,26,10,0
        DEFB  3,26,9,0, 0,26,8,0
        DEFB  1,27,8,2, 2,29,8,0
        DEFB  3,29,9,2@#80, 4,29,11,0
        DEFB  1,28,11,0, 0,27,11,0
        DEFB  2,26,21,0, 3,26,22,0
```

```
        DEFB  5,26,23,0, 1,27,23,2
        DEFB  4,29,23,0, 3,29,21,2@#80
        DEFB  2,29,20,0, 1,28,20,0
        DEFB  5,27,20,0, 3,27,12,8@#80
        DEFB  0,3,9,0, 2,4,9,0
        DEFB  4,4,10,0, 5,3,10,0
        DEFB  0,27,9,0, 2,28,9,0
        DEFB  4,28,10,0, 5,27,10,0
        DEFB  0,27,21,0, 2,28,21,0
        DEFB  4,28,22,0, 5,27,22,0
        DEFB  0,3,21,0, 2,4,21,0
        DEFB  4,4,22,0, 5,3,22,0
        DEFB  -1              ;конец данных
; Данные символов (элементов лабиринта)
D_SYMB DEFB  127,213,159,191,253,182,248,181
        DEFB  255,85,255,255,85,170,0,255
        DEFB  254,83,249,245,121,181,121,181
        DEFB  249,181,249,181,249,181,249,181
        DEFB  249,117,249,245,81,169,3,254
        DEFB  249,189,255,191,213,170,192,127
```

Используя эту программу без всяких переделок и изменив только ее блоки данных, можно с успехом рисовать различные рамки для оформления кадров меню или окон, предназначенных для вывода всевозможной оценочной информации, правил игры и т. д.

Прежде чем перейти к следующему разделу, поясним встретившуюся в блоке данных LAB_0 не очень понятную запись @#80. В комментарии было сказано, что это означает установку 7-го бита в числе. Дело в том, что ассемблер GENS помимо простых арифметических операций сложения, вычитания, умножения и деления предоставляет возможность использования в выражениях поразрядных операций OR, XOR и AND. Обозначаются они соответственно символами **@**, **!** и **&**. Поэтому, например, выражение **2@#80** (2 OR #80) примет значение #82 или 130, а запись **%101!%110** (%101 XOR %110) после вычислений заменится числом %011 или 3.

ФОРМИРОВАНИЕ ИЗОБРАЖЕНИЯ В ПАМЯТИ

В данном разделе мы собираемся рассказать, пожалуй, о самом совершенном способе вывода спрайтов и пейзажа на экран. Разобравшись с приведенной здесь программой, вы поймете, как формируется игровое пространство в наиболее солидных фирменных игрушках, где общая площадь лабиринта, города или иной местности во многие десятки (а то и сотни) раз превышает тот клочок земли, который мы наблюдаем на экране.

Скажем несколько слов о принципе работы этой программы, о том, как создается протяженный пейзаж и как формируется изображение. В отличие от приведенных выше примеров здесь картинка не выводится сразу на экран, а строится в памяти. И только после того как наложен последний штрих, весь кадр быстро перебрасывается в видеобуфер как один спрайт. Мало того, что такой метод позволяет избежать мелькания картинок, здесь не нужно заботиться о восстановлении фона и размышлять на тему, как «подсунуть» один спрайт под другой. Строится изображение так: в предварительно очищенную область памяти, используемую в качестве так называемого «теневого» или виртуального окна, выводятся спрайты, занимающие самое «дальнее» положение, затем, последовательно приближаясь к ближнему плану, строится и все остальное изображение. Кстати, именно таким способом получена «трехмерная» графика в играх вроде ALIEN 8, KNIGHT LORE и им подобных.

В «серьезных» игровых программах «теневое» окно обычно занимает лишь немногим большее пространство, чем окно на экране, отмечающее игровое поле. Но вы прекрасно знаете, что, имея дело с окнами, приходится идти на определенные трудовые затраты. Обычно процедуры, выполняющие вывод в виртуальное окно, довольно хитроумны и отличаются весьма солидными размерами, поэтому поместить их в книге оказалось просто нереально. Чтобы выйти из положения и все же продемонстрировать такой прием, мы решили отойти от принципа экономии памяти и вместо окна использовали целый экран. Однако для программ, после трансляции которых исполняемый файл занимает меньше 30-35 килобайт, такое «расточительство», в общем, допустимо. Зато при правильном выборе адреса «теневого» экрана работа с ним практически не отличается от вывода в физический видеобуфер, поэтому к нему применимы все приведенные ранее процедуры лишь с несущественной доработкой.

Для облегчения всех расчетов виртуальный экран должен располагаться по «ровному» шестнадцатеричному адресу: #6000, #7000, #8000 и т. д. Тогда все вычисления будут такими же, как при работе с физическим экраном, только к полученному адресу нужно добавлять смещение (достаточно только разницу старших байтов). В нашей программе адрес «теневого» экрана равен #8000, следовательно, разность старших байтов будет #80-#40=#40. Зададим это смещение константой V_OFFS, чтобы при желании несложно было перенести виртуальный экран в любое другое место.

На рис. 7.4 показана часть игрового пространства, состоящего из протяженного пейзажа и микроавтобуса, проезжающего мимо домов. На переднем плане мелькают дорожные фонари. Выехав за пределы города, автомобиль останавливается на пару секунд, а затем начинает свое движение с исходной точки трассы. Просмотрев этот мультфильм несколько раз, вы обнаружите, что дома появляются не каким-то случайным образом, а занимают строго определенное положение на пути следования автомобиля. То есть в программе создается вполне конкретный пейзаж и каждый момент времени на экране виден только небольшой его фрагмент.

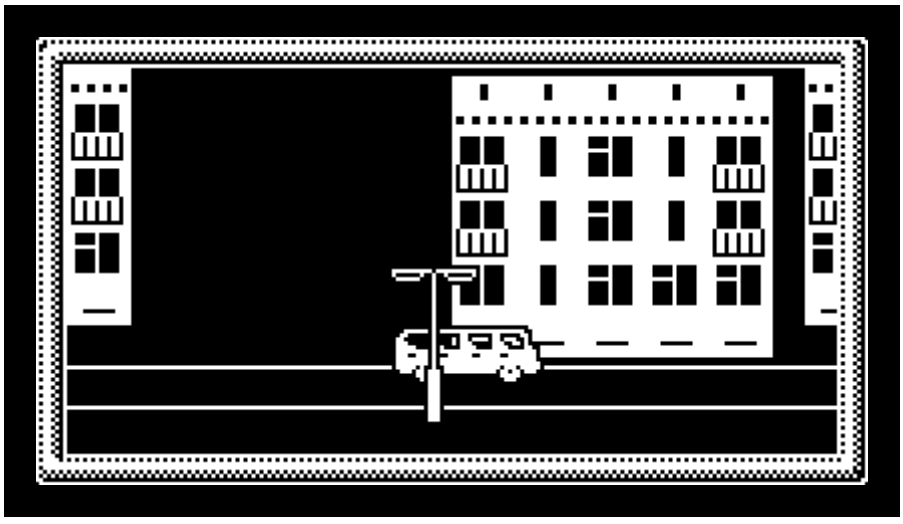


Рис. 7.4. Вывод изображения из «теневого» экрана

Поясним, как это достигается. В блоке данных, описывающих пейзаж (в программе он обозначен меткой D_LAND), указывается положение каждого дома (или другого объекта) на трассе, протяженность дома, его «удаленность» от дороги и адрес другого блока, задающего его внешний вид. Программа просматривает данные пейзажа и, исходя из положения автомобиля (он всегда выводится в центре окна), выбирает только те дома, которые целиком или частично попадают в окно экрана, то есть если правый угол дома не выходит за левую

границу и левый - за правую, а остальные пропускает. После этого начинается вывод объектов в «теневой» экран: сначала дома, затем рисуется дальний тротуар, автомобиль, ближний тротуар и, наконец, фонари. Как только построение закончено, окно «теневого» экрана переносится в физический видеобуфер и тем самым изображение становится видимым.

Так как объем книги не безграничен, в программе заданы только два типа зданий, а протяженность пейзажа измеряется 200 знакомест. Но вы можете дополнить графику, составив из имеющихся спрайтов другие дома. Попробуйте увеличить пробег автомобиля, введите новые спрайты, изображающие деревья, решетки ограды, арки, попытайтесь создать загородный пейзаж. Для этого вам нужно будет только расширить блоки данных, формат которых подробно описан в тексте программы. Не бойтесь экспериментировать, и результат, уверены, порадует вас.

Несмотря на относительную сложность программы, в ней не встретится неизвестных доселе команд. Единственное нововведение - это процедура ПЗУ, находящаяся по адресу 8020, которая служит для проверки нажатия клавиши **Break** (**Caps Shift/Space**). Подпрограмма не требует никаких входных параметров и сообщает о том, что **Break** нажата установкой на выходе флага переноса. В противном случае выполняется условие NC.

До того, как мы приведем текст программы, обращаем ваше внимание, что вызывается она только из Бейсика, так как ассемблер GENS пользуется своим собственным внутренним стеком и поскольку он может оказаться в любом месте (все зависит от адреса загрузки GENS), то в результате «теневой» экран может перекрыть стек, а что из этого следует, догадаться нетрудно.

```
       ORG   60000
; Адрес «теневого» (или виртуального) экрана = #8000
V_OFFS EQU   #40          ;старший байт смещения адреса
                          ; «теневого» экрана относительно
                          ; начального адреса физического экрана
       LD    A,69
       LD    (23693),A
       XOR   A
       CALL  8859
       CALL  3435
       LD    A,2
       CALL  5633         ;здесь необходима только для правильного
                          ; вывода атрибутов основного экрана
; Инициализация переменных
MAIN   LD    A,2
       LD    (X_LAMP),A
       LD    HL,0
       LD    (X_LAND),HL
       CALL  FRAME        ;рамка вокруг окна
MAIN1  CALL  CLS_V        ;очистка «теневого» экрана
; Формирование игрового поля в «теневом» экране
       CALL  LAND         ;вывод заднего плана (дома)
       CALL  LINE1        ;рисование дальнего тротуара
       CALL  PUTCAR       ;автомобиль
       CALL  LINE2        ;ближний тротуар
       CALL  LAMPS        ;фонари
; ----------------
       CALL  PUTVRT       ;вывод окна виртуального экрана
                          ; на физический
       CALL  8020         ;проверка нажатия клавиши Break
       RET   NC           ;выход из программы, если нажата
       LD    HL,(X_LAND)  ;изменение координаты автомобиля
```

```
       INC   HL
       LD    (X_LAND),HL
       LD    DE,200      ;если пройдено расстояние
       AND   A           ; меньше 200 знакомест,
       SBC   HL,DE
       JR    NZ,MAIN1    ; то движение продолжается
       LD    BC,100      ;иначе - пауза 2 сек.
       CALL  7997
       JR    MAIN        ; и переход к началу пути
; Рисование горизонтальных линий белого цвета, изображающих тротуары
LINE1  LD    L,#40
       LD    H,#4A+V_OFFS
       JR    LINE3
LINE2  LD    L,#60
       LD    H,#4C+V_OFFS
LINE3  PUSH  HL          ;адрес экрана здесь не рассчитывется,
                         ; а задается в явном числовом виде
       LD    D,H
       LD    E,L
       INC   DE
       LD    (HL),255    ;сплошная линия (если изменить число,
                         ; получится пунктирная)
       LD    BC,31
       LDIR              ;проводим линию
       POP   HL
       LD    A,H         ;расчет адреса атрибутов
       SUB   V_OFFS
       AND   #18
       RRCA
       RRCA
       RRCA
       ADD   A,#58+V_OFFS
       LD    H,A
       LD    D,H
       LD    E,L
       INC   DE
       LD    (HL),7      ;INK 7, PAPER 0
       LD    BC,31
       LDIR
       RET
; Вывод автомобиля в виртуальный экран
PUTCAR LD    BC,#90E     ;координаты фиксированы (B = 9, C = 14)
       LD    HL,SPR7     ;маска
       LD    A,SPRAND    ;по принципу AND
       CALL  PTBL
       LD    HL,SPR6     ;автомобиль
       LD    A,SPROR     ;по принципу OR
       JP    PTBL
; Рисование в виртуальном экране дорожных фонарей
LAMPS  LD    HL,X_LAMP
       LD    C,(HL)      ;горизонтальная координата самого левого
                         ; на экране фонаря
       LD    B,7         ;вертикальная координата фиксирована
       PUSH  HL
LAMPS1 LD    HL,SPR9     ;маска
       LD    A,SPRAND
       CALL  PTBL
       LD    HL,SPR8     ;фонарь
       LD    A,SPROR
       CALL  PTBL
       LD    A,20        ;следующий через 20 знакомест
       ADD   A,C
       LD    C,A
       CP    28
       JR    C,LAMPS1    ;закончить вывод?
```

```
        POP   HL
        DEC   (HL)          ;уменьшение координаты на 2
        DEC   (HL)
        RET   P             ;выход, если начальная координата
                            ; не отрицательна
        LD    (HL),18       ;иначе задаем начальную координату
        RET
X_LAMP  DEFB  0
; Подпрограмма очистки виртуального экрана
CLS_V   LD    H,#40+V_OFFS
        LD    L,0
        LD    D,H
        LD    E,L
        INC   DE
        LD    (HL),0
        LD    BC,6144
        LDIR
; Атрибуты
        LD    (HL),1        ;INK 1, PAPER 0
        LD    BC,767
        LDIR
        RET
; Формирование изображения здания, скомбинированного из нескольких
;  спрайтов (адрес соответствующего блока данных в паре HL)
PUT_B   LD    A,(HL)        ;количество блоков в доме
        INC   HL
PUT_B1  PUSH  AF
        PUSH  BC
        LD    A,(HL)        ;горизонтальная координата блока
        INC   HL
        ADD   A,C
        CP    2             ;проверка выхода за пределы окна
        JR    C,PUT_B3
        CP    29
        JR    NC,PUT_B3
        LD    C,A
        LD    A,(HL)        ;вертикальная координата блока
        INC   HL
        ADD   A,B           ;проверка выхода за пределы окна
        JP    M,PUT_B4
        LD    B,A
        LD    E,(HL)        ;адрес спрайта, изображающего блок
        INC   HL
        LD    D,(HL)
        INC   HL
        PUSH  HL
        EX    DE,HL
        LD    A,SPRPUT
        CALL  PTBL          ;выводим спрайт в «теневой» экран
        POP   HL
PUT_B2  POP   BC
        POP   AF
        DEC   A
        JR    NZ,PUT_B1     ;следующий блок
        RET
PUT_B3  INC   HL            ;обход, если блок выходит за пределы окна
PUT_B4  INC   HL
        INC   HL
        JR    PUT_B2
; Вывод помещающейся в окно части пейзажа
X_LAND  DEFW  0             ;координата автомобиля на трассе
LAND    LD    HL,D_LAND     ;блок данных, описывающих
                            ; местоположение и внешний вид домов
LAND1   LD    BC,(X_LAND)
        LD    E,(HL)        ;горизонтальная координата дома на трассе
```

```
        INC   HL
        LD    D,(HL)
        INC   HL
        LD    A,D
        OR    E
        RET   Z           ;0 - маркер конца блока данных
        LD    A,(HL)      ;протяженность дома в знакоместах
        INC   HL
        PUSH  HL
        PUSH  DE
        EX    DE,HL       ;вычисляем координату дальнего конца дома
        ADD   A,L
        LD    L,A
        LD    A,H
        ADC   A,0
        LD    H,A
        INC   HL
        AND   A
        SBC   HL,BC       ;сравниваем с текущим положением
                          ; автомобиля
        POP   DE
        JR    C,LAND3     ;если дом выходит за левый край окна
        LD    HL,28
        ADD   HL,BC
        SBC   HL,DE
        JR    C,LAND3     ;если дом выходит за правый край окна
        EX    DE,HL
        SBC   HL,BC       ;вычисление экранной координаты дома
        LD    C,L
        POP   HL
        LD    B,(HL)      ;вертикальная экранная координата дома
        INC   HL
        LD    E,(HL)      ;адрес блока данных, описывающих дом
        INC   HL
        LD    D,(HL)
        PUSH  HL
        EX    DE,HL
        CALL  PUT_B       ;вывод изображения дома в «теневой» экран
        POP   HL
LAND2   INC   HL
        JR    LAND1       ;следующий дом
LAND3   POP   HL          ;обход, если изображение дома не
                          ; попадает в окно экрана
        INC   HL
        INC   HL
        JR    LAND2
; ВНИМАНИЕ! В процедуре PTBL есть изменения!
PTBL
SPRPUT EQU    0
SPROR  EQU    #B6
SPRAND EQU    #A6
SPRXOR EQU    #AE
        PUSH  HL
        LD    (MODE),A
        LD    A,(HL)
        INC   HL
        PUSH  HL
        LD    L,A
        LD    H,0
        LD    E,L
        LD    D,H
        ADD   HL,HL
        ADD   HL,DE
        POP   DE
        ADD   HL,DE
```

```
        EX    DE,HL
PTBL1   PUSH  AF
        PUSH  BC
        LD    A,(HL)
        INC   HL
        PUSH  HL
        ADD   A,B
        CP    24
        JR    NC,PTBL4
        PUSH  DE
        CALL  3742
        POP   DE
; +++ Добавление +++
        LD    A,V_OFFS
        ADD   A,H
        LD    H,A
; +++
        EX    (SP),HL
        LD    A,(HL)
        EX    (SP),HL
        ADD   A,C
        CP    32
        JR    NC,PTBL4
        ADD   A,L
        LD    L,A
        LD    B,8
        PUSH  HL
PTBL2   LD    A,(DE)
MODE    NOP
        LD    (HL),A
        INC   DE
        INC   H
        DJNZ  PTBL2
        POP   BC
        LD    A,B
        AND   #18
        SRA   A
        SRA   A
        SRA   A
; +++ Изменение: вместо ADD A,#58 +++
        ADD   A,#58+V_OFFS
; +++
        LD    B,A
        POP   HL
        INC   HL
        LD    A,(HL)
        DEC   HL
        LD    (BC),A
PTBL3   POP   BC
        POP   AF
        INC   HL
        INC   HL
        DEC   A
        JR    NZ,PTBL1
        POP   HL
        RET
PTBL4   LD    HL,8
        ADD   HL,DE
        EX    DE,HL
        POP   HL
        JR    PTBL3
; Вывод окна «теневого» экрана на физический
PUTVRT  LD    A,8          ;отступ сверху (в пикселях)
        LD    B,12*8       ;высота окна (в пикселях)
PUT_V1  PUSH  AF
```

```
        PUSH  BC
        LD    C,4*8       ;отступ слева (в пикселях)
        CALL  8880        ;адрес физического экрана
        LD    D,H         ;сохраняем в DE
        LD    E,L
        LD    A,V_OFFS
        ADD   A,H         ;в HL - соответствующий адрес
        LD    H,A         ; «теневого» экрана
        LD    BC,24       ;ширина окна - 24 знакоместа
        LDIR
        POP   BC
        POP   AF
        INC   A           ;следующий ряд пикселей
        DJNZ  PUT_V1
; Перенос атрибутов
        LD    DE,#5824    ;адрес верхнего левого угла окна
        LD    L,E         ; в области атрибутов
        LD    A,D
        ADD   A,V_OFFS    ;вычисляем соответствующий адрес
        LD    H,A         ; в «теневом» экране
        LD    B,12
PUT_V2  PUSH  BC
        LD    BC,24
        LDIR              ;переносим 24 байта (по ширине окна)
        LD    BC,8        ;переходим к следующей
        ADD   HL,BC       ; строке экрана (32-24 = 8)
        EX    DE,HL
        ADD   HL,BC
        EX    DE,HL
        POP   BC
        DJNZ  PUT_V2
        RET
; Рисование рамки вокруг окна
;  (подпрограммы LABS1 и PRINT описаны в предыдущем разделе)
FRAME   LD    IX,DFRAME   ;Данные для построения рамки
LABS1   .........
PRINT   .........
; Блоки дома:
; Окно 1 (с рамой)
SPR1    DEFB  4
        DEFB  0,0,65, 0,1,65, 1,0,65, 1,1,65
        DEFB  255,193,193,193,255,193,193,193
        DEFB  255,7,7,7,7,7,7,7
        DEFB  193,193,193,255,255,255,255,255
        DEFB  7,7,7,255,255,255,255,255
; Окно 2 (без рамы)
SPR2    DEFB  4
        DEFB  0,0,65, 0,1,65, 1,0,65, 1,1,65
        DEFB  255,252,252,252,252,252,252,252
        DEFB  255,63,63,63,63,63,63,63
        DEFB  252,252,252,255,255,255,255,255
        DEFB  63,63,63,255,255,255,255,255
; Окно с балконом
SPR3    DEFB  4
        DEFB  0,0,65, 0,1,65, 1,0,65, 1,1,65
        DEFB  255,193,193,193,193,193,193,193
        DEFB  255,7,7,7,7,7,7,7
        DEFB  191,182,182,182,182,182,128,255
        DEFB  251,219,219,219,219,219,3,255
; Духовые окна на крыше дома
SPR4    DEFB  4
        DEFB  0,0,65, 0,1,65, 1,0,65, 1,1,65
        DEFB  0,0,255,255,254,254,254,254
        DEFB  0,0,255,255,127,127,127,127
        DEFB  255,255,255,255,153,153,255,255
```

```
       DEFB  255,255,255,255,153,153,255,255
; Подвальные окна
SPR5   DEFB  2
       DEFB  0,0,65, 0,1,65
       DEFB  255,255,255,255,240,255,255,255
       DEFB  255,255,255,255,15,255,255,255
; Автомобиль
SPR6   DEFB  10
       DEFB  0,0,66, 0,1,66, 0,2,66, 0,3,66, 0,4,66
       DEFB  1,0,66, 1,1,66, 1,2,66, 1,3,66, 1,4,66
       DEFB  0,15,24,48,56,62,63,57
       DEFB  0,255,48,48,56,60,255,249
       DEFB  0,255,48,48,56,60,255,249
       DEFB  0,255,48,48,56,60,255,243
       DEFB  0,192,96,32,32,48,248,248
       DEFB  63,63,62,29,1,0,0,0
       DEFB  255,255,239,183,240,224,0,0
       DEFB  255,255,255,255,0,0,0,0
       DEFB  255,255,247,237,15,7,0,0
       DEFB  248,252,124,184,128,0,0,0
; Маска для спрайта «автомобиль»
SPR7   DEFB  10
       DEFB  0,0,66, 0,1,66, 0,2,66, 0,3,66, 0,4,66
       DEFB  1,0,66, 1,1,66, 1,2,66, 1,3,66, 1,4,66
       DEFB  224,192,128,131,128,128,128,128
       DEFB  0,0,0,135,129,0,0,0
       DEFB  0,0,0,135,129,0,0,0
       DEFB  0,0,0,135,129,0,0,0
       DEFB  31,15,15,15,135,3,3,3
       DEFB  128,128,128,128,192,252,254,255
       DEFB  0,0,0,0,0,7,15,255
       DEFB  0,0,0,0,0,255,255,255
       DEFB  0,0,0,0,0,224,240,255
       DEFB  1,1,1,1,3,63,127,255
; Фонарный столб
SPR8   DEFB  7
       DEFB  0,0,70, 0,1,70, 0,2,70, 1,1,70
       DEFB  2,1,70, 3,1,70, 4,1,70
       DEFB  0,0,127,64,63,0,0,0
       DEFB  0,0,239,146,17,16,16,16
       DEFB  0,0,252,4,248,0,0,0
       DEFB  16,16,16,16,16,16,16,16
       DEFB  16,16,16,16,16,16,16,16
       DEFB  16,16,16,56,56,56,56,56
       DEFB  56,56,56,56,56,56,56,56
; Маска для спрайта «фонарный столб»
SPR9   DEFB  7
       DEFB  0,0,70, 0,1,70, 0,2,70, 1,1,70
       DEFB  2,1,70, 3,1,70, 4,1,70
       DEFB  255,0,0,0,0,128,255,255
       DEFB  255,0,0,0,0,68,199,199
       DEFB  255,1,1,1,1,3,255,255
       DEFB  199,199,199,199,199,199,199,199
       DEFB  199,199,199,199,199,199,199,199
       DEFB  199,199,131,131,131,131,131,131
       DEFB  131,131,131,131,131,131,131,131
; Данные для формирования зданий:
; 1-й байт - количество блоков в доме
; Далее следуют данные для каждого блока
; 1-й байт: горизонтальная координата блока в доме относительно
;       верхнего левого угла изображения
; 2-й байт: вертикальная координата блока в доме
; 3-й и 4-й байты: адрес спрайта блока
D_BLD1 DEFB  25
       DEFW  #000,SPR4,#002,SPR4,#004,SPR4,#006,SPR4,#008,SPR4
```

```
       DEFW  #200,SPR3,#202,SPR2,#204,SPR1,#206,SPR2,#208,SPR3
       DEFW  #400,SPR3,#402,SPR2,#404,SPR1,#406,SPR2,#408,SPR3
       DEFW  #600,SPR1,#602,SPR2,#604,SPR1,#606,SPR1,#608,SPR1
       DEFW  #800,SPR5,#802,SPR5,#804,SPR5,#806,SPR5,#808,SPR5
D_BLD2 DEFB  15
       DEFW  #000,SPR3,#002,SPR1,#004,SPR2
       DEFW  #200,SPR3,#202,SPR1,#204,SPR2
       DEFW  #400,SPR3,#402,SPR1,#404,SPR2
       DEFW  #600,SPR1,#602,SPR1,#604,SPR2
       DEFW  #800,SPR5,#802,SPR5,#804,SPR5
; Данные для формирования пейзажа:
; 1-й и 2-й байты (слово): горизонтальная координата дома
; 3-й байт: протяженность дома в знакоместах
; 4-й байт: вертикальная координата дома
; 5-й и 6-й байты (слово): адрес данных для формирования дома
D_LAND DEFW  1,#10A,D_BLD1,14,#FE0A,D_BLD1,25,#FE0A,D_BLD1
       DEFW  39,#106,D_BLD2,46,#10A,D_BLD1,63,6,D_BLD2
       DEFW  70,10,D_BLD1,90,#10A,D_BLD1,101,10,D_BLD1
       DEFW  112,#FF0A,D_BLD1,123,#FE0A,D_BLD1,138,#106,D_BLD2
       DEFW  145,#FE06,D_BLD2,152,#FF06,D_BLD2,159,6,D_BLD2
       DEFW  166,#106,D_BLD2,178,#FE0A,D_BLD1,190,#FE06,D_BLD1
       DEFW  0
; Данные рамки вокруг окна (формат описан в предыдущем разделе)
DFRAME DEFB  0,3,0,0, 1,4,0,24, 2,28,0,0
       DEFB  3,3,1,12@#80, 3,28,1,12@128
       DEFB  5,3,13,0, 1,4,13,24, 4,28,13,0
       DEFB  -1
; Элементы рамки
D_SYMB DEFB  127,213,159,191,253,182,248,181
       DEFB  255,85,255,255,85,170,0,255
       DEFB  254,83,249,245,121,181,121,181
       DEFB  249,181,249,181,249,181,249,181
       DEFB  249,117,249,245,81,169,3,254
       DEFB  249,189,255,191,213,170,192,127
```

ГЛАВА ВОСЬМАЯ,

в которой рассказывается о том, как управлять ходом игры

Одним из главных достоинств компьютерных игр, как вы сами прекрасно знаете, является возможность играющего влиять на события, разворачивающиеся перед ним на экране. Это влияние можно реализовать по-разному, например, нажимая определенные клавиши клавиатуры или наклоняя в ту или иную сторону ручку джойстика и нажимая кнопку «огонь». При этом в большинстве игр предусматривается выбор любого вида управления, который обычно осуществляется через «Меню». Настало время и нам рассмотреть эти вопросы с разных точек зрения, а именно: как управлять спрайтами с помощью клавиатуры или джойстика и при этом учесть ограничения, обусловленные размерами экрана, как определить, подключен к компьютеру джойстик или нет, как организовать управление игрой, если играющих двое, и некоторые другие.

УПРАВЛЕНИЕ С ПОМОЩЬЮ КЛАВИАТУРЫ

При разработке игровых программ немыслимо обойтись без опроса клавиатуры. Действительно, чтобы во время игры управлять спрайтами, перемещая их хотя бы по четырем направлениям, программа обязана безошибочно различать одну из четырех нажатых клавиш. Например, **O** должна соответствовать движению влево, **P** - вправо, **Q** - вверх и **A** - вниз. В Бейсике, как вы помните, для этой цели мы пользовались функцией `INKEY$`, а одна из возможных программ, «узнающих», к примеру, символ **P**, могла бы выглядеть так:

```
100 IF INKEY$<>"P" THEN GO TO 100
```

Соответствующий ей фрагмент ассемблерной программы имеет следующий вид:

```
       XOR   A
       LD    (23560),A   ;в системную переменную LAST_K (код
                         ; последней нажатой клавиши) заносится 0
LOOP   LD    A,(23560)   ;из этой системной переменной
                         ; считывается значение кода нажатой клавиши
       CP    "P"         ;сравнение двух кодов - находящегося
                         ; в регистре A и символа P
       JR    NZ,LOOP     ;если результат сравнения не равен 0,
                         ; то переход на метку LOOP, если 0,
       RET               ; то выход
```

Надо заметить, что эта программка уже неоднократно применялась нами для разных целей, например, для выхода из циклов, для перехода к кадрам в многокадровой заставке, но она может использоваться и как образец для создания более сложных программ, например, следующей:

```
KEY    XOR   A
       LD    (23560),A
MET1   LD    A,(23560)
       CP    "P"         ;сравнение двух кодов
; Если результат сравнения не равен нулю (то есть нажата не P),
;  то переход на метку MET2, после которой проверяются нажатия других клавиш
       JR    NZ,MET2
       LD    DE,TXT1
PRINT  LD    BC,5        ;вывод на экран символа,
       CALL  8252        ; соответствующего нажатой клавише
       LD    A,13
       RST   16
       JR    KEY         ;переход на начало программы
MET2   CP    "O"         ;проверка нажатия клавиши O
       JR    NZ,MET3
       LD    DE,TXT2
       JR    PRINT
MET3   CP    "Q"         ;проверка нажатия клавиши Q
       JR    NZ,MET4
       LD    DE,TXT3
       JR    PRINT
MET4   CP    "A"         ;проверка нажатия клавиши A
       JR    NZ,MET5
       LD    DE,TXT4
       JR    PRINT
MET5   CP    "0"         ;проверка нажатия клавиши 0
       JR    NZ,MET1     ;если коды не совпадают,
                         ; повторяем все сначала
       RET               ; иначе - выход из программы
; Данные для печати
TXT1   DEFM "KEY P"
TXT2   DEFM "KEY O"
TXT3   DEFM "KEY Q"
TXT4   DEFM "KEY A"
```

После того как вы нажмете клавишу **P**, **O**, **Q** или **A**, программа напечатает в левом верхнем углу экрана одну из фраз, перечисленных в блоке данных, например, «KEY Q».

В игровых программах, как вы знаете, часто требуется опрашивать несколько клавиш одновременно, например, чтобы выполнять сложные перемещения спрайтов, включающие помимо вертикальных и горизонтальных еще и диагональные направления. Для подобных ситуаций приведенные выше способы опроса клавиатуры оказываются непригодными и, чтобы заставить все-таки спрайты перемещаться в любую сторону, придется воспользоваться командой IN, с помощью которой выполняется ввод данных из какого-либо порта. Существует несколько способов чтения из портов, но нас будут интересовать только два из них:

IN reg,(C) - ввод байта из порта и помещение его в регистр, причем reg - один из регистров A, B, C, D, E, H или L, а адрес порта содержится в паре BC (в C - младший байт адреса, в B - старший).

IN A,(port) - ввод байта из порта с номером port и помещение его в аккумулятор. При этом полный 16-разрядный адрес порта составляется из значения port (младший байт) и значения аккумулятора (старший байт).

Применяя одну или другую команду, можно получить два способа опроса клавиатуры. Первый из них очень похож на использование функции `IN` в Бейсике. Напомним, что все клавиши группируются по полурядам, то есть по 5 клавиш. Каждому полуряду соответствует определенный порт. Адреса «клавиатурных» портов отличаются только старшим байтом, а младший всегда равен 254 (#FE). Все эти адреса представлены в табл. 8.1 в десятичной, шестнадцатеричной и двоичной нотации. Предположим, что нам нужно определить нажатие клавиши **M**, тогда в регистровую пару BC необходимо записать адрес 32766. Из порта считываем значение для полуряда, а затем, чтобы определить нажатие конкретной клавиши, проверяем соответствующий ей бит (от бита 0 - для крайних клавиш до 4-го бита - для центральных). Так как клавиша **M** занимает третье место от края, то она будет определяться состоянием 2-го бита полученного из порта байта. Если этот бит окажется сброшенным в 0, это будет означать, что клавиша нажата. (Из-за упрощенной аппаратной реализации клавиатуры, примененной в ZX Spectrum, достоверно (в общем случае) можно определить одновременное нажатие не более двух каких-либо клавиш - *Примеч. ред.*)

Таблица 8.1. Адреса портов для опроса клавиатуры

Полуряд	DEC	HEX	BIN
Space...B	32766	7FFE	01111111 11111110
Enter...H	49150	BFFE	10111111 11111110
P...V	57342	DFFE	11011111 11111110
0...6	61438	EFFE	11101111 11111110
1...5	63486	F7FE	11110111 11111110
Q...T	64510	FBFE	11111011 11111110
A...G	65022	FDFE	11111101 11111110
CS...V	65278	FEFE	11111110 11111110

```
; Адрес 32766=127×256+254, в B заносится адрес полуряда,
;  а в C - адрес порта (254).
KEY     LD    BC,32766
        IN    A,(C)
; Один из способов проверки данного бита - у отпущенной клавиши
;  бит установлен (1), у нажатой сбрасывается в 0
        BIT   2,A
        JR    NZ,KEY
```

```
        RET
```

Второй способ принципиально не отличается от первого. Перед чтением в аккумулятор помещается старший байт адреса соответствующего порта, а младший байт задается в явном виде в команде IN:

```
KEY     LD    A,#7E       ;в аккумулятор заносится старший байт
                          ; адреса порта #7EFE
        IN    A,(254)     ;считывание из порта (254 или #FE -
                          ; младший байт адреса)
        BIT   2,A         ;проверка нажатия третьей от края
                          ; клавиши (M)
        JR    NZ,KEY
        RET
```

Рассмотрим программу, в которой при нажатии клавиш **Q**, **A**, **O** и **P** изменяются координаты точки на экране. Сами точки будем ставить в бейсик-программе, которую напишем позже, но подразумевая использование процедуры из Бейсика, воспользуемся для передачи координат точки, как и раньше, областью буфера принтера, определив адрес передаваемых параметров константой XY.

```
        ORG   60000
XY      EQU   23296
KEY     LD    HL,(XY)     ;запись координат точки в HL
; В регистр A заносится старший байт полуряда,
;  в котором располагается клавиша Q
        LD    A,251
        IN    A,(254)     ;читаем из порта значения для полуряда
; Проверка бита 0 (команду RRCA вместо BIT здесь удобнее применять
;  потому, что клавиша Q в полуряду занимает крайнее положение)
        RRCA
; Если клавиша не нажата (на что указывает установленный бит),
;  то следующую команду пропускаем
        JR    C,KEY1
; Увеличиваем значение вертикальной координаты, которое находится в регистре H
        INC   H
KEY1    LD    A,253
        IN    A,(254)
        RRCA              ;клавиша A
        JR    C,KEY2
        DEC   H           ;уменьшаем вертикальную координату
KEY2    LD    A,223
        IN    A,(254)
        RRCA              ;клавиша P
        JR    C,KEY3
        INC   L           ;увеличиваем горизонтальную координату
; Так как клавиши P и O находятся в одном полуряду,
;  то выполнять команду IN дважды нет необходимости
KEY3    RRCA              ;клавиша O
        JR    C,KEY4
        DEC   L           ;уменьшаем горизонтальную координату
KEY4    LD    (XY),HL
        LD    A,127
        IN    A,(254)
        BIT   2,A
        RET   NZ          ;выход, если клавиша M не нажата
        JP    3435        ; иначе очищаем экран
```

Чтобы увидеть эту процедуру в действии, необходимо дополнить ее небольшой бейсик-программкой, задача которой состоит только в том, чтобы ставить на экране точку в соответствии с координатами (XY), полученными в ассемблерной программе.

```
100 POKE 23296,100: POKE 23297,100
110 PLOT PEEK 23296, PEEK 23297
120 RANDOMIZE USR 60000: GO TO 110
```

Попробуйте ее ввести и исполнить, а затем понажимайте клавиши **Q**, **A**, **O** и **P** - по экрану в разных направлениях потянутся четкие прямые линии подобно использованию функции PEN в графическом редакторе. Нажав клавишу **M**, в любой момент можно очистить экран и начать рисовать новую «картину».

В заключение этого раздела приведем еще один пример управления с помощью клавиатуры, с которым мы иногда встречаемся, загружая те или иные игровые программы. Он полезен еще и тем, что дает вариант решения некоторых побочных проблем, таких, например, как учет ограничений на перемещение курсора (или спрайта) по экрану, введение дополнительных функций управления и некоторые другие.

Рис. 8.1. Ввод имени играющего

Представим себе, что в конце игры необходимо набрать имя играющего, чтобы затем записать его в раздел меню HI SCORE. Для этого, при достижении определенных результатов, вызывается кадр, в котором вы видите примерно такую таблицу, какая изображена на рис. 8.1. Далее, управляя курсором с помощью клавиш **Q**, **A**, **O** и **P**, требуется выбрать из таблицы буквы вашего имени, нажимая после каждой клавишу выбора **M**. При этом набранные буквы из таблицы будут переноситься в строку, расположенную ниже. Если какой-то символ набран неверно, его можно стереть, «нажав» в таблице букву d (delete), для печати пробела используется буква s (space), а для ввода имени и завершения этой части программы - буква e (enter). Надо сказать, что такой способ ввода имени не самый удобный, однако он имеет право на существование в случаях, когда играющий еще плохо знаком с клавиатурой ZX Spectrum, но имеет некоторое представление о латинском алфавите.

```
ORG   60000
ENT   $
XOR   A
CALL  8859
LD    A,68
LD    (23693),A
CALL  3435
```

```
       LD    A,2
       CALL  5633
; Очистка строки для ввода имени
       LD    HL,NAME
       LD    DE,NAME+1
       LD    BC,19
       LD    (HL)," "
       LDIR
; Вывод таблицы символов в рамке
       CALL  TABL
       CALL  LINES
       LD    A,68
       LD    (23693),A
       LD    BC,#506     ;начальные координаты курсора в таблице
       LD    E,0         ;номер символа в строке ввода
       SET   3,(IY+48)   ;режим ввода прописных букв
; Управление курсором и печать выбранного символа в строку
KEYS   CALL  SETCUR      ;вывод курсора
       XOR   A
       LD    (23560),A
WAIT   LD    A,(23560)   ;ожидание нажатия клавиши
       AND   A
       JR    Z,WAIT
       CP    "P"         ;перемещение курсора на
       JR    Z,RIGHT     ; один шаг вправо
       CP    "O"         ;перемещение курсора
       JR    Z,LEFT      ; на один шаг влево
       CP    "Q"         ;перемещение курсора
       JR    Z,UP        ; на один шаг вверх
       CP    "A"         ;перемещение курсора
       JR    Z,DOWN      ; на один шаг вниз
       CP    "M"         ;печать выбранного символа
       JR    Z,SELECT    ; в строке ввода
       JR    KEYS
; Перемещение курсора вправо
RIGHT  LD    A,C         ;проверка достижения курсором
       CP    24          ; правой границы таблицы
       JR    NC,KEYS
       CALL  RESCUR      ;удаление курсора на прежнем месте
       INC   C           ;изменение положения курсора
       INC   C
       CALL  SETCUR      ;установка курсора на букву таблицы
       JR    KEYS
; Перемещение курсора влево
LEFT   LD    A,C         ;проверка достижения курсором
       CP    7           ; левой границы таблицы
       JR    C,KEYS
       CALL  RESCUR
       DEC   C
       DEC   C
       CALL  SETCUR
       JR    KEYS
; Перемещение курсора вверх
UP     LD    A,B         ;проверка достижения курсором
       CP    6           ; верхней границы таблицы
       JR    C,KEYS
       CALL  RESCUR
       DEC   B
       DEC   B
       CALL  SETCUR
       JR    KEYS
; Перемещение курсора вниз
DOWN   LD    A,B         ;проверка достижения курсором
       CP    11          ; нижней границы таблицы
       JR    NC,KEYS
```

```
        CALL  RESCUR
        INC   B
        INC   B
        CALL  SETCUR
        JR    KEYS
; Выбор символа, который затем будет напечатан в строке или выбор
;  функции для редактирования этой строки
SELECT PUSH  BC
        PUSH  DE
        CALL  SND          ;звуковой сигнал, издаваемый при
                           ; перемещении символа из таблицы в
                           ; набираемую строку
        POP   DE
        POP   BC
        LD    A,B
        CP    11
        JR    NZ,MOVE      ;печать символа
        LD    A,C
        CP    20
        JR    Z,DELETE     ;удаление символа в строке
        CP    22
        JR    Z,SPACE      ;печать пробела в строке
        CP    24
        RET   Z            ;выход из программы
; Перемещаем символ из таблицы в набираемую строку и смещаем курсор
;  на позицию вправо, при этом делаем проверку того, чтобы символ
;  не вышел за заданные границы строки (слева и справа).
MOVE    LD    A,E
        CP    20
        JP    NC,KEYS
        LD    D,0
        PUSH  BC
        PUSH  DE
        LD    A,B          ;по вертикальной координате курсора
                           ; определяем адрес данных строки
                           ; таблицы (STR1, STR2, STR3 или STR4)
        SUB   5
        LD    HL,D_STR
        LD    E,A
        ADD   HL,DE
        LD    E,(HL)
        INC   HL
        LD    D,(HL)
        EX    DE,HL
        LD    A,C          ;по горизонтальной координате находим
                           ; код символа в блоке данных
        SUB   6
        LD    C,A
        LD    B,0
        ADD   HL,BC
        POP   DE
        POP   BC
        LD    A,(HL)       ;помещаем код символа в A
        LD    HL,NAME      ;определяем адрес в строке NAME
        ADD   HL,DE        ; для ввода символа
        LD    (HL),A       ;помещаем символ в строку ввода
        CALL  PR_STR       ;выводим строку ввода на экран
        INC   E            ;смещаем позицию ввода вперед
        JP    KEYS
; Удаление неправильно набранного символа
DELETE LD    A,E          ;проверка достижения начала строки ввода
        AND   A
        JP    Z,KEYS
        DEC   E            ;уменьшаем позицию ввода
        LD    D,0
```

```
       LD    HL,NAME
       ADD   HL,DE
       LD    (HL)," "     ;заменяем удаляемый символ пробелом
       CALL  PR_STR
       JP    KEYS
; Ввод пробела
SPACE  LD    A,E          ;проверка достижения конца строки ввода
       CP    20
       JP    NC,KEYS
       LD    D,0
       LD    HL,NAME
       ADD   HL,DE
       LD    (HL)," "
       CALL  PR_STR
       INC   E            ;увеличиваем позицию ввода
       JP    KEYS
; Вывод курсора изменением байта атрибутов
RESCUR LD    A,68         ;PAPER 0, INK 4, BRIGHT 1
       JR    PRATTR
; Удаление курсора восстановлением байта атрибутов
SETCUR LD    A,79         ;PAPER 1, INK 7, BRIGHT 1
; Вычисляем адрес атрибутов знакоместа и заносим
;  по этому адресу байт из аккумулятора
PRATTR LD    L,B
       LD    H,0
       ADD   HL,HL
       ADD   HL,HL
       ADD   HL,HL
       ADD   HL,HL
       ADD   HL,HL
       PUSH  AF
       LD    A,H
       ADD   A,#58
       LD    H,A
       LD    A,L
       ADD   A,C
       LD    L,A
       POP   AF
       LD    (HL),A
       RET
; Подпрограмма печати таблицы символов
TABL   LD    DE,STR
       LD    BC,LENSTR
       JP    8252
; Подпрограмма печати введенной строки
PR_STR PUSH  BC
       PUSH  DE
       LD    DE,STR5
       LD    BC,LENLIN
       CALL  8252
       POP   DE
       POP   BC
       RET
; Подпрограмма рисования рамки
LINES  EXX
       PUSH  HL
       LD    A,66
       LD    (23695),A
       LD    BC,#8A2C     ;B = 138, C = 44
       CALL  8933
       LD    DE,#101
       LD    BC,160       ;B = 0, C = 160
       CALL  9402
       LD    DE,#FF01
       LD    BC,#3D00     ;B = 61, C = 0
```

```
        CALL  9402
        LD    DE,#1FF
        LD    BC,160
        CALL  9402
        LD    DE,#101
        LD    BC,#3D00
        CALL  9402
        POP   HL
        EXX
        RET
; Короткий звуковой сигнал
SND     LD    B,30
        LD    HL,350
        LD    DE,2
SND1    PUSH  BC
        PUSH  DE
        PUSH  HL
        CALL  949
        POP   HL
        POP   DE
        POP   BC
        SBC   HL,DE
        DJNZ  SND1
        RET
; Данные таблицы символов
STR     DEFB  22,5,6
STR1    DEFM  "1 2 3 4 5 6 7 8 9 0" ;символы через один пробел
        DEFB  22,7,6
STR2    DEFM  "A B C D E F G H I J"
        DEFB  22,9,6
STR3    DEFM  "K L M N O P Q R S T"
        DEFB  22,11,6
STR4    DEFM  "U V W X Y Z . d s e"
STR5    DEFB  22,19,5,16,5,">",16,2
NAME    DEFM  "....................."
        DEFB  16,5,"<"
LENSTR  EQU   $-STR       ;длина строки для печати таблицы
LENLIN  EQU   $-STR5      ;длина строки ввода имени
; Адреса данных символов в таблице
D_STR   DEFW  STR1,STR2,STR3,STR4
```

Эту программу можно рассматривать как вполне независимый кадр заставки. Если вы решите использовать ее в своей собственной игре, единственное, что вам потребуется, это перенести после выхода введенное имя из строки NAME в таблицу «рекордов». И, конечно же, вам нужно будет проследить, чтобы имена меток, задействованные в приведенной программе не повторялись в вашей игре. Естественно, что при необходимости наименования меток можно и заменить.

УПРАВЛЕНИЕ ДЖОЙСТИКОМ

В динамичных играх управление спрайтами с помощью Kempston-джойстика часто оказывается более удобным, чем от клавиатуры - нужно помнить всего лишь о четырех сторонах света, да в пылу борьбы не забыть, что есть еще кнопка «огонь».

Наверное, вам известно, что в Бейсике определить положение ручки джойстика можно с помощью функции `IN 31`. В ассемблере для этих же целей удобнее всего применять команду IN A,(31), после выполнения которой в аккумуляторе появится некоторое число, отдельные биты которого и определяют «статус» джойстика. Значения имеют не все биты, а только 5

младших, причем, в отличие от клавиатуры, в нейтральном положении все биты сброшены в 0 (конечно, если порт джойстика вообще подключен), а установка какого-то бита в 1 означает поворот ручки или нажатие кнопки «огонь». При диагональном наклоне ручки будут установлены сразу два бита. В табл. 8.2 показано соответствие пяти младших битов, получаемых в аккумуляторе после выполнения команды IN A,(31), направлениям наклона ручки джойстика и нажатию кнопки «огонь». Три старших бита не определены, поэтому в таблице они заменены знаками вопроса.

Таблица 8.2. Значения битов порта джойстика.

Направление	**Код**
Вправо	???00001
Влево	???00010
Вверх	???00100
Вниз	???01000
Огонь	???10000

Давайте сначала на примере простой программки перемещения точки, аналогичной той, которую мы описали в начале первого раздела данной главы, рассмотрим принцип использования джойстика. Причем, как и в случае с клавиатурой, точку на экран будем ставить с помощью бейсик-программы.

```
        ORG   60000
XY      EQU   23296
JOY     LD    HL,(XY)      ;в регистре H вертикальная координата,
                           ; а в L - горизонтальная
        IN    A,(31)       ;читаем из порта джойстика
        RRCA               ;проверяем бит 0
        JR    NC,JOY1      ;если в 0, переходим к проверке
                           ; следующего бита
        INC   L            ;увеличиваем горизонтальную координату
JOY1    RRCA               ;аналогично проверяем остальные биты
        JR    NC,JOY2
        DEC   L            ;уменьшаем горизонтальную координату
JOY2    RRCA
        JR    NC,JOY3
        DEC   H            ;увеличиваем вертикальную координату
JOY3    RRCA
        JR    NC,JOY4
        INC   H            ;уменьшаем вертикальную координату
JOY4    LD    (XY),HL      ;новые координаты передаем
                           ; бейсик-программе
        RRCA
        RET   NC
        JP    3435         ;при нажатии кнопки «огонь»
                           ; экран очищается
```

Чтобы точка, перемещаясь по экрану, не ушла за его пределы, необходимо ввести ограничения. Как это осуществить, покажем на конкретном примере, в котором мы используем приведенный выше принцип для воспроизведения реальной игровой ситуации, например, для управления самолетом. Прежде всего создадим три спрайта (блоки SAM1, SAM2 и SAM3), первый из которых будет соответствовать полету самолета прямо, второй повороту вправо (самолет при этом должен слегка наклониться) и, наконец, третий - его повороту влево. Таким образом, наклоняя ручку джойстика в ту или иную сторону, вы можете в приведенной ниже программе легко изменять положение спрайта на экране.

```
        ORG   60000
        ENT   $
        LD    A,5
        LD    (23693),A
        XOR   A
        CALL  8859
        CALL  3435
; Основная часть программы
        LD    BC,#505       ;в регистре B вертикальная координата Y,
                            ; а в C горизонтальная
JOY     IN    A,(31)        ;читаем данные из порта джойстика
        LD    E,A           ;освобождаем аккумулятор
                            ; для проверки границ
        LD    HL,SAM1       ;задаем адрес первого спрайта
        RRC   E             ;сдвигаем E вправо на один бит
        JR    NC,JOY1
        LD    A,C
        CP    30            ;задаем границу перемещения вправо
        JR    NC,JOY1       ;если правая граница достигнута,
                            ; то увеличивать X уже нельзя -
                            ; переходим на метку JOY1
        INC   C             ;увеличиваем координату X, что соответствует
                            ; перемещению самолета вправо
        LD    HL,SAM2       ;задаем адрес второго спрайта
JOY1    RRC   E
        JR    NC,JOY2
        LD    A,C
        CP    1
        JP    M,JOY2
        DEC   C             ;уменьшаем координату X
        LD    HL,SAM3       ;задаем адрес третьего спрайта
JOY2    RRC   E
        JR    NC,JOY3
        LD    A,B
        CP    22
        JR    NC,JOY3
        INC   B             ;увеличиваем координату Y
JOY3    RRC   E
        JR    NC,JOY4
        LD    A,B
        CP    1
        JP    M,JOY4
        DEC   B             ;уменьшаем координату Y
JOY4    RRC   E
        RET   C             ;если кнопка «огонь» нажата -
                            ; выходим из программы
; Вывод на экран по принципу XOR одного из трех спрайтов самолета
        PUSH  BC
        PUSH  HL
        LD    A,SPRXOR      ;устанавливаем режим вывода XOR
        CALL  PTBL          ;печатаем один из самолетов
        LD    BC,10         ;вводим задержку
        CALL  7997
        POP   HL
        POP   BC
        PUSH  BC
        LD    A,SPRXOR      ;устанавливаем режим вывода XOR
        CALL  PTBL          ;стираем изображение самолета
        POP   BC
        JR    JOY           ;переходим в начало программы
                            ; для изменения координат
PTBL    .........
; Заголовок данных спрайта первого самолета, соответствующего полету вперед и
назад
```

```
SAM1    DEFB  4
        DEFB  0,0,5,0,1,5,1,0,5,1,1,5
; Данные для первого самолета
        DEFB  0,0,5,0,95,254,56,66
        DEFB  0,0,160,0,250,127,28,66
        DEFB  61,1,1,5,13,0,0,0
        DEFB  188,128,128,160,176,0,0,0
; Заголовок данных спрайта второго самолета, соответствующего повороту вправо
SAM2    DEFB  4
        DEFB  0,0,5,0,1,5,1,0,5,1,1,5
; Данные для второго самолета
        DEFB  0,0,5,0,11,95,78,35
        DEFB  0,0,160,0,244,62,28,8
        DEFB  29,1,1,2,2,0,0,0
        DEFB  176,128,128,192,224,192,0,0
; Заголовок данных спрайта третьего самолета, соответствующего повороту влево
SAM3    DEFB  4
        DEFB  0,0,5,0,1,5,1,0,5,1,1,5
; Данные для третьего самолета
        DEFB  0,0,5,0,47,124,56,16
        DEFB  0,0,160,0,208,250,114,196
        DEFB  13,1,1,3,7,3,0,0
        DEFB  184,128,128,64,64,0,0,0
```

СОВМЕСТНОЕ УПРАВЛЕНИЕ КЛАВИАТУРОЙ И ДЖОЙСТИКОМ

Прочитав название параграфа, многие наверняка подумали - а в чем тут собственно проблема, достаточно объединить вместе блоки управления клавиатурой и джойстиком и вроде бы все, можно пользоваться как тем так и другим. Не будем вас разочаровывать, так оно и есть, и в принципе подобную схему вполне можно использовать для совместного управления. Но при этом нужно помнить о том, что если потребуется изменить управляющие клавиши, вам придется вносить в текст программы довольно много изменений. То же самое можно сказать и о смене типа джойстика. Поэтому хотелось бы иметь универсальную процедуру опроса клавиатуры и джойстика, в которой перечисленные изменения можно было выполнить с наименьшей затратой сил. Ниже приводится программа, использующая подобную процедуру, обозначенную меткой KBDJOY. В ней на экран выводится вертолет с вращающимся винтом (рис. 8.2 *а*,*б*) и стрекочущим двигателем. Нажимая клавиши **Q**, **A**, **O**, **P** или наклоняя ручку джойстика, вы сможете легко убедиться в том, что программа работает как положено. А для усиления эффекта можно попробовать нажать какую-нибудь из клавиш и одновременно повернуть джойстик - вертолет послушно полетит по диагонали.

а б

Рис. 8.2. Спрайт «вертолет»

```
        ORG   60000
        ENT   $
        LD    A,5
        LD    (23693),A
        XOR   A
        CALL  8859
        CALL  3435
; Основная часть программы
        LD    BC,#505     ;задаем исходное положение вертолета
KEY     PUSH  BC
```

```
        CALL  KBDJOY      ;читаем данные из портов
        POP   BC
        RRCA              ;поворачиваем ручку джойстика вправо
                          ; или нажимаем клавишу P -
                          ; полет вертолета вправо
        JR    NC,KEY1
        INC   C
KEY1    RRCA              ;поворачиваем ручку джойстика влево
                          ; или нажимаем клавишу O -
                          ; полет вертолета влево
        JR    NC,KEY2
        DEC   C
KEY2    RRCA              ;поворачиваем ручку джойстика вниз
                          ; или нажимаем клавишу A -
                          ; полет вертолета вниз
        JR    NC,KEY3
        INC   B
KEY3    RRCA              ;поворачиваем ручку джойстика вверх
                          ; или нажимаем клавишу Q -
                          ; полет вертолета вверх
        JR    NC,KEY4
        DEC   B
KEY4    RRCA              ;при нажатии кнопки «огонь» джойстика
                          ; или клавиши M - выход
        RET   C
; Подпрограмма вывода на экран изображения вертолета в двух фазах,
;  каждая из которых соответствует одному из положений винта
        XOR   A           ;формируем звуковой сигнал,
        OUT   (254),A     ; имитирующий работу двигателя
        CALL  CHECK       ;проверка достижения границ экрана
        LD    A,16
        OUT   (254),A
        LD    A,SPRXOR    ;задаем режим вывода спрайта
        LD    HL,WERT1    ;устанавливаем адрес спрайта
        PUSH  BC
        PUSH  HL
        CALL  PTBL        ;выводим вертолет в первой фазе
        LD    BC,5        ;задаем задержку между фазами
        CALL  7997        ; вращения винта
        POP   HL
        POP   BC
        LD    A,SPRXOR    ;режим вывода спрайта
        PUSH  BC
        PUSH  HL
        CALL  PTBL        ;стираем вертолет в первой фазе
        POP   HL
        POP   BC
        XOR   A
        OUT   (254),A
; Вывод вертолета во второй фазе
        LD    A,16        ;звуковой сигнал
        OUT   (254),A
        LD    A,SPRXOR    ;режим вывода спрайта
        LD    HL,WERT2    ;устанавливаем адрес спрайта
                          ; с другим расположением винта
        PUSH  BC
        PUSH  HL
        CALL  PTBL        ;выводим спрайт во второй фазе
        LD    BC,5
        CALL  7997
        POP   HL
        POP   BC
        LD    A,SPRXOR    ;режим вывода спрайта
        PUSH  BC
        PUSH  HL
```

```
        CALL   PTBL         ;стираем с экрана спрайт во второй фазе
        POP    HL
        POP    BC
        JR     KEY
; Подпрограмма проверки границ экрана
CHECK   LD     A,C
        AND    A            ;сравниваем координату X вертолета
                            ; с заданной левой границей экрана
        JR     NZ,CONT1
        INC    C
CONT1   CP     29           ;сравниваем координату X вертолета
                            ; с заданной правой границей экрана
        JR     NZ,CONT2
        DEC    C
CONT2   LD     A,B          ;задаем верхнюю границу экрана
        AND    A            ;сравниваем координату Y вертолета
                            ; с заданной верхней границей экрана
        JR     NZ,CONT3
        INC    B
CONT3   CP     21           ;сравниваем координату Y вертолета
                            ; с заданной нижней границей экрана
        RET    NZ
        DEC    B
        RET
; Подпрограмма чтения данных из портов клавиатуры и джойстика
KBDJOY  IN     A,(31)       ;опрашиваем порт джойстика
        LD     E,A          ;запоминаем полученные биты
; Проверяем, подключен ли порт джойстика (ручку невозможно
;  повернуть сразу и вправо и влево - если оба бита установлены,
;  порт не подключен)
        AND    3
        CP     3
        JR     NZ,KBDJ1     ;если да, переходим к опросу клавиатуры
        LD     E,0          ; иначе очищаем коллектор битов
KBDJ1   LD     HL,DKEY      ;адрес блока данных клавиатуры
KBDJ2   LD     C,(HL)       ;младший байт адреса порта
        INC    C            ;проверка на 0 (конец блока данных)
        DEC    C
        LD     A,E          ;значение коллектора в аккумулятор
        RET    Z            ;выход, если конец данных
        INC    HL
        LD     B,(HL)       ;старший байт адреса порта
        INC    HL
        IN     A,(C)        ;читаем из порта
        CPL                 ;инвертируем биты
        AND    (HL)         ;проверяем конкретный бит
        INC    HL
        JR     Z,KBDJ3
        LD     A,(HL)       ;если бит установлен, читаем код направления
        OR     E            ; и объединяем с коллектором
        LD     E,A
KBDJ3   INC    HL
        JR     KBDJ2        ;продолжаем чтение
; Данные управляющих клавиш:
; первое число - младший байт порта
; второе число - старший байт порта
; третье число - маска бита
; четвертое    - код направления (аналогично кодам джойстика)
DKEY    DEFB   #FE,#FB,1,8  ;Q - вверх
        DEFB   #FE,#FD,1,4  ;A - вниз
        DEFB   #FE,#DF,2,2  ;O - влево
        DEFB   #FE,#DF,1,1  ;P - вправо
        DEFB   #FE,#7F,4,16 ;M - «огонь»
        DEFB   0            ;метка конца блока данных
PTBL    .........
```

```
; Заголовок первой фазы спрайта «вертолет»
WERT1  DEFB  7
       DEFB  0,1,6,1,0,6,1,1,6,1,2,6
       DEFB  2,0,6,2,1,6,2,2,6
; Данные первой фазы спрайта «вертолет»
       DEFB  0,0,4,28,56,32,24,24
       DEFB  0,0,0,1,2,2,4,4
       DEFB  60,60,255,129,66,36,36,24
       DEFB  0,0,0,128,64,64,32,32
       DEFB  7,2,1,0,1,2,4,14
       DEFB  255,36,36,255,0,0,0,0
       DEFB  224,64,128,0,128,64,32,112
; Заголовок второй фазы спрайта «вертолет»
WERT2  DEFB  9
       DEFB  0,0,6,0,1,6,0,2,6
       DEFB  1,0,6,1,1,6,1,2,6
       DEFB  2,0,6,2,1,6,2,2,6
; Данные второй фазы спрайта «вертолет»
       DEFB  0,0,30,127,255,127,14,0
       DEFB  0,0,0,129,231,129,24,24
       DEFB  0,0,120,254,255,254,112,0
       DEFB  0,0,0,1,2,2,4,4
       DEFB  60,60,255,129,66,36,36,24
       DEFB  0,0,0,128,64,64,32,32
       DEFB  7,2,1,0,1,2,4,14
       DEFB  255,36,36,255,0,0,0,0
       DEFB  224,64,128,0,128,64,32,112
```

ГЛАВА ДЕВЯТАЯ,

из которой вы узнаете, как подсчитать число заработанных очков и вообще оценить состояние игры

В этой главе мы коснемся наиболее часто встречающихся типов оценки игровой ситуации, таких как подсчет очков (жизней, боеприпасов, сбитых самолетов) и контроля времени, а также рассмотрим некоторые приемы их применения на конкретных примерах игровых программ. Поскольку получение оценки немыслимо без различных математических действий, то здесь же приводятся процедуры умножения, деления, извлечения квадратного корня как для целых чисел с учетом знака, так и для дробных. В последнем случае уже не обойтись без обращения к калькулятору.

Сразу скажем, что оценка игровой ситуации не сводится к одним лишь только математическим расчетам. На самом деле не вся оценочная информация может выводиться на экран в виде чисел, а кое-что остается, так сказать, для «внутреннего пользования» самой программе, которая следит за развитием событий и соответствующим образом себя ведет. Например, в игре Tetris при завершении очередного ряда он должен автоматически удаляться, а все ряды выше «списанного» опускаться вниз. Это можно назвать уже не количественной, а качественной оценкой, используемой самой программой.

ПОДСЧЕТ КОЛИЧЕСТВЕННЫХ ВЕЛИЧИН

Покажем в начале, каким образом реализовать в игровой программе наиболее простые виды оценок, например, подсчет количества очков, контроль числа оставшихся жизней, количество попаданий в противника и так далее, в основе которых лежит, в общем-то, простая операция сложения чисел. До начала подсчета числа очков, следует обнулить переменную SUM, которая будет выполнять функции счетчика:

```
        XOR   A
        LD    (SUM),A
        ..............
```

Если же контролируется количество «жизней», то следует наоборот занести в SUM какое-то начальное значение:

```
        LD    A,10
        LD    (SUM),A
        ..............
```

Затем эта переменная будет изменяться в блоке оценки игровой ситуации, увеличиваясь или уменьшаясь в зависимости от ее типа:

```
        ............
        LD    A,(SUM)
        INC   A             ;или DEC A
        LD    (SUM),A
        CALL  PRINT         ;вывод числовой оценки на экран
        ............
SUM     DEFB  0             ;переменная для накопления суммы
```

Однако все так просто лишь до тех пор, пока подсчеты не требуют разного рода дополнительных проверок. Более сложным и интересным является случай, когда одновременно с подсчетами, выполняется еще и анализ игровой ситуации, а результаты этого анализа оказывают влияние на сами оценки.

Для иллюстрации рассмотрим программу МИШЕНЬ, которая уже использовалась нами в пятой главе для демонстрации случайных чисел. Дополним теперь эту программу некоторыми оценками, например, после каждого выстрела будем суммировать набранные очки, выводить общее количество произведенных выстрелов и, наконец, подсчитаем средний балл, полученный за один выстрел.

Но прежде приведем несколько процедур для выполнения арифметических действий с целыми числами. В них реализованы интересные только для математиков алгоритмы вычислений и больше ничего, поэтому здесь мы обойдемся без пояснений и предлагаем вам эти процедуры в качестве стандартных библиотечных функций.

Мы уже упоминали подпрограмму ПЗУ, расположенную по адресу 12457, выполняющую умножение двух целых чисел, находящихся в регистровых парах HL и DE. Ниже показана аналогичная процедура, отличающаяся только тем, что при умножении учитываются знаки сомножителей, заданных также в HL и DE. То есть числа, участвующие в операции могут находиться в пределах от -32768 до +32767. Произведение возвращается в регистровой паре HL.

```
MULT    LD    B,8
        LD    A,D
        AND   A
        JR    Z,MULT1
        RLC   B
MULT1   LD    C,D
        LD    A,E
        EX    DE,HL
        LD    HL,0
```

```
MULT2  SRL   C
       RRA
       JR    NC,MULT3
       ADD   HL,DE
MULT3  EX    DE,HL
       ADD   HL,HL
       EX    DE,HL
       DJNZ  MULT2
       RET
```

Следующая подпрограмма предназначена для деления знаковых величин. Делимое перед обращением к ней должно находиться в паре HL, а делитель - в DE. Напоминаем, что деление на 0 невозможно, поэтому в программе имеет смысл выполнять подобную проверку. Если такая ошибка все же произойдет, будет выдано сообщение Бейсика Number too big. После выполнения процедуры частное окажется в паре HL, а остаток от деления будет отброшен.

```
DIVIS  LD    A,D
       OR    E
       JR    NZ,DIVIS1
       RST   8
       DEFB  5            ;Number too big
DIVIS1 CALL  DIVIS5
       PUSH  BC
       LD    C,E
       LD    B,D
       LD    DE,0
       PUSH  DE
       EX    DE,HL
       INC   HL
DIVIS2 ADD   HL,HL
       EX    DE,HL
       ADD   HL,HL
       LD    A,C
       SUB   L
       LD    A,B
       SBC   A,H
       EX    DE,HL
       JR    NC,DIVIS2
       EX    DE,HL
DIVIS3 EX    DE,HL
       XOR   A
       LD    A,H
       RRA
       LD    H,A
       LD    A,L
       RRA
       LD    L,A
       OR    H
       JR    Z,DIVIS4
       EX    DE,HL
       XOR   A
       RR    H
       RR    L
       LD    A,C
       SUB   L
       LD    A,B
       SBC   A,H
       JP    M,DIVIS3
       LD    A,C
       SUB   L
       LD    C,A
       LD    A,B
       SBC   A,H
       LD    B,A
       EX    (SP),HL
```

```
        ADD   HL,DE
        EX    (SP),HL
        JR    DIVIS3
DIVIS4  POP   HL
        POP   BC
        BIT   7,B
        JR    NZ,MINUS
        RET
DIVIS5  LD    B,H
        LD    A,H
        RLA
        CALL  C,MINUS
        EX    DE,HL
        LD    A,H
        XOR   B
        LD    B,A
        BIT   7,H
        RET   Z
MINUS   LD    A,H
        CPL
        LD    H,A
        LD    A,L
        CPL
        LD    L,A
        INC   HL
        RET
```

Как вы помните, извлечение квадратного корня из отрицательного числа невозможно, результат также не может быть меньше нуля, поэтому и соответствующая процедура работает с величинами из диапазона 0...65535. Перед обращением к ней число, из которого нужно извлечь корень, поместите в пару HL. Результат, как и в предыдущих подпрограммах, будет возвращен в HL. Дробная часть при этом, к сожалению, также теряется.

```
SQR     LD    A,L
        LD    L,H
        LD    H,0
        LD    DE,64
        LD    B,8
SQR1    SBC   HL,DE
        JR    NC,SQR2
        ADD   HL,DE
SQR2    CCF
        RL    D
        ADD   A,A
        ADC   HL,HL
        ADD   A,A
        ADC   HL,HL
        DJNZ  SQR1
        LD    L,D
        LD    H,A
        RET
```

А теперь приводим модифицированный текст программы МИШЕНЬ, в которой присутствуют некоторые типы оценок, а также демонстрируется применение только что предложенных математических процедур:

```
        ORG   60000
        ENT   $
        LD    A,7
        LD    (23693),A
        XOR   A
        CALL  8859
        CALL  3435
        LD    A,2
        CALL  5633
```

```
;Основная часть программы МИШЕНЬ
       LD    HL,UDG
       LD    (23675),HL
       LD    HL,0
       LD    (SCORE),HL  ;обнуление счетчика количества очков,
                         ; заработанных при стрельбе по «мишени»
       XOR   A
       LD    (KOL_W),A   ;обнуление счетчика числа выстрелов,
                         ; произведенных по «мишени»
       CALL  MISH        ;рисование «мишени»
MAIN   CALL  WAIT        ;ожидание нажатия любой клавиши
       LD    A,22
       RST   16
       LD    E,19        ;диапазон изменения координаты Y
                         ; для пулевого отверстия
       CALL  RND         ;задаем координату Y
       LD    (ROW),A     ;заносим ее в переменную ROW
       RST   16
       LD    E,30        ;диапазон изменения координаты X
                         ; для пулевого отверстия
       CALL  RND         ;задаем координату X
       LD    (COL),A     ;заносим ее в переменную COL
       RST   16
       LD    A,16
       RST   16
       LD    A,6         ;задаем цвет пулевых отверстий
       RST   16
       LD    E,3         ;количество видов пулевых отверстий
       CALL  RND
       ADD   A,144
       RST   16
       CALL  SND         ;звуковой сигнал, имитирующий полет пули
       CALL  OCENKA      ;вычисление оценок результата стрельбы
                         ; и вывод их на экран
       LD    A,(23560)   ;выход из программы
       CP    " "
       JR    NZ,MAIN
       RET
; Подпрограмма оценки результата стрельбы
OCENKA LD    BC,(COL)    ;заносим в BC координаты выстрела
       LD    A,10        ;вертикальная координата центра «мишени»
       CP    B
       JR    C,BOT_Y
       SUB   B           ;пулевое отверстие находится в верхней
                         ; половине «мишени»
       JR    CONT1
BOT_Y  LD    A,B
       SUB   10          ;пулевое отверстие находится в нижней
                         ; половине «мишени»
CONT1  LD    B,A         ;в регистре B длина катета по Y
       LD    A,15        ;горизонтальная координата центра «мишени»
       CP    C
       JR    C,BIG_X
       SUB   C
       JR    CONT2
BIG_X  LD    A,C
       SUB   15
CONT2  LD    C,A         ;в регистре C длина катета по X
; Определяем длину гипотенузы прямоугольного треугольника
       LD    H,0
       LD    L,B
       LD    D,H
       LD    E,L
       PUSH  BC
       CALL  MULT        ;вычисляем квадрат величины Y
```

```
        LD    B,H
        LD    C,L
        POP   HL
        LD    H,0
        LD    D,H
        LD    E,L
        PUSH  BC
        CALL  MULT          ;вычисляем квадрат величины X
        POP   BC
        ADD   HL,BC
        CALL  SQR           ;определяем длину гипотенузы,
                            ; величину которой помещаем в пару HL
; По длине гипотенузы находим количество заработанных очков
;  в результате одного выстрела
        LD    C,0
        LD    A,L
        CP    11
        JR    NC,SUM
        LD    DE,D_SUM
        ADD   HL,DE
        LD    C,(HL)
; Вычисление трех оценочных характеристик стрельбы и их вывод на экран
SUM     PUSH  BC
        LD    DE,TXT1
        LD    BC,11
        CALL  8252
        POP   BC
        LD    HL,(SCORE)
        LD    B,0
        ADD   HL,BC
        LD    (SCORE),HL
        LD    B,H
        LD    C,L
        CALL  11563
        CALL  11747         ;печать общего количества заработанных
                            ; в результате стрельбы очков
        LD    DE,TXT2
        LD    BC,11
        CALL  8252
        LD    HL,KOL_W
        INC   (HL)
        LD    C,(HL)
        LD    B,0
        CALL  11563
        CALL  11747         ;печать количества произведенных
                            ; по «мишени» выстрелов
        LD    DE,TXT3
        LD    BC,10
        CALL  8252
        LD    HL,(SCORE)
        LD    D,H
        LD    E,L
        LD    A,(KOL_W)
        LD    L,A
        LD    H,0
        CALL  DIVIS
        LD    B,H
        LD    C,L
        CALL  11563
        JP    11747         ;печать среднего числа очков за один выстрел
MULT    .........
DIVIS   .........
SQR     .........
; Подпрограммы MISH, CIRC и другие, а также блоки TEXT и UDG,
;  на которые имеются ссылки в основной программе, описывались нами
```

```
;  в первом варианте программы МИШЕНЬ
MISH    .........
CIRC    .........
RND     .........
SND     .........
WAIT    .........
; Данные для мишени
TEXT    .........
LENTXT EQU   $-TEXT
; Данные для пулевых отверстий
UDG     .........
; Данные оценок
D_SUM   DEFB  10,10,8,8,8,6,6,6,4,4,4
TXT1    DEFB  22,21,0,16,4
        DEFM  "SCORE:"
TXT2    DEFB  22,21,12,16,1
        DEFM  "SHOTS:"
TXT3    DEFB  22,21,22,16,2
        DEFM  "MEAN:"
; Переменные для оценок
SCORE   DEFW  0
KOL_W   DEFB  0
COL     DEFB  0
ROW     DEFB  0
```

РАБОТА С КАЛЬКУЛЯТОРОМ

Как вы уже могли заметить, в игровых программах в большинстве случаев вполне можно обойтись только целыми числами. Но иногда все же приходится привлекать к расчетам и вещественные величины, особенно в блоке оценки игровой ситуации (это вы могли заметить в программе МИШЕНЬ: при расчете среднего арифметического явно требуются дробные числа). В свое время мы говорили, что для подобных вычислений можно обращаться к программе ПЗУ, выполняющей различные математические операции именно с такими числами и именуемой калькулятором. Эта программа расположена по адресу 40, что позволяет вызывать ее командой RST 40.

Работать с этой программой непросто, что объясняется, во-первых, большим количеством допустимых операций, а во-вторых, необходимостью следить за порядком обмена данными со стеком калькулятора. Поэтому мы расскажем лишь о самых необходимых в игровых программах функциях.

Необходимо знать, что параметры калькулятору передаются через его собственный стек, о котором вы уже знаете достаточно, а выполняемое действие определяется последовательностью *байтов-литералов*, записываемых непосредственно за командой RST 40. Поскольку все математические операции калькулятор выполняет на своем стеке, то прежде всего необходимо научиться записывать туда числа и затем снимать со стека результат. По крайней мере с двумя процедурами записи в стек значений из аккумулятора и пары BC мы вас уже познакомили, но существуют и другие подпрограммы, о которых также не мешает знать.

По адресу 10934 в ПЗУ имеется процедура, записывающая в стек калькулятора вещественное число в пятибайтовом представлении. Эти пять байт числа перед обращением к процедуре нужно последовательно разместить на регистрах A, E, D, C и B. Основная сложность здесь заключена в разбивке числа с плавающей запятой на 5 компонентов, так как при этом

применяются довольно хитрые расчеты. Однако если вам требуется записать заранее предопределенную константу, то можно воспользоваться очень простым способом, заставив операционную систему саму выполнить все необходимые действия. Идея сводится к тому, что при вводе строки в редакторе Бейсика все числа, прежде чем попадут в программу, переводятся интерпретатором из символьного в пятибайтовое представление. Делается это для того, чтобы во время выполнения программы уже не заниматься такими расчетами и тем самым сэкономить время. Следом за символами каждого числа записывается байт 14 и затем рассчитанные 5 байт. Например, число 12803.52 в памяти будет выглядеть таким образом:

```
 1  2  8  0  3  .  5  2  Префикс        Число
49 50 56 48 51 46 53 50    14    142  72  14  20 123
```

Код 14 и пять байт числа при выводе листинга бейсик-программы на экран пропускаются, но в памяти они всегда присутствуют. Просмотрев дамп программы, нетрудно найти нужные байты. Можно воспользоваться и небольшой программкой, которая будет печатать нужные числа на экране, так что останется только записать их, а затем использовать в своей программе на ассемблере. Вот примерный текст такой программки:

```
10 PRINT 12803.52
20 LET addr=PEEK 23635+256*PEEK 23636+5
30 LET addr=addr+1: IF PEEK (addr-1)<>14 THEN GO TO 30
40 FOR n=addr TO addr+4: PRINT PEEK n: NEXT n
```

Дадим некоторые пояснения относительно этой программки. В строке 10 после оператора `PRINT` записывается любое вещественное число, пятибайтовое представление которого вы хотите узнать. Эта строка может иметь другой номер, но обязательно должна располагаться в самом начале программы. Учтите, что перед ней не должно быть даже комментариев.

Далее, в 20-й строке вычисляется адрес начала бейсик-программы (берется из системной переменной `PROG`) и пропускается 5 байт, включающих номер, длину строки и код оператора `PRINT`.

Операторы строки 30 отыскивают байт с кодом 14, за которым в памяти располагаются нужные нам байты числа. А в следующей строке эти 5 байт последовательно считываются в цикле и выводятся на экран.

Узнав таким образом значения составляющих числа в пятибайтовом представлении, можно загрузить регистры и вызвать процедуру 10934 для записи его на вершину стека калькулятора:

```
        LD    A,142       ;размещаем 5 байт числа на регистрах A,
        LD    E,72        ; E
        LD    D,14        ; D
        LD    C,20        ; C
        LD    B,123       ; и B
        CALL  10934       ;заносим число в стек калькулятора
```

Можно предложить еще один способ укладки десятичного числа в стек калькулятора с применением процедуры 11448. Именно этой процедурой пользуется интерпретатор, работая с числовыми величинами в символьном представлении. Выполняя программу, Бейсик сохраняет адрес текущего интерпретируемого кода в системной переменной `CH_ADD` (23645/23646) и в данном случае нам достаточно записать в нее адрес символьной строки, содержащей требуемое число, чтобы заставить интерпретатор разбить его на 5 байт и уложить в стек калькулятора. Не помешает предварительно сохранить, а затем восстановить прежнее значение переменной `CH_ADD`, иначе нормальный выход в операционную систему, а тем более, продолжение выполнения бейсик-программы окажется невозможным. Не забывайте, пользуясь этим методом, в конце строки, представляющей десятичное число,

ставить код 13 (в принципе, это может быть практически любой символ, кроме цифр, точки, плюса и минуса, а также букв E и e).

```
        LD    HL,(23645)  ;запоминаем в машинном стеке
        PUSH  HL          ; значение переменной CH_ADD
        LD    HL,NUMBER   ;адрес строки с десятичным числом
        LD    (23645),HL  ; записываем в переменную CH_ADD
        LD    A,(HL)      ;берем в аккумулятор первый символ
                          ; (обязательно!)
        CALL  11448       ;помещаем число из текстовой строки
                          ; NUMBER в стек калькулятора
        POP   HL          ;восстанавливаем прежнее значение
        LD    (23645),HL  ; системной переменной CH_ADD
        .........         ;продолжаем программу
; Символьное представление десятичного числа
NUMBER DEFM  "12803.52"
        DEFB  13          ;байт-ограничитель символьной строки
```

Надо добавить, что хотя этот способ и кажется наиболее удобным, но у него есть один серьезный недостаток - работает он несравненно дольше всех предыдущих. Самое смешное, что он требует даже больше времени, чем в Бейсике, так как эта операция выполняется при вводе строки и во время исполнения программы интерпретатор уже располагает пятибайтовым представлением каждого числа.

После занесения в стек калькулятора тем или иным способом числовых значений, с ними нужно что-то сделать, для чего и предназначена команда RST 40. Как вы помните, раньше мы использовали стек калькулятора для вывода чисел на экран, а также для рисования линий и окружностей. Теперь посмотрим, как над числами в стеке производить различные математические операции.

Как мы уже сказали, для этого нужно записать специальные управляющие последовательности байтов непосредственно за командой RST 40. В табл. 9.1 перечислены наиболее употребительные команды калькулятора, выполняемые ими функции и состояние стека после выполнения операции, считая, что изначально в стеке были записаны два числа: X - на вершине (был записан последним) и Y - под ним. Например, для сложения этих двух вещественных чисел применяется литерал 15, а для деления - 5. В одной команде можно перечислить произвольное количество действий, а для завершения расчетов в конце последовательности литералов всегда обязательно указывать байт 56, который возвращает управление на следующую за ним ячейку памяти. Понятно, что последовательность литералов в программу на ассемблере может быть вставлена с помощью директивы DEFB.

Таблица 9.1. Значение некоторых кодов калькулятора

Литерал	**Операция**	**Состояние стека после операции**		
1	Замена элементов	X	Y	
3	Вычитание	Y - X		
4	Умножение	$Y \times X$		
5	Деление	Y / X		
6	Возведение в степень	Y^X		
15	Сложение	Y + X		
27	Изменение знака	Y	-X	
39	Целая часть числа	Y	INT X	

40	Квадратный корень	Y	SQR X	
41	Знак числа	Y	SGN X	
42	Абсолютная величина	Y	ABS X	
49	Копирование стека	Y	X	X
56	Конец расчетов	Y	X	
88	Округление числа	Y	INT(X+.5)	
160	Дописать 0	Y	X	0
161	Дописать 1	Y	X	1
162	Дописать 0.5	Y	X	.5
163	Дописать PI/2	Y	X	PI / 2
164	Дописать 10	Y	X	10

В качестве иллюстрации приведем программку, вычисляющую выражение 823×5503/(32-17) и выводящую результат на экран. При выполнении расчетов необходимо внимательно следить за очередностью выполнения операций, поэтому прежде нужно продумать порядок занесения чисел в стек (помните, что калькулятор имеет доступ только к величинам, находящимся на вершине стека). Поскольку в данном случае первым должно выполняться действие в скобках, а умножение и деление имеют одинаковый приоритет, то укладывать числа в стек будем в той же последовательности, в которой они встречаются в выражении, чтобы калькулятор мог выбирать их в обратном порядке:

```
ORG   60000
ENT   $
CALL  3435        ;очищаем экран
LD    A,2         ; и подготавливаем его для печати
CALL  5633
LD    BC,823      ;заносим в стек все части выражения
CALL  11563
LD    BC,5503
CALL  11563
LD    A,32
CALL  11560
LD    A,17
CALL  11560
RST   40          ;вызываем калькулятор
DEFB  3           ; X = 32 - 17
DEFB  5           ; X = 5503 / X
DEFB  4           ; X = 823 × X
DEFB  56          ; конец расчетов
CALL  11747       ;выводим результат на экран
RET
```

После запуска этой подпрограммы вы увидите на экране число 301931.27. Тот же результат получается и при выполнении оператора `PRINT 823*5503/(32-17)`.

Обязательным условием при работе с калькулятором является не только соблюдение порядка выполнения расчетов. При ошибке вы в худшем случае получите неверный результат. Гораздо важнее следить за состоянием стека калькулятора, так как если после завершения программы он окажется не в том же виде, как и в начале, последствия могут даже оказаться фатальными. Для безопасности перед выходом в Бейсик можно вызвать процедуру по адресу 5829, которая очистит стек калькулятора, хотя нужно сказать, что и это лекарство в тяжелых

случаях может не помочь. Поэтому при особо сложных вычислениях (а по началу и в самых простых случаях) желательно проследить за стеком на каждом шаге расчетов. Для приведенной выше программки можно сделать примерно такую схемку:

Последовательно заносим числа в стек:

```
823
823         5503
823         5503        32
823         5503        32        17
```

Вызываем калькулятор (RST 40):

```
823         5503        15              ; 32 - 17
823         366.867                     ; 5503 / 15
301931.27                               ; 823 × 366.867
```

Из этой схемы сразу видно, что в конце расчетов на вершине стека калькулятора осталось единственное число - результат. Перед выходом в Бейсик необходимо удалить также и его. Для этого мы вызвали процедуру 11747, которая сняла полученное значение с вершины стека и напечатала его на экране. Таким образом, состояние стека осталось тем же, что до начала работы нашей программки.

Если вы не собираетесь сразу после вычислений печатать результат на экране или использовать его для вывода графики, нужно каким-то образом снять полученное значение со стека и сохранить его для будущего применения. Для этого нужно обратиться к одной из перечисленных ниже процедур, выбрав из них наиболее подходящую для каждого конкретного случая.

Подпрограмма, расположенная по адресу 11682, снимает число с вершины стека, округляет его до ближайшего целого и помещает в регистровую пару BC. Если число было положительным или нулем, то устанавливается флаг Z, в противном случае он будет сброшен. Может оказаться, что значение в стеке по абсолютной величине превышает максимально допустимое для регистровых пар (как это произошло в предыдущем примере). В этом случае на ошибку укажет флаг CY, который будет установлен в 1. Поэтому если вы не уверены в том, что результат не превысит 65535, лучше всегда проверять условие C и при его выполнении производить в программе те или иные коррекции, либо выводить на экран соответствующее сообщение.

Процедура 8980 похожа на предыдущую, но округленное значение из стека калькулятора помещается в аккумулятор. Здесь знак числа возвращается в регистр C: 1 для положительных чисел и нуля и -1 для отрицательных. Если число в стеке превысит величину байта и выйдет из диапазона -255...+255, то будет выдано сообщение Бейсика `Integer out of range`. Естественно, что ни о каком продолжении программы в этом случае речи быть не может, поэтому не применяйте ее, если не уверены, что результат не окажется слишком велик.

Округление чисел не всегда может оказаться удовлетворительным решением. Иногда требуется сохранить число в первозданном виде и для этого можно применить вызов процедуры ПЗУ, находящейся по адресу 11249. Она выполняет действие, обратное подпрограмме 10934 и извлекает из стека калькулятора все 5 байт числа, а затем последовательно размещает их на регистрах A, E, D, C и B. Выделив в программе на ассемблере область в 5 байт с помощью директивы DEFS 5, можно сохранить там полученный результат, чтобы впоследствии вновь им воспользоваться при расчетах.

Однако приведенные процедуры мало пригодны при работе с большим количеством пятибайтовых переменных. В этом случае лучше не обращаться за помощью к ПЗУ, а написать собственные процедуры для обмена данными между переменными и стеком калькулятора.

В процедуре укладки в стек пятибайтовой переменной не повредит предварительная проверка на предмет наличия свободной памяти. Для этого вызовем подпрограмму 13225, которая проверит, можно ли разместить на стеке 5 байт, и в случае нехватки памяти выдаст сообщение об ошибке `Out of memory`. Затем перенесем 5 байт переменной на вершину стека калькулятора и увеличим системную переменную `STKEND`, выполняющую ту же роль, что и регистр SP для машинного стека. Перед обращением к процедуре в паре HL нужно указать адрес пятибайтовой переменной.

```
PUTNUM CALL  13225       ;проверка наличия свободной памяти
       LD    BC,5        ;переносим 5 байт
       LD    DE,(23653)  ;адрес вершины стека калькулятора
       LDIR              ;переносим
       LD    (23653),DE  ;новый адрес вершины стека
       RET
```

Процедура GETNUM будет выполнять противоположное действие: перемещение пяти байт числа с вершины стека калькулятора и уменьшение указателя `STKEND`. Адрес переменной также будем указывать в HL. Заодно можно выполнить проверку перебора стека, так как именно эта ошибка наиболее опасна.

```
GETNUM PUSH  HL
       LD    DE,(23653)  ;проверка достижения «дна» стека
       LD    HL,(23651)  ;системная переменная STKBOT, адресующая
                         ; основание стека калькулятора
       AND   A
       SBC   HL,DE       ;сравниваем значения STKEND и STKBOT
       JR    NC,OUTDAT   ;переход на сообщение, если стек
                         ; полностью выбран
       POP   HL
       LD    BC,5
       ADD   HL,BC       ;указываем на последний байт переменной
       DEC   HL
       DEC   DE
       EX    DE,HL
       LDDR              ;переносим 5 байт из стека в переменную
       INC   HL
       LD    (23653),HL  ;обновляем указатель на вершину
                         ; стека калькулятора
       RET
OUTDAT RST   8           ;сообщение об ошибке
       DEFB  13          ; Out of DATA
```

КОНТРОЛЬ ВРЕМЕНИ (РАБОТА С ПРЕРЫВАНИЯМИ)

Прерывания с полным правом можно отнести к наиболее мощным и интересным ресурсам компьютера. К сожалению, работа с ними из Бейсика абсолютно невозможна и именно поэтому данный вопрос для многих из вас может оказаться совершенно новым и неизведанным. К настоящему моменту вы уже, должно быть, достаточно освоились с ассемблером и, надеемся, неплохо представляете, как работает микропроцессор, что дает возможность приступить наконец к изучению этой серьезной темы.

Для начала выясним, что же собой представляют прерывания. Попробуем, не вдаваясь в конструкторские тонкости, объяснить принцип этого явления просто «на пальцах». Когда вы находитесь в редакторе Бейсика или GENS и размышляете над очередной строкой программы, компьютер не торопит вас и терпеливо ожидает нажатия той или иной клавиши. Может даже показаться, что микропроцессор в это время и вовсе не работает. Но, как вы уже знаете, это не так. Просто выполняется некоторая часть программы, аналогичная процедуре WAIT, описанной ранее: в цикле опрашивается системная переменная LAST_K и когда вы нажимаете какую-то клавишу, код ее появляется в ячейке 23560. Но, спрашивается, откуда он там берется? Программа ведь только читает ее значение, никак не модифицируя ее содержимое. А разрешается эта загадка довольно просто. Дело в том, что 50 раз в секунду микропроцессор отвлекается от основной программы и переключается на выполнение специальной процедуры обработки прерываний, расположенной по адресу 56, словно бы встретив команду RST 56 или CALL 56, только переход этот происходит не программным, а аппаратным путем. У процедуры 56 есть две основных задачи: опрос клавиатуры и изменение текущего значения таймера (системная переменная FRAMES - 23672/73/74). Результаты опроса клавиш также заносятся в область системных переменных, в частности, код нажатой клавиши помещается в LAST_K. После выхода из прерывания микропроцессор как ни в чем не бывало продолжает выполнять основную программу. В результате получается довольно интересный эффект: создается впечатление, будто бы параллельно работают два микропроцессора, каждый из которых выполняет свою независимую задачу.

Все это прекрасно, но какую пользу для себя мы можем из этого извлечь? Ведь в ПЗУ ничего не изменишь. Действительно, от прерываний программистам было бы не много проку, если бы невозможно было переопределять адрес процедуры для их обработки. Мы уже говорили о существовании регистра I, называемого регистром вектора прерываний, а сейчас расскажем, какую роль он выполняет в программах, использующих собственные прерывания.

Прежде всего вам нужно знать, что существует три различных режима прерываний. Они обозначаются цифрами от 0 до 2. Стандартный режим имеет номер 1, и о нем мы уже кое-что сказали. Нулевой режим нам не интересен, поскольку на практике он ничем не отличается от первого (именно, на практике, потому что на самом деле имеются существенные различия, но в ZX Spectrum они не реализованы). А вот о втором режиме нужно поговорить более основательно.

Сначала скажем несколько слов о том, как он работает и что при этом происходит в компьютере. С приходом сигнала прерываний микропроцессор определяет адрес указателя на процедуру обработки прерываний. Он составляется из байта, считанного с шины данных (младший), который, собственно, и называется *вектором прерывания* и содержимого регистра I (старший байт адреса). Затем на адресную шину переписывается значение полученного указателя, но предварительно прежнее состояние шины адреса заносится в стек. Таким образом, совершается действие, аналогичное выполнению команды микропроцессора CALL. Поскольку в ZX Spectrum вектор прерывания, как правило, равен 255, то на практике адрес указателя может быть определен только регистром I. Для этого его значение нужно умножить на 256 и прибавить 255.

Для установки нового обработчика прерываний нужно выполнить ряд действий. Перечислим их в том порядке, в котором они должны производиться:

1. Запретить прерывания, так как есть вероятность того, что сигнал прерываний придет во время установки, а это может привести к нежелательным последствиям. Достигается это выполнением команды микропроцессора DI.
2. Записать в память по рассчитанному заранее адресу указатель на процедуру обработки прерываний (то есть адрес этой процедуры).

3. Задать в регистре вектора прерываний I старший байт адреса указателя на обработчик.
4. Установить командой IM 2 второй режим прерываний.
5. Вновь разрешить прерывания командой EI.

Естественно, что к этому моменту сама процедура обработки прерываний должна иметься в памяти. Для возврата к стандартному режиму обработки прерываний нужно выполнить похожие действия:

1. Запретить прерывания.
2. Не помешает восстановить значение регистра I, записав в него число 63.
3. Назначить командой IM 1 первый режим прерываний.
4. Разрешить прерывания.

Несколько подробнее нужно остановиться на втором и третьем пунктах установки прерываний. Предположим, что процедура-обработчик находится по адресу 60000 (#EA60) и память, начиная с адреса 65000, никак в программе не используется. Значит указатель можно поместить именно в эту область. Для регистра I в этом случае можно выбрать одно из двух значений: 253 или 254. Тогда для размещения указателя можно использовать либо адреса 65023/65024 (253×256+255/256) либо 65279/65280 (254×256+255/256). Например, при I равном 254 запишем по адресу 65279 младший байт адреса обработчика - #60, а в 65280 поместим старший байт - #EA.

Однако нужно учитывать, что некоторые внешние устройства могут изменять значение вектора прерывания. Кроме того, если ваш Speccy сработан не слишком добросовестным производителем, то вектор прерывания иногда может скакать совершенно произвольным и непредсказуемым образом. Принимая это во внимание, даже во многих фирменных играх используется несколько иной подход. Вместо записи двух байтов по определенному адресу выстраивается целая таблица размером как минимум 257 байт с таким расчетом, чтобы при любом значении вектора прерываний считывался один и тот же адрес. Понятно, что для этого все байты таблицы должны быть одинаковыми. Это несколько осложняет установку прерывания и требует больше памяти, но зато значительно увеличивает надежность работы программы.

Наиболее удачным для такой таблицы представляется байт 255 (#FF). В этом случае обработчик прерываний должен находиться по адресу 65535 (#FFFF). На первый взгляд может показаться странным выбор такого адреса, ведь остается всего один байт! Но и этого единственного байта оказывается достаточным, если в него поместить код команды JR. Следующий байт, находящийся уже по адресу 0, укажет смещение относительного перехода. По нулевому адресу в ПЗУ записан код команды DI (#F3), поэтому полностью команда будет выглядеть как JR 65524. Далее в ячейке 65524 можно разместить уже более «длинную» команду JP address и заданный в ней адрес может быть совершенно произвольным.

Приведем пример такой подпрограммы установки прерываний:

```
IMON    LD    A,24         ;код команды JR
        LD    (65535),A
        LD    A,195        ;код команды JP
        LD    (65524),A
        LD    (65525),HL   ;в HL - адрес обработчика прерываний
        LD    HL,#FE00     ;построение таблицы для векторов прерываний
        LD    DE,#FE01
        LD    BC,256       ;размер таблицы минус 1
        LD    (HL),#FF     ;адрес перехода #FFFF (65535)
        LD    A,H          ;запоминаем старший байт адреса таблицы
```

```
        LDIR            ;заполняем таблицу
        DI              ;запрещаем прерывания на время
                        ; установки второго режима
        LD    I,A       ;задаем в регистре I старший байт адреса
                        ; таблицы для векторов прерываний
        IM    2         ;назначаем второй режим прерываний
        EI              ;разрешаем прерывания
        RET
```

Перед обращением к ней в регистровой паре HL необходимо указать адрес соответствующей процедуры обработки прерываний. Учтите, что в области памяти, начиная с адреса 65024, менять что-либо не желательно. Если все же возникнет такая необходимость, убедитесь прежде, что своими действиями вы не затроните установленные процедурой байты.

Подпрограмма восстановления первого режима выглядит заметно проще и в комментариях уже не нуждается:

```
IMOFF   DI
        LD    A,63
        LD    I,A
        IM    1
        EI
        RET
```

При составлении процедуры обработки прерываний нужно придерживаться определенных правил. Во-первых, написанная вами подпрограмма должна выполняться за достаточно короткий промежуток времени. Желательно, чтобы ее быстродействие было сопоставимо с «пульсом» прерываний, то есть чтобы ее продолжительность не превышала 1/50 секунды. Это правило не является обязательным, но в противном случае трудно будет получить эффект «параллельности» процессов. Во-вторых, и это уже совершенно необходимо, все регистры, которые могут изменить свое значение в вашей процедуре, должны быть сохранены на входе и восстановлены перед выходом. Это же относится и к любым переменным, используемым не только в прерывании, но и в основной программе. В связи с этим не рекомендуется обращаться к подпрограммам ПЗУ, по крайней мере, до тех пор, пока вы не знаете совершенно точно, какие в них используются регистры и какие системные переменные при этом могут быть изменены. Вызов подпрограмм ПЗУ не желателен еще и потому, что некоторые из них разрешают прерывания, что совершенно недопустимо во избежание рекурсии (т. е. самовызова) обработчика, который должен работать при запрещенных прерываниях. Однако использовать команду DI в самом начале процедуры не обязательно, так как это действие выполняется автоматически и вам нужно только позаботиться о разрешении прерываний перед выходом.

Если вы не хотите лишаться возможностей, предоставляемых стандартной процедурой обработки прерываний, можете завершать свою подпрограмму командой JP 56. А при использовании прерываний в бейсик-программах без этого просто не обойтись, иначе клавиатура окажется заблокирована. В общем случае обработчик прерываний может иметь такой вид:

```
INTERR  PUSH  AF
        PUSH  BC
        PUSH  DE
        PUSH  HL
        .......
        POP   HL
        POP   DE
        POP   BC
        POP   AF
        JP    56
```

В заключение этого раздела приведем процедуру, отсчитывающую секунды, остающиеся до окончания игры. Эта процедура может вызываться как из машинных кодов, так и из программы на Бейсике. В верхнем левом углу экрана постоянно будет находиться число, уменьшающееся на единицу по истечении каждой секунды. Для применения этой подпрограммы в реальной игре вам достаточно изменить адрес экранной области, куда будут выводиться числа и, возможно, начальное значение времени, отводимое на игру. Момент истечения времени определяется содержимым ячейки по смещению ORG+4. Если ее значение окажется не равным 0, значит игра закончилась.

```
        ORG   60000
        JR    INITI
        JR    IMOFF
OUTTIM  DEFB  0
INITI   LD    HL,D_TIM0
        LD    DE,D_TIME
        LD    BC,3
        LDIR
        XOR   A
        LD    (OUTTIM),A
        LD    HL,TIM0
        LD    (HL),50
        INC   HL
        LD    (HL),A
        INC   HL
        LD    (HL),A
        LD    HL,TIMER    ;установка прерывания
IMON    .........
IMOFF   .........
TIMER   PUSH  AF
        PUSH  BC
        PUSH  DE
        PUSH  HL
        CALL  CLOCK
        POP   HL
        POP   DE
        POP   BC
        POP   AF
        JP    56
TIM0    DEFB  50          ;количество прерываний в секунду
TIM1    DEFB  0           ;время «проворота» третьего символа
TIM2    DEFB  0           ;время «проворота» второго символа
TIM3    DEFB  0           ;время «проворота» первого символа
D_TIM0  DEFM  "150"       ;символы, выводимые на экран
D_TIME  DEFM  "150"       ;начальное значение времени
; Проверка необходимости изменения текущего времени
CLOCK   LD    HL,TIM0
        DEC   (HL)
        JR    NZ,CLOCK1
        LD    (HL),50
; Уменьшение секунд
        LD    A,8         ;символ «проворачивается» за 8
        LD    (TIM1),A    ; тактов прерывания
        LD    HL,D_TIME+2
        LD    A,(HL)
        DEC   (HL)
        CP    "0"
        JR    NZ,CLOCK1
        LD    (HL),"9"
; Уменьшение десятков секунд
        LD    A,8
        LD    (TIM2),A
        DEC   HL
        LD    A,(HL)
```

```
        DEC   (HL)
        CP    "0"
        JR    NZ,CLOCK1
        LD    (HL),"9"
; Уменьшение сотен секунд
        LD    A,8
        LD    (TIM3),A
        DEC   HL
        LD    A,(HL)
        DEC   (HL)
        CP    "0"
        JR    Z,ENDTIM    ;если время истекло
CLOCK1  LD    DE,#401D    ;адрес экранной области
        LD    A,(D_TIME)  ;первый символ - сотни секунд
        LD    HL,TIM3
        CALL  PRNT
        LD    A,(D_TIME+1) ;второй символ - десятки секунд
        CALL  PRNT
        LD    A,(D_TIME+2) ;третий символ - секунды
; Печать символов с учетом их «проворота»
PRNT    PUSH  HL
        LD    L,A         ;расчет адреса символа
        LD    H,0         ; в стандартном наборе
        ADD   HL,HL
        ADD   HL,HL
        ADD   HL,HL
        LD    A,60
        ADD   A,H
        LD    H,A
        EX    (SP),HL
        LD    A,(HL)
        LD    C,A
        AND   A
        JR    Z,PRNT1     ;если символ «проворачивать» не нужно
        DEC   (HL)
        EX    (SP),HL
        NEG               ;пересчет адреса символьного набора для
                          ; создания иллюзии «проворота» цифры
        LD    B,A
        LD    A,L
        SUB   B
        LD    L,A
        JR    PRNT2
PRNT1   EX    (SP),HL
PRNT2   LD    B,8
        PUSH  DE
PRNT3   LD    A,(HL)
        LD    (DE),A
        INC   HL
        INC   D
        LD    A,C
        AND   A
        JR    Z,PRNT4
; После цифры 9 при «провороте» должен появляться 0, а не двоеточие
        LD    A,L
        CP    208         ;адрес символа :
        JR    C,PRNT4
        SUB   80          ;возвращаемся к адресу символа 0
        LD    L,A
PRNT4   DJNZ  PRNT3
        POP   DE
        POP   HL
        INC   DE
        DEC   HL
        RET
```

```
; Истечение времени - выключение 2-го режима прерываний
ENDTIM POP   HL          ;восстановление значения указателя стека
                         ; после команды CALL CLOCK
       CALL  IMOFF
       LD    A,1         ;установка флага истечения времени
       LD    (OUTTIM),A
       POP   HL          ;восстановление регистров
       POP   DE
       POP   BC
       POP   AF
       RET
```

ГЛАВА ДЕСЯТАЯ,

в которой показано, как заставить компьютер звучать по вашим нотам

Как вы понимаете, никакая игра не сможет называться полноценной, если она будет протекать при гробовой тишине. Посему получение различных акустических эффектов и написание хотя бы простейшей компьютерной музыки является достаточно важным этапом создания игровой программы. Из данной главы вы узнаете, какими средствами достигаются эти цели, получите представление о способах вывода звука как в стандартный канал ZX Spectrum, так и о работе с трехканальным музыкальным сопроцессором AY-3-8912, которым снабжен компьютер Spectrum 128.

ПОЛУЧЕНИЕ ЧИСТОГО ТОНА

Наиболее простой способ получения звука определенной длительности и высоты - обратиться к подпрограмме ПЗУ, ответственной за выполнение оператора Бейсика BEEP. Мы уже упоминали о ней в шестой главе, но тем не менее напомним, что располагается она по адресу 949 и требует определения регистровых пар HL и DE. Например:

```
       LD    DE,440
       LD    HL,964
       CALL  949
       RET
```

Таким способом можно получить звук практически любой высоты и продолжительности - ограничения Бейсика здесь отсутствуют. Однако при этом отсутствуют и удобства, предоставляемые интерпретатором. Чтобы написать даже очень коротенькую музыкальную фразу, придется немало попотеть, рассчитывая значения задаваемых параметров.
А рассчитываются они так. В регистровую пару DE заносится число, определяемое как f×t, где f - частота, измеряемая в герцах, а t - время в секундах (при извлечении звука ЛЯ первой октавы, который имеет частоту 440 Гц, длительностью в 1 секунду получится 440×1=440). Пара HL на входе должна содержать число, равное 437500/f-30.125 (если выполнить указанные вычисления, то получим величину 964). Таким образом, приведенная выше программа делает то же самое, что и оператор Бейсика

```
BEEP 1,9
```

Чтобы упростить задачу, можно предложить небольшую программку на Бейсике, которая, конечно, не может претендовать на роль музыкального редактора, но, по крайней мере,

автоматизирует расчеты требуемых значений. После ее запуска на экране появится некоторое подобие меню. Нажав цифровую клавишу **1**, вы сможете прослушать свое произведение. При нажатии клавиши **2** на экран выводится два столбика чисел: значения из левого столбика предназначены для загрузки пары DE, а для HL числа берутся из правого столбика. Клавиша **0** позволяет выйти в редактор Бейсика. После нажатия клавиш **1** или **2** нужно ввести желаемый темп исполнения по метроному (количество четвертных нот, исполняемых в минуту).

Ноты записываются в операторе DATA, начиная со строки 1000, парами чисел, первое из которых определяет высоту и задается так же, как и в операторе BEEP (0 - нота ДО первой октавы), а второе представляет собой относительную длительность звуков, то есть четверти, например, записываются дробью 1/4, восьмушки - 1/8 и так далее. Для обозначения конца блока данных в самом его конце нужно написать два нуля (строка 8990).

Ниже приводится текст программы, в котором для примера уже включены несколько строк данных короткой музыкальной фразы (строки 1010...1030). Наберите и сохраните программу без этих строк, а затем, дописав их, можете проверить ее работу.

```
  10 DIM f(12): RESTORE 9000
  20 FOR n=1 TO 12: READ f(n): NEXT n
  50 CLS : PRINT "1. Listen"'"2. Code"'"0. Stop"
  60 PAUSE 0: IF INKEY$="1" THEN CLS : GO TO 100
  70 IF INKEY$="2" THEN CLS : GO TO 200
  80 IF INKEY$="0" THEN STOP
  90 GO TO 60
 100 REM Прослушивание
 110 RESTORE 1000: INPUT "TEMPO: (M.M.) = ";temp
 120 READ n,d: IF d THEN BEEP d/temp*240,n: GO TO 120
 130 GO TO 50
 200 REM Расчет значений для DE и HL.
 210 RESTORE 1000: INPUT "TEMPO: (M.M.) = ";temp
 220 READ n,d: IF NOT d THEN GO TO 300
 230 LET d=d/temp*240
 240 LET f1=INT (n/12): LET f2=n-f1*12
 250 LET f=f(f2+1)/2(4-f1)
 260 PRINT "LD DE,";INT (f*d+.5),"LD HL,";INT ((437500/f-30.125)+.5)
 270 GO TO 220
 300 PRINT #0;"Press any key": PAUSE 0: GO TO 50
1000 REM Данные мелодии.
1010 DATA 7,1/16,5,1/16,4,1/16,2,1/16
1020 DATA 4,1/8,7,1/8
1030 DATA -5,1/8,-1,1/8,0,1/4
8990 DATA 0,0
9000 REM Частота в герцах для звуков одной октавы
9010 DATA 4186.01,4434.92,4698.64,4978.03
9020 DATA 5274.04,5587.65,5919.91,6271.93
9030 DATA 6644.87,7040,7458.62,7902.13
```

Получив тем или иным способом ряд чисел для загрузки регистров, нужно каким-то образом обратить их в мелодию. Для этих целей напишем процедуру, последовательно считывающую из блока данных, адресованного парой HL, двухбайтовые значения регистровых пар и извлекающую соответствующие звуки. Подпрограмма завершит свою работу либо при достижении конца блока данных (два байта 0) либо при нажатии на любую клавишу:

```
MELODY XOR   A           ;опрос клавиатуры
       IN    A,(254)
       CPL
       AND   #1F
       JR    NZ,MELODY   ;пока все клавиши не отпущены
MELOD1 XOR   A           ;опрос клавиатуры
```

```
        IN    A,(254)
        CPL
        AND   #1F
        RET   NZ          ;выход, если нажата любая клавиша
        LD    E,(HL)      ;считывание в пару DE
        INC   HL
        LD    D,(HL)
        LD    A,D
        OR    E
        RET   Z           ;выход, если конец блока данных (DE=0)
        INC   HL
        LD    A,(HL)      ;считывание данных для пары HL
        INC   HL
        PUSH  HL
        LD    H,(HL)      ;загрузка пары HL
        LD    L,A
        CALL  949         ;вывод звука
        POP   HL
        INC   HL
        JR    MELODY      ;следующая нота
```

Прежде чем обратиться к данной процедуре выпишем числа, полученные с нашим импровизированным «редактором» при заданном темпе, равном 150, в виде блока двухбайтовых данных, который завершим числом 0 (тоже двухбайтовым):

```
DATMEL DEFW  39,1086,35,1223,33,1297
       DEFW  29,1460,66,1297,78,1086
       DEFW  39,2202,49,1742,105,1642
       DEFW  0                        ;маркер конца блока данных
```

Теперь можно вызвать процедуру MELODY, например, таким образом:

```
        ORG   60000
        ENT   $
        LD    HL,DATMEL
        CALL  MELODY
        RET
DATMEL .........
MELODY .........
```

Существует и другой способ извлечения звуков, при котором все необходимые расчеты выполняются в самой программе. Хотя он и несколько сложнее только что описанного, но может показаться вам более привлекательным. Во всяком случае он более привычен, так как исходными данными при этом являются те же числа, что и параметры оператора BEEP. Подпрограмма, расположенная по адресу 1016, сама выполняет приведенные выше вычисления и обращается к процедуре вывода звука 949, а значения длительности и высоты звука передаются ей через стек калькулятора:

```
        LD    A,1         ;загружаем в аккумулятор значение
                          ; длительности звучания (1 секунда).
        CALL  11560       ;содержимое аккумулятора заносим
                          ; в стек калькулятора.
        LD    A,12        ;нота ДО второй октавы
        CALL  11560       ;посылаем в стек калькулятора
        CALL  1016        ;вызываем процедуру извлечения звука
        RET
```

Этот метод хорош всем, за исключением одной «мелочи» - числа, записываемые в аккумулятор могут быть только целыми и не отрицательными. Чтобы исправить этот недостаток, можно условиться, что длительности будут задаваться не в секундах, а в сотых долях секунды, а значение высоты звука будем интерпретировать как число со знаком. Такой подход позволит получать продолжительность звучания нот примерно до двух с половиной секунд, что в большинстве случаев вполне достаточно; диапазон звуков останется таким же, как и в Бейсике: от -60 до +68. Параметры для этой подпрограммы будем задавать в регистрах B (длительность в сотых долях секунды) и C (высота в полутонах):

```
BEPER  PUSH  BC
       LD    A,B          ;берем в аккумулятор первый параметр
       CALL  11560        ;заносим его в стек калькулятора
       LD    A,100
       CALL  11560        ;помещаем в стек число 100
       RST   40
       DEFB  5,56         ;выполняем деление
       POP   BC
       LD    A,C          ;берем второй параметр
       AND   A
       JP    M,BEPER1     ;если отрицательный, переходим на BEPER1
       CALL  11560        ; иначе помещаем в стек без изменений
       JP    1016         ; и извлекаем звук
BEPER1 NEG                ;получаем абсолютное значение
       CALL  11560        ;отправляем в стек
       RST   40
       DEFB  27,56        ;меняем знак
       JP    1016         ; и извлекаем звук
```

Взяв за основу эту подпрограмму, можно написать процедуру, аналогичную MELODY, которая бы проигрывала произвольный музыкальный фрагмент, считывая параметры из заранее подготовленного блока данных. Приведем такой пример:

```
       ORG   60000
       ENT   $
       LD    HL,D_MEL1
       CALL  MELBEP
       RET
MELBEP LD    B,(HL)       ;в B - длительность
       INC   B            ;один из способов проверки
       DEC   B            ; содержимого регистра на 0
       RET   Z
       INC   HL
       LD    C,(HL)       ;в C - высота звука
       INC   HL
       PUSH  HL
       CALL  BEPER        ;извлечение звука
       POP   HL
       JR    MELBEP
BEPER  .........
D_MEL1 DEFB  10,-5,10,0,10,1,10,2
       DEFB  20,5,10,2,20,5,10,7
       DEFB  0
```

Во всех приведенных выше фрагментах в конечном итоге использовалась подпрограмма ПЗУ 949. В этом случае, пока звучит очередная нота, компьютер оказывается полностью выключен из работы. Поэтому иногда бывает очень важно уметь получать чистый тон на низком уровне, непосредственно программируя порт динамика. Принцип получения звука таким способом предельно прост: достаточно с определенной частотой попеременно то включать, то выключать динамик. С таким способом в некоторой степени вы также уже знакомы, поэтому отметим лишь некоторые важные моменты, без знания которых невозможно получить качественный звук. Во-первых, напомним, что динамик управляется четвертым битом 254-го порта. При установке или сбросе этого бита слышны короткие щелчки, которые при быстром чередовании сливаются в сплошной звук. Но кроме динамика этот же порт отвечает и за цвет бордюра, поэтому кроме четвертого бита нужно правильно устанавливать и три младших разряда выводимого байта. Не менее важно при извлечении звука помнить о необходимости запрета прерываний. Если этого не сделать, то невозможно будет получить чистый тон. Это связано с тем, что 50 раз в секунду микропроцессор будет отвлекаться на выполнение подпрограммы обработки прерываний, что неминуемо скажется на частоте создаваемого звука.

Теперь перейдем к делу и создадим программку, извлекающую различные звуки при нажатии цифровых клавиш. При этом звучание будет продолжаться до тех пор, пока клавиша не будет отпущена.

```
        ORG   60000
        ENT   $
        DI
        LD    A,(23624)    ;получаем в аккумуляторе цвет бордюра
        AND   #38          ;выделяем биты 3, 4 и 5
        RRA                ;сдвигаем на место битов 0, 1 и 2
        RRA
        RRA
        LD    E,A          ;запоминаем в регистре E
ORGAN1  CALL  8020         ;проверка нажатия клавиши BREAK
        JR    NC,EXIT      ;если нажата, выход из программы
        PUSH  DE
        LD    DE,#100      ;счетчики для определения
                           ; нажатых клавиш
        LD    A,#F7        ;опрос полуряда 1...5
        CALL  KEYS
        LD    A,#EF        ;опрос полуряда 6...0
        CALL  KEYS
        LD    D,0          ;выбор высоты звука из таблицы
        LD    HL,DATNOT
        ADD   HL,DE
        POP   DE
        LD    B,(HL)
        INC   B
        DEC   B
        JR    Z,ORGAN1     ;если в B ноль, клавиши не нажаты, звука нет
        LD    A,E
        OUT   (254),A      ;извлечение звука
        XOR   16
        LD    E,A
ORGAN2  LD    C,20         ;цикл задержки для получения звука
                           ; определенной высоты
ORGAN3  DEC   C
        JR    NZ,ORGAN3
        DJNZ  ORGAN2
        JR    ORGAN1
EXIT    EI                 ;выход из программы
        RET
KEYS    IN    A,(254)      ;опрос выбранного полуряда
        LD    B,5          ;5 клавиш в полуряду
KEYS1   RRCA               ;сдвигаем биты вправо
        JR    C,KEYS2      ;если младший бит установлен,
                           ; клавиша отпущена
        LD    E,D          ;иначе запоминаем номер нажатой клавиши
KEYS2   INC   D            ;увеличиваем номер определяемой клавиши
        DJNZ  KEYS1        ;переходим к следующей
        RET
; Данные для получения необходимой задержки для каждого звука
DATNOT  DEFB  0,55,49,44,41,36
        DEFB  21,24,27,29,32
```

СОЗДАНИЕ ЗВУКОВЫХ ЭФФЕКТОВ

Различные звуковые и шумовые эффекты, которыми изобилуют компьютерные игры, достигаются через изменение по тому или иному закону частоты выводимого звука. Нетрудно догадаться, что высота звука напрямую зависит от продолжительности цикла задержки между командами вывода в порт динамика: чем больше задержка, тем более низким получится звук.

Техника вывода звука вам уже известна, поэтому без лишних слов сразу перейдем к делу и продемонстрируем несколько наиболее часто употребляемых в играх эффектов. Первый из них больше всего напоминает щебет птиц, особенно если его вызывать с небольшими и неравными промежутками времени:

```
TWEET  LD    A,(23624)   ;определение цвета бордюра
       AND   #38
       RRA
       RRA
       RRA
       DI
TWEET1 XOR   16          ;переключение 4-го бита
       OUT   (254),A
       PUSH  BC
       DJNZ  $           ;цикл задержки
       POP   BC
       DJNZ  TWEET1
       EI
       RET
```

Длительность эффекта перед обращением к процедуре TWEET задается в регистре B, например:

```
       LD    B,200
       CALL  TWEET
       RET
```

Прежде чем привести следующий пример, скажем несколько слов, относящихся не только к этой подпрограмме, но и ко всем остальным. Поскольку в реальных программах цвет бордюра обычно не изменяется и определен заранее, то он, как правило, не вычисляется в программе, а задается в явном виде загрузкой в аккумулятор кода нужного цвета. Вы также можете вместо первых строк от метки TWEET до команды DI просто написать XOR A для получения черного бордюра или, например, LD A,4 - для зеленого.

Другой интересный момент касается уже не самой программы, а собственно ассемблера. Вы, наверное, обратили внимание на запись

```
       DJNZ  $
```

Как известно, символ доллара при трансляции принимает значение текущего адреса размещения машинного кода, а точнее, адрес начала строки ассемблерного текста. Поэтому такая запись полностью равноценна записи

```
LOOP   DJNZ  LOOP
```

но позволяет обойтись без дополнительных меток.

После такого небольшого лирического отступления давайте продолжим «изобретение» звуковых эффектов.

Особо часто в игровых программах можно услышать множество разновидностей вибрирующих звуков. Получить такой эффект можно, периодически увеличивая и уменьшая частоту (то есть количество циклов задержки). Вибрация характеризуется двумя

параметрами: собственной частотой и глубиной (амплитудой), поэтому для такой процедуры потребуется, кроме длительности звучания, задавать и некоторые другие входные данные. Сначала приведем текст подпрограммы для получения вибрирующего звука, а затем объясним, какие значения в каких регистрах следует разместить перед обращением к ней.

```
VIBR   LD    A,(23624)
       AND   #38
       RRA
       RRA
       RRA
       LD    C,A
       DI
VIBR1  LD    D,E          ;продолжительность цикла спада (подъема)
VIBR2  LD    A,C
       XOR   16
       LD    C,A
       OUT   (254),A
       LD    A,H          ;изменение частоты звука
       ADD   A,L
       LD    H,A
VIBR3  DEC   A            ;цикл задержки
       JR    NZ,VIBR3
       DEC   D
       JR    NZ,VIBR2
       LD    A,L          ;смена направления изменения частоты
       NEG
       LD    L,A
       DJNZ  VIBR1
       EI
       RET
```

В регистр H нужно занести начальную частоту звука (имеется в виду, конечно, частота не в герцах, а в относительных единицах). Содержимое регистра E влияет на частоту вибрации: чем меньше его значение, тем быстрее спад будет сменяться подъемом и наоборот.
В регистре B задается количество циклов вибрации, то есть в конечном счете - длительность звука, а в L заносится величина, определяющая глубину вибрации, или иначе, скорость изменения высоты звука. Мы предлагаем такие значения регистров:

```
       LD    H,100
       LD    E,120
       LD    B,4
       LD    L,1
       CALL  VIBR
       RET
```

однако это только один из многих возможных вариантов. Попробуйте поэкспериментировать и подобрать наиболее интересные варианты звучания, которые впоследствии сможете использовать в своих собственных разработках.

Приведем еще одну подпрограмму, создающую другой тип вибрации, при котором частота звука, достигнув наивысшей (или же низшей) точки, возвращается к начальной своей величине. Тем самым частотная характеристика имеет внешний вид, схожий с зубьями пилы. Подпрограмма, создающая похожий звук, имеется в известном пакете Suprcode и значится там под именем «Laser». Вот как примерно она может выглядеть:

```
LASER  LD    A,(23624)
       AND   #38
       RRA
       RRA
       RRA
```

```
        DI
LASER1  PUSH  BC
        LD    L,H
LASER2  XOR   16
        OUT   (254),A
        LD    B,H
        DJNZ  $
        INC   H           ;другой вариант - DEC H
        DEC   C
        JR    NZ,LASER2
        LD    H,L
        POP   BC
        DJNZ  LASER1
        EI
        RET
```

Прежде чем обратиться к данной процедуре, необходимо в регистр B загрузить количество «зубчиков пилы», в C - продолжительность каждого «зубца», а в регистре H задать исходную высоту звука. Например:

```
        LD    B,5
        LD    C,200
        LD    H,50
        CALL  LASER
        RET
```

Если вы работали с музыкальным редактором Wham, то, вероятно, задавались вопросом, как в одном звуковом канале удается получить сразу два тона различной высоты. Во многих играх, особенно последних лет, музыкальное сопровождение выполнено в аранжировке той или иной степени сложности. Иногда можно слышать не два, а три и более голосов (например, в DEFLEKTOR или MIG-29). Конечно, написание музыки на два голоса - дело вовсе непростое, но принцип получения подобных звуков знать все же стоит. Тем более, что таким способом можно создать ряд весьма недурных эффектов. Двуголосие достигается наложением двух различных частот, поэтому программа, генерирующая одновременно два тона, может выглядеть примерно так:

```
TWOTON  LD    A,(23624)
        AND   #38
        RRA
        RRA
        RRA
        LD    H,D
        LD    L,E
        DI
TWOTN1  DEC   H           ;задержка для получения первого тона
        JR    NZ,TWOTN2
        XOR   16
        OUT   (254),A     ;извлечение первого звука
        LD    H,D         ;восстановление значения задержки
                          ; для первого тона
TWOTN2  DEC   L           ;задержка для получения второго тона
        JR    NZ,TWOTN1
        XOR   16
        OUT   (254),A     ;извлечение второго звука
        LD    L,E         ;восстановление значения задержки
                          ; для второго голоса
        PUSH  AF
        LD    A,B         ;проверка окончания звучания
        OR    C
        JR    Z,TWOTN3
        POP   AF
        DEC   BC
        JR    TWOTN1
TWOTN3  POP   AF
```

```
        EI
        RET
```

Перед обращением к процедуре в регистровой паре BC нужно указать длительность звучания, а в регистрах D и E - высоту звука соответственно в первом и втором голосах. Если в регистрах D и E задать близкие значения, то вместо двух различных тонов получится звук приятного тембра, слегка вибрирующий и как бы объемный. Послушайте, например, такое звучание:

```
        LD    BC,500      ;длительность звучания
        LD    D,251       ;высота первого тона
        LD    E,250       ;высота второго тона
        CALL  TWOTON
        RET
```

Не меньшим спросом в игровых программах пользуются и различные шумовые эффекты, имитирующие выстрелы, разрывы снарядов, стук копыт и т. п. Такие звуки характеризуются отсутствием какой-то определенной частоты - в них присутствуют частоты всего спектра. Это так называемый «белый» шум. Но обычно в шуме все же преобладают тона определенной высоты, что позволяет отличить шипение змеи от грохота обвала. И это необходимо учитывать при создании нужного эффекта.

Первый звук этого типа, который мы хотим предложить, при подборе соответствующей длительности позволяет имитировать звуки от хлопков в ладоши до шипения паровоза, выпускающего пар. Продолжительность звучания задается в регистровой паре DE, однако ее значение не должно превышать 16384, так как в качестве генератора «случайных» чисел, определяющих частоту тона используются коды ПЗУ:

```
HISS    LD    A,(23624)
        AND   #38
        RRA
        RRA
        RRA
        LD    B,A
        LD    HL,0        ;начальный адрес ПЗУ
        DI
HISS1   LD    A,(HL)      ;берем байт в аккумулятор
        AND   16          ;выделяем 4-й бит
        OR    B           ;объединяем с цветом бордюра
        OUT   (254),A     ;получаем звук
        INC   HL          ;переходим к следующему байту
        DEC   DE          ;уменьшаем значение длительности
        LD    A,D
        OR    E
        JR    NZ,HISS1    ;переходим на начало,
                          ; если звук не закончился
        EI
        RET
```

Следующий пример немного похож на предыдущий, но позволяет выделять более низкие частоты. При наличии воображения его вполне можно сравнить с отдаленными раскатами грома. Продолжительность звучания здесь определяется в регистре B:

```
CRASH   LD    A,(23624)
        AND   #38
        RRA
        RRA
        RRA
        DI
        LD    HL,100      ;начальный адрес в ПЗУ
CRASH1  XOR   16
        OUT   (254),A     ;извлекаем звук
```

```
        LD    C,A          ;сохраняем значение аккумулятора
        LD    E,(HL)       ;получаем в паре DE
        INC   HL           ; продолжительность цикла задержки
        LD    A,(HL)
        AND   3            ;ограничиваем величину старшего байта
        LD    D,A
CRASH2  LD    A,D          ;цикл задержки
        OR    E
        JR    Z,CRASH3
        DEC   DE
        JR    CRASH2
CRASH3  LD    A,C          ;восстанавливаем значение аккумулятора
        DJNZ  CRASH1
        EI
        RET
```

Неплохие результаты можно получить, если изменять во времени среднюю частоту выводимого шума. Ниже приведен текст подпрограммы, построенной именно по такому принципу:

```
EXPLOS  LD    A,(23624)
        AND   #38
        RRA
        RRA
        RRA
        LD    L,A
        DI
EXPL1   PUSH  BC
        PUSH  DE
EXPL2   PUSH  DE
EXPL3   LD    B,E
        DJNZ  $            ;задержка
        LD    A,(BC)       ;в паре BC один из первых 256 адресов ПЗУ
        AND   16
        OR    L
        OUT   (254),A
        INC   C
        DEC   D
        JR    NZ,EXPL3
        POP   DE
; Изменение высоты шума (понижение среднего тона;
;  если заменить на DEC E, тон будет наоборот повышаться)
        INC   E
        DEC   D
        JR    NZ,EXPL2
        POP   DE
        POP   BC
        DJNZ  EXPL1        ;повторение всего эффекта
        EI
        RET
```

Перед обращением к ней в регистр B заносится количество повторений эффекта (что позволяет получить звук, напоминающий описанный выше эффект «Laser»), в D задается длительность звучания и в E - величина, определяющая начальную среднюю частоту. Например, для создания звука, напоминающего взрыв бомбы, можно предложить такие значения регистров:

```
        LD    B,1
        LD    D,100
        LD    E,-1
        CALL  EXPLOS
        RET
```

а пулеметную очередь можно получить с другими исходными данными:

```
        LD    B,5
```

```
        LD    D,35
        LD    E,0
        CALL  EXPLOS
        RET
```

У всех перечисленных процедур есть один общий недостаток: на время звучания выполнение основной программы затормаживается. Существует несколько способов обхода этой неприятности. Наиболее удачным из них представляется использование прерываний. Конечно, в этом случае получить чистый тон окажется совершенно невозможно, но для создания звуковых имитаций это и не особенно важно.

Поясним суть идеи. Пятьдесят раз в секунду с приходом очередного сигнала прерываний программа приостанавливается и выполняется процедура, извлекающая серию коротких звуков - по звуку на прерывание. Понятно, что звуки должны быть действительно очень короткими, иначе толку от прерываний будет чуть.

Основная сложность в написании такой процедуры состоит, пожалуй, только в способе передачи параметров подпрограмме, генерирующей тон. Поскольку прерывания выполняются автономно и не зависят от работы основной программы, входные значения не могут передаваться через регистры, а только через память. То есть необходимо составить блок данных, описывающих высоту и длительность каждого отдельного звука - по два байта на каждый. Адреса начальной и текущей пары значений в блоке данных также должны передаваться через переменные, чтобы каждое очередное прерывание «знало», какие параметры требуется считывать. Кроме этого нужна еще одна переменная, которая будет выполнять роль флага разрешения извлечения звука. Эта же переменная может служить счетчиком циклов, если вы хотите иметь возможность многократно исполнять запрограммированный в блоке данных фрагмент.

Учитывая все сказанное, можно написать такую процедуру обработки прерываний:

```
INTERR PUSH  AF          ;сохраняем используемые
       PUSH  BC          ; в прерывании регистры
       PUSH  HL
INTER1 LD    A,(REPEAT)
       AND   A
; Если эффект прозвучал нужное количество раз,
;  завершаем обработку прерывания
       JR    Z,EXITI
       LD    HL,(CURADR) ;определяем текущий адрес
                         ; в блоке данных
       LD    B,(HL)      ;высота звука
       INC   B
       DEC   B
; Если встретился маркер конца блока данных,
;  переходим к следующему повторению
       JR    Z,EXITI0
       INC   HL
       LD    C,(HL)      ;длительность звука
       INC   HL
       LD    (CURADR),HL ;запоминаем текущий адрес
       CALL  BEEP        ;извлекаем звук
EXITI  POP   HL          ;восстанавливаем регистры
       POP   BC
       POP   AF
       JP    56          ;переходим к стандартному
                         ; обработчику прерываний
; Переход к началу эффекта - повторение
EXITI0 LD    HL,(ADREFF) ;восстанавливаем начальный
       LD    (CURADR),HL ; адрес блока данных
```

```
       LD    HL,REPEAT
       DEC   (HL)         ;уменьшаем счетчик повторений
       JR    INTER1
REPEAT DEFB  0            ;количество повторений эффекта
ADREFF DEFW  0            ;начальный адрес блока данных эффекта
CURADR DEFW  0            ;текущий адрес в блоке данных
; Извлечение звука
BEEP   XOR   A
BEEP1  XOR   16
       OUT   (254),A
       PUSH  BC
       DJNZ  $
       POP   BC
       DEC   C
       JR    NZ,BEEP1
       RET
```

Составив процедуру обработки прерываний, нужно теперь позаботиться об управлении ею. В первую очередь необходимо установить второй режим прерываний, как это было показано в предыдущей главе:

```
IMON   XOR   A            ;в начале на всякий случай
       LD    (REPEAT),A   ; запрещаем вывод звука
       LD    A,24         ;код команды JR
       LD    (65535),A
       LD    A,195        ;код команды JP
       LD    (65524),A
       LD    HL,INTERR    ;переход на процедуру
       LD    (65525),HL   ; обработки прерываний
       LD    HL,#FE00     ;формируем таблицу векторов прерываний
       LD    DE,#FE01
       LD    BC,256
       LD    (HL),#FF     ;на адрес 65535 (#FFFF)
       LD    A,H          ;запоминаем старший байт адреса таблицы
       LDIR
       DI
       LD    I,A          ;загружаем регистр вектора прерываний
       IM    2            ;включаем 2-й режим
       EI
       RET
```

Сразу же напишем и процедуру восстановления первого режима прерываний, которая будет вызываться при окончании работы программы. Она вам уже известна, но тем не менее повторим:

```
IMOFF  DI
       LD    A,63
       LD    I,A
       IM    1
       EI
       RET
```

Теперь можно написать блоки данных, характеризующие различные эффекты. При их составлении нужно учитывать две вещи: во-первых, как мы уже говорили, каждый звук должен быть достаточно коротким, чтобы он не задерживал выполнение основной программы (не более нескольких сотых долей секунды), а во-вторых, при увеличении первого параметра (высота тона) второй (длительность звучания) нужно уменьшать, иначе более низкие звуки окажутся и более продолжительными. Завершаться каждый блок данных обязан нулевым байтом, обозначающим конец звучания эффекта и переход на его начало. Приведем два приблизительных варианта:

```
EFF1   DEFB  200,5,220,4,200,5
       DEFB  100,8,80,9,50,20
       DEFB  0
EFF2   DEFB  50,20,100,6,200,3,100,6
```

```
       DEFB  0
```

Наконец, напишем управляющую часть, которая позволит легко обратиться к любой подпрограмме: включения или выключения второго режима прерываний, а также активизации того или иного эффекта. Напомним, что для «запуска» любого из заданных в блоках данных эффектов необходимо сначала занести в переменные ADREFF и CURADR начальный адрес соответствующего блока и в переменной REPEAT указать количество повторений звука.

Чтобы любую процедуру было удобно вызывать даже из Бейсика, не имеющего ни малейшего представления о метках ассемблерного текста, применим распространенный прием, часто используемый в таких случаях и которым мы однажды уже воспользовались (см. программу ГЕНЕРАТОР СПРАЙТОВ) - вставим в самом начале программы в машинных кодах ряд инструкций «длинного» перехода (JP) с указанием адресов каждой из «внешних», то есть вызываемых из другой программы, процедур. Тогда адреса обращения к любой из них будут увеличиваться с шагов в 3 байта (размер команды JP). Таким образом, наш пакет процедур будет выглядеть так:

```
       ORG   60000
; 60000 - включение второго режима прерываний
       JP    IMON
; 60003 - переход к подпрограмме выключения 2-го режима прерываний
       JP    IMOFF
; 60006 - включение эффекта1
       JP    ONEFF1
; 60009 - включение эффекта2
ONEFF2 LD    HL,EFF2
       LD    A,5
       JR    ONEFF
ONEFF1 LD    HL,EFF1
       LD    A,3
ONEFF  LD    (ADREFF),HL
       LD    (CURADR),HL
       LD    (REPEAT),A
       RET
; Блоки данных эффектов
EFF1   .........
EFF2   .........
; Инициализация второго режима прерываний
IMON   .........
; Выключение второго режима прерываний
IMOFF  .........
; Процедура обработки прерываний
INTERR .........
```

А вот фрагмент программы на Бейсике, демонстрирующий использование приведенных звуковых эффектов:

```
 10 INK 5: PAPER 0: BORDER 0: CLEAR 59999
 20 RANDOMIZE : LET x=INT (RND*30)+1: LET y=INT (RND*20)+1
 30 LET dx=1: IF RND>=.5 THEN LET dx=-1
 40 LET dy=1: IF RND>=.5 THEN LET dy=-1
 50 INK 2: PLOT 0,0: DRAW 255,0: DRAW 0,175:
    DRAW -255,0: DRAW 0,-175: INK 5
 60 RANDOMIZE USR 60000
100 PRINT AT y,x; INK 8; OVER 1;"O": LET x1=x: LET y1=y: PAUSE 3
110 IF x=0 OR x=31 THEN RANDOMIZE USR 60006: LET dx=-dx
120 IF y=0 OR y=21 THEN RANDOMIZE USR 60009: LET dy=-dy
130 LET x=x+dx: LET y=y+dy
140 IF INKEY$<>"" THEN GO TO 200
150 PRINT AT y1,x1; INK 8; OVER 1;"O": GO TO 100
200 RANDOMIZE USR 60003
```

После запуска этой программки экран окрасится в черный цвет, по краю его будет нарисована рамка и в случайном месте возникнет шарик, который начнет метаться из стороны в сторону, отскакивая от «стенок». При соприкосновении с преградами будет раздаваться протяжный вибрирующий звук, причем разный в зависимости от того, в горизонтальную или в вертикальную «стенку» ударился мячик. Для остановки программы достаточно нажать любую клавишу.

МУЗЫКАЛЬНЫЙ СОПРОЦЕССОР

Несоизмеримо большими возможностями для создания звукового оформления игровых программ обладают компьютеры ZX Spectrum 128 и Scorpion ZS 256, благодаря встроенному в них трехканальному музыкальному сопроцессору.

Мы уже показали, как можно в одном стандартном звуковом канале совместить два голоса, однако из-за наложения частот звук при этом получается «грязным», чересчур насыщенным гармониками, напоминая по тембру широкораспространенный в свое время гитарный эффект FUZZ. Музыкальный же сопроцессор позволяет получить одновременно до трех чистых тонов. Но его достоинства не ограничиваются только этим.

Как известно, музыкальный звук характеризуется тремя основными параметрами: частотой, которая определяет высоту тона, тембром (окраской), зависящим от количества и состава гармоник и амплитудой, определяющей громкость звучания. Используя только стандартный звуковой канал, легко можно управлять высотой звука, несколько сложнее получить различную окраску тона и практически невозможно влиять на громкость, а вот музыкальный сопроцессор позволяет манипулировать и этим третьим параметром, варьируя громкость звука от Forte до полного его исчезновения. Правда, что касается тембра, то и здесь, к сожалению, нет простых решений и приходится прибегать примерно к тем же методам, что и при управлении стандартным выводом. Зато вы можете без труда получить эффект «белого» шума, причем его среднюю высоту также можно регулировать.

Нельзя не упомянуть и еще об одной особенности музыкального сопроцессора. Он работает совершенно независимо, без опеки центрального процессора, поэтому последний может быть занят каким-нибудь полезным делом (например, опрашивать клавиатуру либо выводить на экран текст или графику), в то время как музыкальный сопроцессор самостоятельно извлекает звук. CPU лишь изредка нужно отвлекаться от своей работы, чтобы дать своему «коллеге» указание перейти к следующей ноте, а затем он вновь может вернуться к решению более насущных проблем.

Вы знаете, что для управления работой музыкального сопроцессора из Бейсика-128 имеется дополнительный (по отношению к стандартному Spectrum-Бейсику) оператор `PLAY`, но он на самом деле не реализует и десятой доли всех немыслимых возможностей, которые могут быть осуществлены только из ассемблера. Об этом можно судить хотя бы по тем играм, которые написаны специально для Spectrum 128.

Звук извлекается программированием собственных регистров сопроцессора, которые так же, как и регистры CPU имеют по 8 разрядов. Всего их насчитывается 16 (обозначаются от R0 до R15), но нас будут интересовать только 14 из них, так как остальные два служат для несколько иных целей, о чем сказано, например, в [2]. Сначала мы рассмотрим функции этих регистров, а затем расскажем о том, как с ними обращаться.

Первые шесть регистров (R0...R5) образуют три пары и задают высоту звука для каждого из трех каналов в отдельности (сами каналы обозначаются буквами **A**, **B** и **C**). То есть регистровая пара R0/R1 определяет частоту тона в канале A, пара R2/R3 делает то же самое для канала B и R4/R5 - для C. Хотя каждая пара состоит из 16 бит, используются только 12 младших разрядов: все 8 бит младшего регистра (R0, R2 и R4) и 4 младших бита старшего регистра (R1, R3 и R5). Таким образом, числа, определяющие высоту звука, находятся в пределах от 0 до 4095 включительно. В табл. 10.1 приводится соответствие звуков из диапазона неполных девяти октав и чисел, определяющих эти ноты.

Таблица 10.1. Значения для регистров R0...R5

	СК	**К**	**Б**	**М**	**1**	**2**	**3**	**4**	**5**
До		3389	1695	847	424	212	106	53	26
До диез		3199	1600	800	400	200	100	50	25
Ре		3020	1510	755	377	189	94	47	24
Ре диез		2850	1425	712	356	178	89	45	22
Ми		2690	1345	673	336	168	84	42	21
Фа		2539	1270	635	317	159	79	40	20
Фа диез		2397	1198	599	300	150	75	37	19
Соль		2262	1131	566	283	141	71	35	18
Соль диез		2135	1068	534	267	133	67	33	17
Ля	4031	2015	1008	504	252	126	63	31	16
Си бемоль	3804	1902	951	476	238	119	59	30	15
Си	3591	1795	898	449	224	112	56	28	14

Следующий регистр - R6 - определяет среднюю частоту выводимого шума. Поскольку получение шумовых эффектов одновременно в разных голосах не имеет практического применения (их все равно невозможно будет различить на слух), то этот регистр является общим для всех трех каналов. Для него можно задавать значения от 0 до 31, то есть значащими являются только пять младших битов.

Регистр R7 служит для управления звуковыми каналами. Он подобен флаговому регистру центрального процессора и значение имеет каждый отдельный бит. Младшие три бита используются для управления выводом чистого тона в каждый из трех каналов. Если бит установлен, вывод запрещен, а при сброшенном бите вывод звука разрешается. Бит 0 связан с каналом A, 1-й бит относится к каналу B и 2-й - к C. Биты 3, 4 и 5 заведуют выводом в каналы A, B и C соответственно частоты «белого» шума. При установке бита вывод также запрещается, а при сбросе его в 0 - разрешается. 6-й и 7-й биты для извлечения звука значения не имеют.

Регистры R8, R9 и R10 определяют громкость звука, выводимого соответственно в каналы A, B и C. С их помощью можно получить 16 уровней громкости, посылая в них значения от 0 до 15. То есть значение для получаемой амплитуды в этих регистрах имеют 4 младших бита. Однако следует знать еще об одной интересной особенности этих трех регистров. Если в каком-нибудь из них установить 4-й бит (например, послав в него число 16), то получится звук не с постоянной, а с изменяющейся во времени громкостью. В этом случае необходимо указать дополнительную информацию в регистрах R11, R12 и R13.

Спаренные регистры R11 и R12 задают скорость изменения громкости звука: чем больше число, тем более плавной будет огибающая. В них можно записывать значения от 0 до 65535. Надо сказать, что изменение младшего регистра почти не ощущается, поэтому чаще достаточно определять лишь регистр R12.

Регистр R13 формирует огибающую выходного сигнала. Установкой одного или нескольких битов из младшей четверки можно получить несколько разнообразных эффектов. При установке нулевого бита звук получается затухающим, если установить 2-й бит, громкость будет наоборот увеличиваться, а установив одновременно 1-й и 3-й биты, вы получите звук, постоянно изменяющийся по громкости.

Последними тремя регистрами действительно иногда бывает удобно пользоваться, хотя гораздо чаще огибающая формируется программным путем, что позволяет получить значительно большее разнообразие оттенков звучания.

Перейдем теперь к вопросу, как программируются регистры музыкального сопроцессора. Связь с ними осуществляется через порты с адресами 49149 (#BFFD) и 65533 (#FFFD). Чтобы записать какое-либо значение в любой из регистров, его необходимо прежде всего выбрать (или назначить), выполнив команду OUT в порт 65533. Например, регистр R8 выбирается следующими командами:

```
       LD    BC,65533    ;в паре BC - адрес порта
                         ; для выбора регистра
       LD    A,8         ;в аккумуляторе - номер регистра
       OUT   (C),A       ;выбор
```

После этого в выбранный регистр можно записывать данные либо читать его содержимое. Для записи используется порт 49149, а для чтения - опять же 65533. Приведем фрагмент программы, в котором читается значение установленного ранее регистра и если оно не равно 0, содержимое регистра уменьшается на единицу:

```
       LD    BC,65533    ;адрес порта для чтения
       IN    A,(C)       ;читаем значение текущего регистра
       JR    Z,ZERO      ;если 0, обходим
       DEC   A           ;уменьшаем на 1
; Выбираем порт 49149 для записи
;  (значение регистра C остается прежним - #FD)
       LD    B,#BF
       OUT   (C),A       ;записываем значение в выбранный регистр
ZERO   .........         ;продолжение программы
```

Добавим к сказанному, что выбор регистров музыкального сопроцессора и все манипуляции с ними лучше производить при запрещенных прерываниях, хотя в приведенных примерах это и не отражено.

Можно написать универсальную подпрограмму, которая считывает из блока данных значения всех регистров и тем самым задает параметры звуков одновременно для всех трех каналов:

```
OUTREG DI                ;запрещаем прерывания
; Данные будем считывать из массива DATREG
;  в обратном порядке, начиная с последнего элемента
       LD    HL,DATREG+13
       LD    D,13        ;начальный номер загружаемого регистра
       LD    C,#FD       ;младший байт адреса порта сопроцессора
OUTR1  LD    B,#FF       ;адрес для выбора регистра
       OUT   (C),D       ;выбираем регистр
       LD    B,#BF       ;адрес для записи в регистр
; Записываем в порт байт из ячейки, адресуемой парой HL,
```

```
;  и уменьшаем HL на 1
        OUTD
        DEC   D           ;переходим к следующему регистру
                          ; (с меньшим номером)
        JP    P,OUTR1     ;повторяем, если записаны еще не все
                          ; регистры (D >= 0)
        EI                ;разрешаем прерывания
        RET
DATREG DEFS  14           ;массив данных для регистров сопроцессора
```

Эта процедура вполне может быть использована как для получения отдельных звуковых эффектов, так и для создания музыкальных произведений. Основная сложность заключается в написании программы, которая бы изменяла соответствующим образом элементы массива DATREG и тем самым управляла работой сопроцессора. Чтобы получить действительно первоклассное звучание, потребуется достаточно серьезная программа, которую объем книги, к сожалению, не позволяет здесь привести (надеемся, ее все же удастся включить в одну из последующих книг серии «Как написать игру»). Поэтому мы предлагаем более простую процедуру, извлекающую отдельные звуки, характер которых, тем не менее, вы сможете изменять практически в неограниченных пределах. (Упомянутая музыкальная программа строится, в общем, по такому же принципу, что и приводимая здесь процедура. Поэтому после ее досконального изучения вы можете попытаться самостоятельно написать программу, пригодную для исполнения музыкальных произведений.)

Эту программу также лучше составить в виде прерывания, чтобы можно было получать звуки любой продолжительности и не отвлекаться на их формирование в основной программе. Начнем с выяснения, какие переменные нам здесь потребуются. Помимо уже известных по предыдущему примеру значений базового и текущего адресов в блоке данных, описывающем характер звучания и флага разрешения вывода звука понадобятся переменные, задающие частоту тона и шума, длительность звучания, количество повторений эффекта, флаг разрешения вывода в канал тона или шума. Кроме этого, необходимо каким-то образом задавать огибающую звука и характер изменения высоты тона или шума. Эти две последние характеристики должны изменяться независимо друг от друга, поэтому порядок их изменения лучше всего определять в двух дополнительных блоках данных. То есть к перечисленным переменным добавятся еще четыре: базовые и текущие адреса для каждого из этих двух блоков.

В связи с обилием переменных, а также и с тем, что все они должны иметься в трех экземплярах - для каждого канала по полному набору - лучше всего свести их в таблицу, начало которой передавать в регистре IX. Построим такую таблицу с указанием смещений от начала и значений каждого смещения:

```
0/1   - базовый адрес блока данных эффекта
2/3   - текущий адрес в блоке данных
4/5   - частота тона
6     - частота шума
7     - флаг разрешения вывода в канал
8     - длительность звучания эффекта
9     - количество повторений эффекта
10    - вывод тона (1), шума (8) или их комбинации (9)
11/12 - базовый адрес данных для формирования частоты
13/14 - текущий адрес данных для формирования частоты
15/16 - базовый адрес данных для формирования огибающей
17/18 - текущий адрес данных для формирования огибающей
```

Поскольку задавать все значения переменных в основной программе немыслимо, они должны быть определены в блоках данных, а чтобы иметь возможность устанавливать их выборочно (ведь, например, при выводе чистого тона частота шума не имеет значения), используем принцип управляющих кодов. В основном блоке данных будем определять адреса двух

дополнительных блоков и заказывать выводимый звук - чистый тон, шум или их комбинацию, а также длительность звучания эффекта:

```
0 - конец данных (возврат на повторение эффекта)
1 - адрес данных для изменения тона (+2 байта адреса)
2 - адрес данных для формирования огибающей (+2 байта адреса)
3 - управление выводом тона/шума (+2 байта: 1-й определяет вывод тона,
    шума или комбинированный вывод, 2-й - длительность звучания)
```

Когда программа встретит код 0, вывод звука либо прекратится, либо весь эффект повторится еще раз - в зависимости от заданного изначально количества повторений. Следующие два байта после кодов 1 или 2 интерпретируются как адреса дополнительных блоков данных, определяющих характер изменения частоты звука и огибающей соответственно. С этих двух кодов должен начинаться любой блок данных, иначе программа не будет «знать», каким образом изменять звук. Последний код (3) служит собственно для извлечения звука. После него необходимо указать еще два однобайтовых параметра: первый задает вывод тона (1), шума (8) либо одновременно и тона и шума (9), второй определяет продолжительность заданного звука в 50-х долях секунды.

В дополнительных блоках данных также используем некоторые управляющие коды. После байта со значением 128 будет задаваться двухбайтовая величина частоты тона (см. табл. 10.1), а за кодом 129 последует байт средней частоты шума, который может иметь значения от 0 (самый высокий звук) до 31 (самый низкий). Определив частоты, можно поставить метку начала их изменения, поставив код 130. Далее должны следовать значения приращения частоты, которые лежат в диапазоне от -124 до +127. За один такт прерываний (1/50 секунды) будет выполнен один из этих кодов. Завершаться этот блок данных должен кодом 131, после которого все составляющие его числа будут проинтерпретированы с начала или от кода 130, если таковой был использован. Соберем все коды данных для изменения частоты воедино:

```
128          - задание частоты тона (+2 байта частоты тона)
129          - задание частоты шума (+1 байт частоты шума)
130          - метка повторения эффекта
131          - возврат на начало или метку
-124...+127 - приращение частоты
```

В блоке данных для формирования огибающей будет использован только один управляющий код 128, отмечающий конец интерпретации записанных значений и переход на повторение эффекта. Остальные коды, задающие громкость звука, могут передаваться числами от 0 до 15. Каждое из этих чисел также обрабатывается за одно прерывание.

```
128    - возврат на начало данных огибающей
0...15 - значение громкости
```

Прежде чем продолжить, хочется обратить ваше внимание на то, что при построении собственных блоков данных необходимо следить за порядком следования управляющих кодов, иначе результат будет очень далек от ожидаемого. Кроме того, ни один из блоков не может состоять только из управляющих кодов, поскольку в этом случае компьютер просто «зависнет».

Выяснив, что мы хотим в итоге получить, напишем процедуру для интерпретации описанных блоков данных. Она имеет довольно внушительные размеры, но все же попытайтесь с ней разобраться. На входе перед обращением к ней в аккумуляторе задается номер канала (0 для A, 1 для B и 2 для C), а индексный регистр IX, как мы уже говорили, адресует таблицу переменных соответствующего канала:

```
GETSND LD    (N_CHAN),A  ;запоминаем номер канала
```

```
        LD    A,(IX+7)
        AND   A
        RET   Z           ;выход, если вывод в канал не разрешен
        DEC   (IX+8)      ;уменьшаем счетчик длительности звука
        JR    NZ,GETS5
        LD    L,(IX+2)    ;текущий адрес основного блока данных
        LD    H,(IX+3)
GETS1   LD    A,(HL)
        INC   HL
        AND   3
        JR    NZ,GETS2
; Код 0 - возврат на повторение эффекта
        LD    A,(IX+11)   ;установка текущего адреса
        LD    (IX+13),A   ; данных изменения частоты
        LD    A,(IX+12)   ; на начало
        LD    (IX+14),A   ; блока
        LD    L,(IX)      ;начальный адрес основного
        LD    H,(IX+1)    ; блока данных
        DEC   (IX+9)      ;уменьшение количества повторений
        JR    NZ,GETS1
        XOR   A           ;завершение звучания в канале
        LD    (IX+7),A    ;запрет вывода звука в канал
        LD    (IX+10),A
        LD    A,(N_CHAN)
        ADD   A,8
        LD    E,A
        XOR   A
        JP    SETREG      ;выключение громкости
GETS2   DEC   A
        JR    NZ,GETS3
; Код 1 - адрес данных для изменения тона
        LD    A,(HL)      ;младший байт адреса
        LD    (IX+11),A
        INC   HL
        LD    A,(HL)      ;старший байт адреса
        LD    (IX+12),A
        INC   HL
        JR    GETS1
GETS3   DEC   A
        JR    NZ,GETS4
; Код 2 - адрес данных для формирования огибающей
        LD    A,(HL)      ;младший байт адреса
        LD    (IX+15),A
        LD    (IX+17),A
        INC   HL
        LD    A,(HL)      ;старший байт адреса
        LD    (IX+16),A
        LD    (IX+18),A
        INC   HL
        JR    GETS1
; Код 3 - управление выводом тона/шума
GETS4   LD    A,(HL)      ;1 - тон, 8 - шум, 0 - пауза,
                          ; 9 - тон и шум одновременно
        INC   HL
        AND   9
        LD    (IX+10),A
        LD    A,(HL)      ;продолжительность вывода
        INC   HL
        LD    (IX+8),A
        LD    (IX+2),L
        LD    (IX+3),H
; Восстановление текущего адреса данных для изменения тона
        LD    A,(IX+11)
        LD    (IX+13),A
        LD    A,(IX+12)
```

```
        LD    (IX+14),A
GETS5   LD    L,(IX+13)
        LD    H,(IX+14)
GETS6   LD    A,(HL)
        INC   HL
        CP    128         ;задание частоты тона
        JR    NZ,GETS7
        LD    A,(HL)
        LD    (IX+4),A
        INC   HL
        LD    A,(HL)
        AND   15
        LD    (IX+5),A
        INC   HL
        JR    GETS6
GETS7   CP    129         ;задание частоты шума
        JR    NZ,GETS8
        LD    A,(HL)
        AND   31
        LD    (IX+6),A
        JR    GETS6
GETS8   CP    130         ;метка нового начала
        JR    NZ,GETS9
        LD    (IX+11),L
        LD    (IX+12),H
        JR    GETS6
GETS9   CP    131         ;возврат к началу
        JR    NZ,GETS10
        LD    L,(IX+11)
        LD    H,(IX+12)
        JR    GETS6
; Изменение частоты звука или шума
GETS10  LD    (IX+13),L
        LD    (IX+14),H
        LD    D,0
        BIT   7,A
        JR    Z,GETS11
        LD    D,255
GETS11  LD    E,A
        LD    L,(IX+4)
        LD    H,(IX+5)
        ADD   HL,DE
        LD    (IX+4),L
        LD    (IX+5),H
        ADD   A,(IX+6)
        LD    (IX+6),A
; Определение элементов массива DATREG, задающих частоту
        LD    (DATREG+6),A ;частота шума
        LD    A,(N_CHAN)
        ADD   A,A
        LD    E,A
        LD    A,L
        PUSH  HL
        CALL  SETREG      ;младший байт частоты тона
        POP   HL
        INC   E
        LD    A,H
        CALL  SETREG      ;старший байт частоты тона
; Формирование огибающей
        LD    L,(IX+17)
        LD    H,(IX+18)
GETS12  LD    A,(HL)
        INC   HL
        CP    128         ;повторение с начала
        JR    NZ,GETS13
```

```
        LD    L,(IX+15)
        LD    H,(IX+14)
        JR    GETS12
GETS13  LD    (IX+17),L
        LD    (IX+18),H
        AND   15
        PUSH  AF
        LD    A,(N_CHAN)
        ADD   A,8
        LD    E,A
        POP   AF
        JP    SETREG      ;задание громкости звука
```

В процедуре обработки прерываний приведенная подпрограмма будет вызываться трижды для определения характера звучания в каждом из трех каналов независимо друг от друга. Это дает возможность использовать в программе одновременно три самостоятельных источника звука, закрепив за каждым игровым объектом свой звуковой канал. Правда, здесь есть одно небольшое ограничение: поскольку средняя частота «белого» шума общая для всех трех каналов, то его лучше выводить только в какой-то один, а другие два использовать для вывода изменяющегося тона.

Вот описываемая процедура обработки прерываний:

```
SND128  PUSH  AF
        PUSH  BC
        PUSH  DE
        PUSH  HL
        PUSH  IX
        CALL  NXTSND
        POP   IX
        POP   HL
        POP   DE
        POP   BC
        POP   AF
        JP    56
NXTSND  LD    IX,CHAN_A
        XOR   A
        CALL  GETSND      ;задание переменных для канала A
        LD    IX,CHAN_B
        LD    A,1
        CALL  GETSND      ;задание переменных для канала B
        LD    IX,CHAN_C
        LD    A,2
        CALL  GETSND      ;задание переменных для канала C
; Вычисление значения регистра R7,
;  управляющего выводом в каналы тона и шума
        LD    A,(CHAN_C+10)
        AND   9           ;выделяем биты 0 и 3
        RLCA              ;сдвигаем влево
        LD    B,A         ;результат сохраняем в регистре B
        LD    A,(CHAN_B+10)
        AND   9           ;то же самое для других двух каналов
        OR    B
        RLCA
        LD    B,A
        LD    A,(CHAN_A+10)
        AND   9
        OR    B
        CPL               ;инвертируем биты
        LD    E,7         ;устанавливаем данные регистра R7 в DATREG
        CALL  SETREG
; Извлечение звука
OUTREG  LD    HL,DATREG+13
```

```
        LD    D,13
        LD    C,#FD
OUTR1   LD    B,#FF
        OUT   (C),D
        LD    B,#BF
        OUTD
        DEC   D
        RET   M           ;выход, если D < 0
        JR    OUTR1
DATREG  DEFS  14
; Задание элемента E массива DATREG значением из аккумулятора
SETREG  LD    HL,DATREG
        LD    D,0
        ADD   HL,DE
        LD    (HL),A
        RET
N_CHAN  DEFB  0           ;номер текущего канала
; Таблицы переменных для каждого канала
CHAN_A  DEFS  19
CHAN_B  DEFS  19
CHAN_C  DEFS  19
```

Наше прерывание остается дополнить процедурами включения и выключения 2-го режима. Попутно желательно выполнить и некоторые другие действия, а именно, при установке 2-го режима нужно очистить все три таблицы переменных, инициализировав их нулевым байтом, а при возврате первого режима прерываний необходимо еще убедиться, что звук выключен:

```
INITI   LD    HL,CHAN_A   ;инициализация таблиц переменных
        LD    DE,CHAN_A+1
        LD    BC,19*3-1
        LD    (HL),0
        LDIR
        LD    HL,SND128   ;установка прерывания
IMON    .........
STOPI   CALL  IMOFF       ;возврат к 1-му режиму
        LD    A,#FF       ;выключение звука
        LD    E,7
        CALL  SETREG      ;запись в регистр сопроцессора R7
                          ; значения #FF
        JP    OUTREG
IMOFF   .........
```

Порядок действий при использовании описанной программы должен быть следующим.
В начале работы нужно включить 2-й режим прерываний, вызвав процедуру INITI.
Извлечение очередного звука необходимо начинать с определения некоторых переменных в таблицах CHAN_A, CHAN_B или CHAN_C. Для этого нужно записать в первые два байта таблицы, соответствующей выбранному каналу, адрес начала основного блока данных и то же значение продублировать в следующих двух байтах таблицы. Затем указать количество повторений звука по смещению +9, а элементы таблицы +8 и +7 инициализировать байтом 1 (переменную по смещению +7 нужно задавать обязательно в последнюю очередь, так как именно она «запускает» звук). По окончании работы (или если в программе предусмотрены обращения к дисководу) следует восстановить стандартный режим обработки прерываний и выключить звук, обратившись к подпрограмме STOPI.

Продемонстрируем применение описанной процедуры извлечения звуков на примере небольшой игры, которую назовем БИТВА С НЛО. По земле катается грузовик с зенитной лазерной установкой, который методично расстреливает маячащую в небе «летающую тарелку» (рис. 10.1). НЛО также не остается внакладе и отвечает хоть и малоприцельным, но зато плотным веерным огнем.

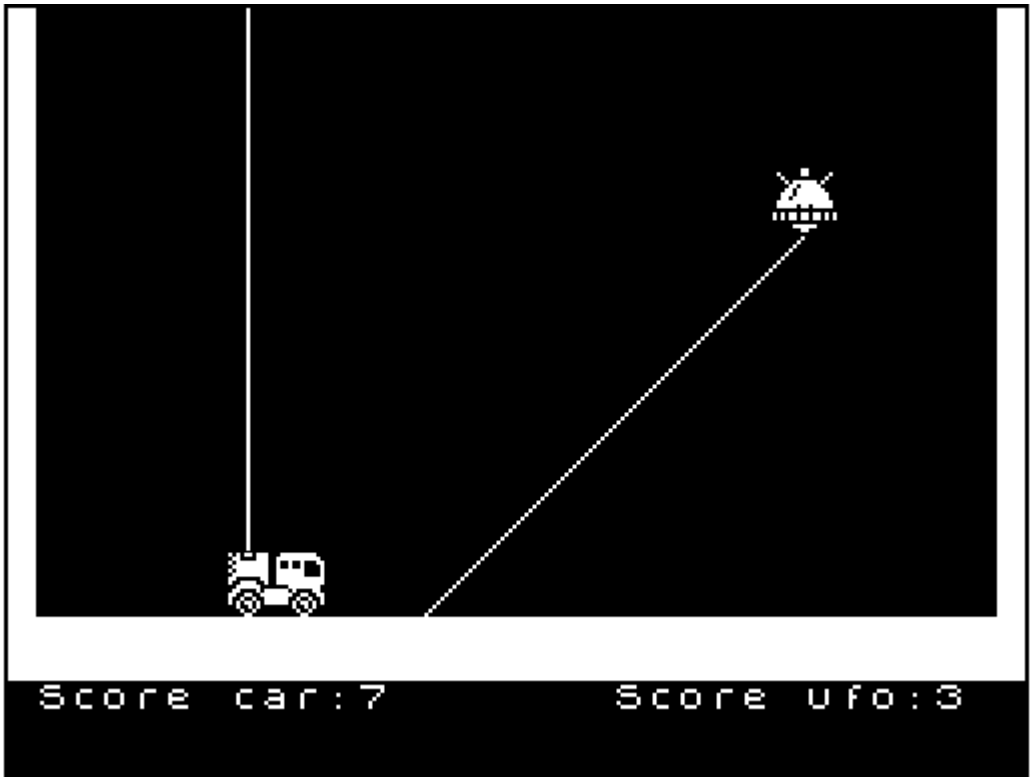


Рис. 10.1. Программа БИТВА С НЛО

Вначале создадим акустическое сопровождение программы. Нам понадобятся такие звуки: выстрелы с обеих сторон, попадания в НЛО и в лазерную установку, а также соударение НЛО со «стенкой», ограничивающей игровое поле. Учитывая характер звуков, а также и то, что шум можно выводить только в один из каналов, составим блоки данных по правилам, описанным выше.

Начнем с выстрелов управляемой игроком лазерной установки. Для получения этого звука используем «белый» шум. Этот и следующий звуки закрепим за каналом музыкального сопроцессора A.

```
D_SND1 DEFB  1
       DEFW  FREQ1
       DEFB  2
       DEFW  ENV1
       DEFB  3,8,2,3,8,1,0
FREQ1  DEFB  129,0,130,5,131 ;изменение частоты
ENV1   DEFB  15,14,12,128    ;изменение громкости
```

Для создания эффекта попадания в автомобиль с лазерной установкой также будем выводить шум, но для большей убедительности в самом начале звука дадим короткий сигнал низкого тона:

```
D_SND2 DEFB  1
       DEFW  FREQ2
       DEFB  2
       DEFW  ENV2
       DEFB  3,1,2,1
       DEFW  FREQ3
       DEFB  3,8,20,0
FREQ2  DEFB  128
       DEFW  1000
       DEFB  130,10,131
FREQ3  DEFB  129,3,130,1,131
```

```
ENV2   DEFB  15,14,15,12,15,14,12,10,14,12
       DEFB  9,8,7,6,5,4,3,2,11,0,128
```

Звуки, сопровождающие выстрелы «летающей тарелки» и попадания в нее будем выводить в канал B. Попробуем сымитировать эти звучания изменением чистого тона. Блок данных для формирования частоты и огибающей «выстрела» может выглядеть примерно так:

```
D_SND3 DEFB  1
       DEFW  FREQ4
       DEFB  2
       DEFW  ENV3
       DEFB  3,1,15,0
FREQ4  DEFB  128
       DEFW  200
       DEFB  130,20,131
ENV3   DEFB  15,14,14,13,12,12,11,128
```

Звук, подражающий попаданию в НЛО, должен быть более протяжным, поэтому и блок данных, описывающий огибающую окажется несколько длиннее:

```
D_SND4 DEFB  1
       DEFW  FREQ5
       DEFB  2
       DEFW  ENV4
       DEFB  3,1,24,0
FREQ5  DEFB  128
       DEFW  700
       DEFB  130,100,131
ENV4   DEFB  15,14,14,15,15,12,10,11,8,7
       DEFB  7,5,6,7,10,12,14,15,10,8,128
```

Все предыдущие звуки были достаточно глухими, поэтому удар НЛО о «стенку» сделаем напоминающим хрустальный звон. Если вам это покажется уж слишком нелогичным, попытайтесь создать более подходящее звучание самостоятельно. Этот эффект будет выводиться в канал C:

```
D_SND5 DEFB  1
       DEFW  FREQ6
       DEFB  2
       DEFW  ENV5
       DEFB  3,1,9,0
FREQ6  DEFB  128
       DEFW  20
       DEFB  130,10,-15,40,-30,-5,131
ENV5   DEFB  15,9,12,10,15,13,128
```

В игре нам понадобится создать изображения грузовика и НЛО. Общую часть программы мы собираемся написать на Бейсике, поэтому проще всего для вывода графических изображений вновь обратиться к символам UDG. Можно было бы закодировать их и в бейсик-программе, но, во-первых, как вы знаете, циклы в интерпретаторе - самое больное место и выполняются они в десятки, если не в сотни раз медленнее, чем в машинных кодах. Но самое главное даже не в этом, а в том, что область памяти, в которой располагаются коды определяемых символов, занята командами перехода на процедуру обработки прерываний (адреса 65524...65526 и 65535, соответствующие положению последних двух символов - **T** и **U**). Поэтому, чтобы застраховаться от неожиданностей, коды символов лучше расположить в ассемблерной части программы, что мы и делаем:

```
SETUDG LD    HL,UDG
       LD    (23675),HL
       RET
; НЛО (A, B, C и D)
UDG    DEFB  1,65,32,23,13,27,59,55
       DEFB  128,130,4,232,240,248,252,252
       DEFB  127,125,0,173,173,0,7,1
       DEFB  254,190,0,181,181,0,224,128
```

```
; Лазерная установка (E, F, G, H, I и J)
        DEFB  173,97,191,127,191,127,192,158
        DEFB  199,207,201,201,207,207,207,79
        DEFB  252,254,35,33,225,225,255,255
        DEFB  63,97,204,146,173,37,18,12
        DEFB  1,126,255,126,126,0,0,0
        DEFB  254,134,51,73,181,148,72,48
```

Теперь остается собрать все подпрограммы в один блок так, чтобы их удобно было вызывать из Бейсика. Кроме того, допишем недостающие процедуры инициализации звуков:

```
        ORG   60000
; 60000 - включение 2-го режима прерываний и инициализация массива DATREG
        JP    INITI
; 60003 - выключение 2-го режима прерываний и звука
        JP    STOPI
; 60006 - выстрел лазерной установки
        JP    SND1
; 60009 - попадание в лазерную установку
        JP    SND2
; 60012 - выстрел НЛО
        JP    SND3
; 60015 - попадание в НЛО
        JP    SND4
; 60018 - соударение НЛО со «стенкой»
        JP    SND5
; 60021 - символы UDG
SETUDG .........
SND5    LD    HL,D_SND5
        LD    (CHAN_C),HL
        LD    (CHAN_C+2),HL
        LD    A,1
        LD    (CHAN_C+9),A
        LD    (CHAN_C+8),A
        LD    (CHAN_C+7),A
        RET
SND4    LD    HL,D_SND4
        JR    SND3_1
SND3    LD    HL,D_SND3
SND3_1  LD    (CHAN_B),HL
        LD    (CHAN_B+2),HL
        LD    A,1
        LD    (CHAN_B+9),A
        LD    (CHAN_B+8),A
        LD    (CHAN_B+7),A
        RET
SND2    LD    HL,D_SND2
        LD    A,1
        JR    SND1_1
SND1    LD    HL,D_SND1
        LD    A,3
SND1_1  LD    (CHAN_A),HL
        LD    (CHAN_A+2),HL
        LD    (CHAN_A+9),A
        LD    A,1
        LD    (CHAN_A+8),A
        LD    (CHAN_A+7),A
        RET
; Блоки данных, описывающие каждый из пяти используемых в программе звуков
D_SND1 .........
D_SND2 .........
D_SND3 .........
D_SND4 .........
D_SND5 .........
; Далее следуют уже описанные процедуры прерывания
```

А вот бейсик-программа самой игры:

```
 10 REM *** НЛО ***
 20 BORDER 0: PAPER 0: INK 7: CLEAR 59999
 30 RANDOMIZE USR 15619 : REM : LOAD "snd_ufo"CODE
 40 LET cd=USR 60021: REM *** UDG ***
 50 LET cd=USR 60000: REM *** Прерывания ***
 60 REM --------------
 70 CLS
 80 FOR n=19 TO 20: PRINT AT n,0; PAPER 4; TAB 31;" ": NEXT n
 90 FOR n=0 TO 18: PRINT AT n,0; PAPER 4;" ";AT n,31;" ": NEXT n
100 LET kl=0: LET ik1=7: LET ik2=ik1: LET pow=0: LET pow1=pow:
    LET x1=4: LET y1=1: LET w=0: LET xp=0: LET yp=0: LET s1=10:
    LET s=10: LET x=4: LET y=1
110 LET a$=INKEY$
120 IF a$="p" AND s<28 THEN LET s=s+1: GO TO 180
130 IF a$="o" AND s>1 THEN LET s=s-1: GO TO 190
140 IF a$=" " THEN LET cd=USR 60006: GO TO 230
150 IF a$="e" THEN LET cd=USR 60003: STOP
160 REM --------------
170 GO TO 200
180 PRINT AT 17,s1;" ";AT 18,s1;" ": GO TO 200
190 PRINT AT 17,s1+2;" ";AT 18,s1+2;" "
200 INK 6: PRINT AT 17,s;"EFG";AT 18,s;"HIJ"
210 LET s1=s
220 LET ik2=7: GO TO 280
230 IF INT x=s OR INT x+1=s THEN LET yy=INT y+1: LET pow1=pow1+1:
    LET cd=USR 60015: LET ik2=2: GO TO 250
240 LET yy=0
250 GO SUB 270: OVER 1: GO SUB 270: OVER 0
260 GO TO 280
270 INK 7: PLOT s*8+5,41: DRAW 0,126-yy*8: RETURN
280 REM ----- нло ----
290 IF w=0 THEN LET xp= (RND*1.5+.5): LET yp= (RND*1.5):
    LET w=INT (RND*25+5): LET xp=xp*SGN (RND*4-2):
    LET yp=yp*SGN (RND*4-2)
300 IF x+xp>=1 AND x+xp<30 AND y+yp>=0 AND y+yp<=12
    AND w>0 THEN LET x=x+xp: LET y=y+yp: LET w=w-1: LET kl=0:
    GO TO 330
310 IF ((INT (x+xp)<1 AND INT x=1) OR (INT x=29 AND INT (x+xp)>29))
    AND kl=0 THEN LET kl=1: LET cd=USR 60018
320 LET w=0: GO TO 290
330 PRINT AT INT y1,INT x1;"··";AT INT y1+1,INT x1;"··"
340 INK 5: PRINT AT INT y,INT x;"AB";AT INT y+1,INT x;"CD"
350 LET x1=x: LET y1=y
360 IF RND*10<3 THEN LET cd=USR 60012: LET xf=RND*255:
    GO SUB 410: OVER 1: GO SUB 410: OVER 0: LET xf1=INT (xf/8):
    IF xf1=s OR xf1=s+1 OR xf1=s+2 THEN LET pow=pow+1:
    LET cd=USR 60009: LET ik1=2: GO TO 380
370 LET ik1=7
380 PRINT AT 21,1; INK ik2;"Score car:";INT pow1;" "
390 PRINT AT 21,19; INK ik1;"Score ufo:";INT pow;" "
400 GO TO 110
410 INK 2: PLOT INT (x)*8+7,159-INT (y)*8:
    DRAW xf-(INT (x)*8+7),25-(159-INT (y)*8): RETURN
```

Прокомментируем немного строки этой программы и поясним значение некоторых переменных.

40 - «включение» символов `UDG`;
50 - включение 2-го режима прерываний;
80...90 - рисование грунта;
100 - задание начальных значений переменных:

`kl` - управление звуком при подлете НЛО к стенке (`kl=0` - звук разрешен, `kl=1` - запрещен);
`ik1`, `ik2` - цвет надписей;
`pow` - количество попаданий НЛО;
`pow1` - количество попаданий автомобиля;
`x1`, `y1` - координаты для удаления предыдущего изображения НЛО;
`w` - количество перемещений НЛО до изменения направления движения;
`xp`, `yp` - приращения координат НЛО;
`s1` - координата для удаления предыдущего изображения автомобиля;
`s` - текущая координата автомобиля;
`x`, `y` - текущие координаты НЛО.

110...150 - опрос клавиатуры;
180...200 - восстановление фона позади движущегося автомобиля и перемещение его в новое положение;
230 - проверка попадания луча лазера в НЛО;
250 - рисование луча лазера;
290...400 - управление полетом НЛО;
410 - рисование следа от выстрела НЛО.

МУЗЫКАЛЬНЫЙ РЕДАКТОР WHAM FX

Кодирование вручную звуковых эффектов, а тем более - музыки, дело весьма утомительное. Хотя такой способ при наличии определенного опыта в конечном итоге и дает наилучшие результаты, но для начинающих он мало пригоден. А что говорить о простых пользователях, общающихся с компьютером только на уровне игровых или, в лучшем случае, прикладных программ. Поэтому стоит сказать хотя бы несколько слов о программе, специально предназначенной для создания компьютерной музыки - музыкальном редакторе.

В первой книге серии «Как написать игру для ZX Spectrum» мы рассказали о работе с редактором Wham. Этот же редактор был достаточно подробно описан и в книге [2]. Поэтому здесь мы не станем еще раз повторяться, а объясним, как работать с другой версией этой программы, специально рассчитанной на возможности музыкального сопроцессора.

Редактор Wham FX был создан тем же автором и внешне не слишком отличается от своего предшественника. Не особенно сильно изменен и принцип управления, а также способ ввода мелодии. Но, конечно, имеются и некоторые существенные отличия, на которых в основном мы и хотим заострить ваше внимание.

После загрузки программы вы услышите мелодию, демонстрирующую возможности редактора. Нажав любую клавишу, можно прервать прослушивание и попасть в главное меню, многие пункты которого повторяют функции первой версии Wham. Среди них уже знакомые LOAD TUNE и SAVE TUNE (загрузка и сохранение пьесы), SET TEMPO (изменение темпа), HELP PAGE (подсказка) и EDIT MODE (режим ввода и редактирования). Остальные две опции - SYST MENU (системное меню) и ENVELOPES (формирование огибающих звука) - мы рассмотрим ниже, а сейчас сразу перейдем к редактированию мелодии, нажав клавишу **6**.

Большая часть экрана в режиме редактирования (рис. 10.2) занята нотными линейками, на которых отображается вводимая вами мелодия. Нижнюю часть экрана занимает изображение клавиатуры фортепиано, которое помогает установить соответствие между клавишами компьютера и вводимыми звуками. В средней части экрана расположено несколько окон: три небольших окна слева символизируют характер звучания в каждом из трех каналов; в четвертом окне указываются номера редактируемого канала (CHN) и текущей октавы (OKT); в следующем, самом большом окне выводится информация об используемых в произведении эффектах; последнее окно, изображающее прибор со стрелкой, носит скорее декоративный характер - при проигрывании мелодии стрелка постоянно прыгает в такт звукам, отражая уровень громкости.

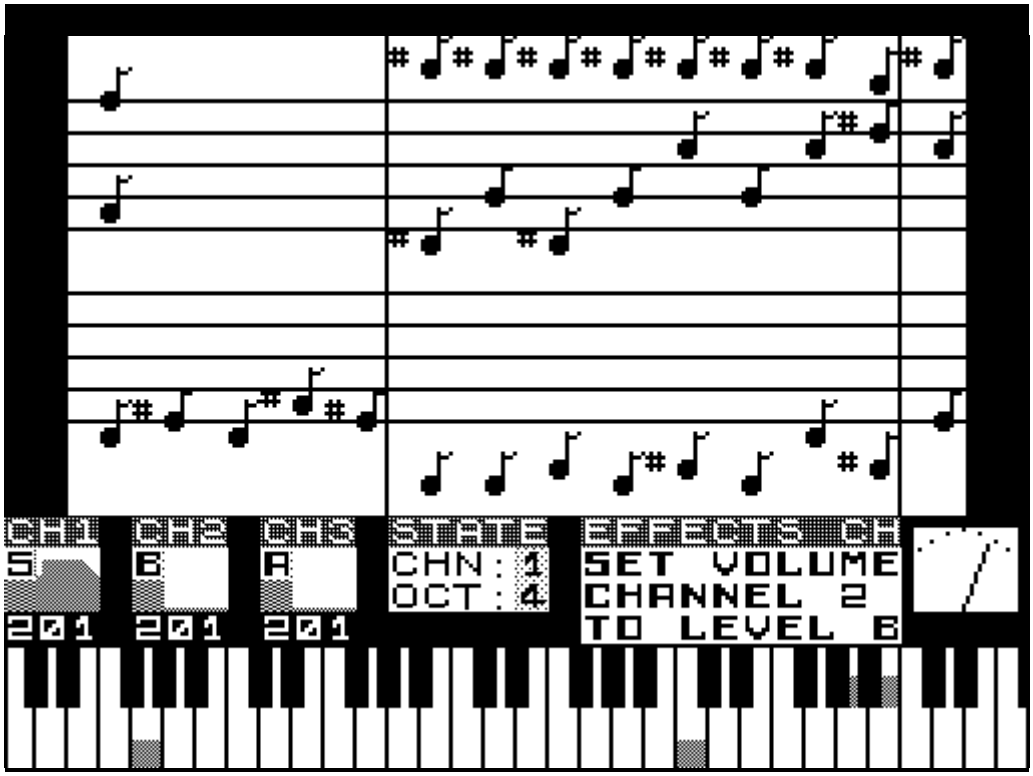


Рис. 10.2. Режим редактирования

Прежде чем начинать вводить новую мелодию, нужно убрать из памяти старую, оставшуюся после загрузки программы или от предыдущих упражнений. Нажмите клавишу **7** и на запрос ERASE CURRENT TUNE (Y/N)? - удалить текущую мелодию (да/нет)? - ответьте утвердительно нажатием клавиши **Y**. Теперь можно записывать ноты.

Ввод мелодии в Wham FX в принципе ничем не отличается от записи музыки в первой версии программы. Для получения звуков здесь также используются клавиши двух нижних рядов и **Enter** - для ввода пауз, октавы переключаются клавишами **1...4**, а переход к редактированию другого голоса происходит при нажатии клавиши **T**. Правда, данная версия рассчитана на компьютер Spectrum 128, а следовательно, на расширенную клавиатуру, поэтому нота ДО извлекается нажатием клавиши **Caps Lock** (на обычной клавиатуре - **Caps Shift**/**2**), а также задействованы кнопки с символами запятой и точки. Клавиши **Caps Shift** и **Symbol Shift** служат здесь для других целей, поэтому при вводе звуков не используются.

При записи музыки вы можете по мере надобности сдабривать ее различными шумовыми эффектами, имитирующими ударные инструменты. Таких инструментов в одной пьесе можно иметь до девяти, а вставляются подобные звуки при одновременном нажатии клавиши **Symbol Shift** и одной из клавиш второго ряда сверху от **Q** до **O**.

Если вы ошиблись при вводе очередного звука, вернуться на одну позицию назад можно с помощью клавиши **Delete** (**Caps Shift/0**), а для быстрой прокрутки назад на несколько тактов воспользуйтесь клавишей **True Video** (**Caps Shift/3**). Для продвижения вперед нажимайте клавиши **O** (быстро) или **P** (медленно). Чтобы прослушать полученную музыку, нужно вернуться в самое начало пьесы, нажав **R**, а затем включить проигрывание клавишей **Q**.

Наиболее интересной особенностью редактора Wham FX является возможность изменения характера звучания каждого голоса в отдельности. Для этого нужно подвести курсор к тому месту в пьесе, начиная с которого вы хотите получить иной звук (первая нота фрагмента должна появиться у правого края экрана) и нажать клавишу **Extend Mode** (**Caps Shift/Symbol Shift**). Внизу экрана появится дополнительное меню, состоящее из пяти пунктов:

ENVELOPE VOLUME BLANK SLIDE LOOP

Выбор интересующей функции осуществляется нажатием клавиши, соответствующей первой букве слова. Поясним назначение каждого из этих пунктов.

После выбора ENVELOPE компьютер даст два дополнительных запроса: какой формы должна быть огибающая (нужно ввести число от 1 до 7) и на какой голос данный эффект будет распространяться (нажмите клавишу от **1** до **3** или **0**, если хотите иметь одинаковое звучание во всех трех голосах).

В отличие от предыдущего пункта, VOLUME задает постоянный уровень громкости. При выборе этой опции нужно сначала ввести шестнадцатеричное число от 1 до F, соответствующее желаемой громкости, а затем опять же указать голос, к которому данное изменение относится.

Опция SLIDE имитирует такой распространенный в эстрадной музыке прием исполнения, как BEND. При этом звук плавно изменяется по высоте в пределах нескольких полутонов, повышаясь или же наоборот, понижаясь. Компьютер попросит сначала ввести интервал (от 0 до 7 полутонов), на который звук «поедет», затем направление (UP или DOWN - вверх или вниз) и, как и для предыдущих пунктов, номер канала.

BLANK используется в том случае, если вы по ошибке поставили эффект не в том месте, где хотели. Никаких дополнительных запросов в этой опции не предусмотрено.

Последний пункт LOOP служит для установки метки цикла в канале эффектов FX и обязательно должен использоваться перед началом компиляции пьесы. Однако до этих пор прибегать к нему не следует, так как если вы решите продолжить ввод мелодии, снять поставленную метку не удастся.

Немного потренировавшись во вводе нот и изменении их звучания, можно приступать к программированию настоящей музыки. Чтобы помочь вам в этом нелегком предприятии, предлагаем сначала ввести небольшой фрагмент, приведенный на рис. 10.3 и в табл. 10.2. Просим музыкальных критиков не придираться к правописанию нот, так как рисунок отражает то, что вы увидите на экране монитора, а не то, как должны быть записаны ноты для исполнения пьесы грамотными музыкантами. В таблице выписана последовательность нажатия клавиш при вводе звуков каждого голоса. Сокращение Okt. обозначает октаву, Ent - клавишу **Enter**, SS - **Symbol Shift** (запись SS/E, например, обозначает одновременное нажатие **Symbol Shift** и **E**), а буквы и цифры справа от обозначения клавиш указывают на установку того или иного эффекта (V - VOLUME, E - ENVELOPE).

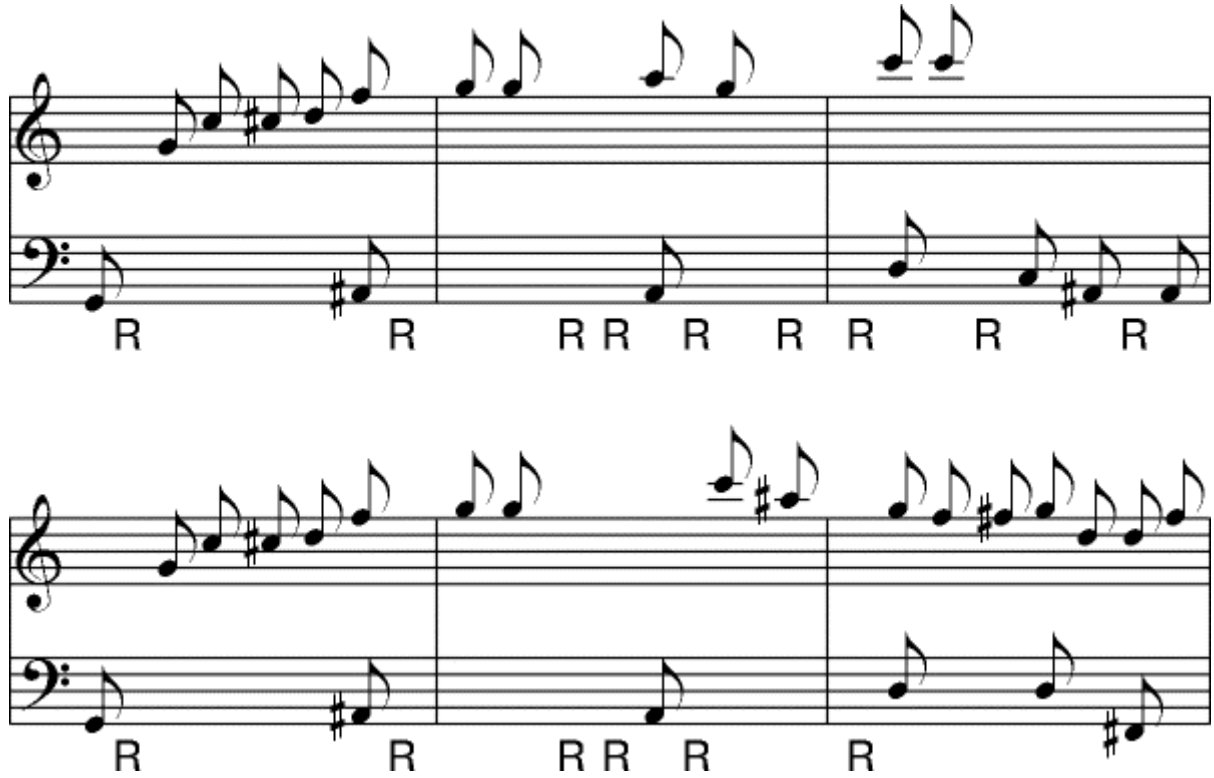


Рис. 10.3. Ноты пьесы

Таблица 10.2. Ввод пьесы

ГОЛОС 1			ГОЛОС 2	ГОЛОС 3		
Okt. 3	Ent		Ent	Okt. 1	V	E5
	Ent		Ent		Ent	
	V	E7	Ent		Ent	
	M		SS / O		Ent	
Okt. 4	A		Ent		Ent	
	Z		Ent		Ent	
	C		Ent		H	
	Ent		Ent		Ent	

	V	VA	Ent		Ent		
	V	V5	SS / E		Ent		
	Ent		Ent		Ent		
	Ent		Ent		Ent		
	H	E5	Ent		B		
	Ent		Ent		Ent		
	V		Ent		Ent		
	Ent		SS / O		Ent		
	Ent		Ent		Ent		
	M	VA	Ent	Okt. 2	Z		
	M	V5	Ent		Ent		
	Ent		Ent		Ent		
	Ent		SS / Q	Okt. 1	M		
	Ent		SS / O		H		
	Ent		Ent		Ent		
	Ent		SS / Q		B		
Okt. 3	Ent		Ent		V		

	Ent		Ent		Ent		
	V	E7	Ent		Ent		
	M		SS / O		Ent		
Okt. 4	A		Ent		Ent		
	Z		Ent		Ent		
	C		Ent		H		
	Ent		Ent		Ent		
	V	VA	Ent		Ent		
	V	V5	SS / E		Ent		
	Ent		Ent		Ent		
	Ent		Ent		Ent		
	Ent		Ent		B		
	Ent		Ent		Ent		
	M	E5	Ent		Ent		
	H		SS / O		Ent		
	Ent		Ent		Ent		
	V		Ent	Okt. 2	Z		

	C		SS / O		Ent		
	F		Ent		Ent		
	V		SS / W		Z		
	Z	VF	SS / R	Okt. 1	F		
	Z	V7	Ent		Ent		
	C	E7	SS / Q		Ent		

После того как мелодия введена в память, желательно сохранить ее на ленте или диске. Для этого выйдите в главное меню, нажав клавишу **6**, и прежде чем воспользоваться пунктом 2 (SAVE TUNE) выберите устройство, на котором пьеса будет сохранена. Это делается так. Войдите в системное меню, нажав клавишу **3**. Перед вами предстанет новый список, в котором нас сейчас интересует третий пункт. В нем белым цветом будет выделена надпись BETA DISK или CASS TAPE. Если надпись не соответствует желаемой, нажмите **3**, а затем выберите нужное устройство, иначе нажмите клавишу **0** для возврата в главное меню. Теперь остается сохранить пьесу, предварительно указав имя файла (при работе с диском возможен также вывод на экран каталога).

Когда работа над пьесой завершена, ее нужно скомпилировать, чтобы получить файл, исполняемый независимо от редактора. Вернитесь в режим редактирования, прокрутите мелодию до конца и расставьте метки цикла во всех голосах: нажмите клавишу **W** и на запрос SET LOOP HERE? (Y/N) ответьте клавишей **Y**, затем повторите то же самое для других двух голосов. Не забудьте поставить метку также и в канале эффектов FX, нажав **Extend Mode** и выбрав в дополнительном меню пункт LOOP.

Расставив метки, переходите в главное меню, а затем в системное, откуда вызывается опция COMPILE & SAVE (компиляция и сохранение). Это пятый пункт системного меню. Если все метки расставлены и компиляция прошла успешно, на экран выводится ряд сообщений, из которых можно понять, что для проигрывания мелодии нужно ввести из Бейсика команду

```
RANDOMIZE USR 64000
```

или аналогичную ей, а из ассемблера полученная подпрограмма вызывается соответственно командой

```
        CALL  64000
```

Тут же сообщается, что для регулирования темпа исполнения нужно записать некоторое число по адресу 64062 (при исполнении нашей пьесы это число будет равно примерно 230). Изучив выданную информацию, наберите имя файла для сохранения скомпилированной мелодии и нажмите **Enter**.

На этом можно было бы поставить точку в данном кратком описании, но все же хочется сказать пару слов и о других возможностях, предоставляемых редактором Wham FX. Если вас не слишком устраивают предлагаемые программой формы огибающих звуков, то вы можете с помощью пункта 4 главного меню исправить положение и создать свои собственные уникальные звуки.

Нажав клавишу **4**, вы попадаете во встроенный редактор для формирования огибающих. В верху экрана выстроятся 8 диаграмм, показывающих существующие формы звуков: первые 7 можно заказывать в дополнительном меню режима редактирования, а восьмой, помеченный двумя звездочками, относится к шумовым эффектам. Нажмите цифровую клавишу, соответствующую звуку, который вы хотите изменить. На экране появится выбранная диаграмма в увеличенном масштабе (рис. 10.4), а под ней - курсор в виде стрелки. Управляя клавишами **Left** (**Caps Shift/5**) и **Right** (**Caps Shift/8**) переместите стрелку в нужную точку огибающей и отрегулируйте громкость с помощью клавиш **Down** (**Caps Shift/6**) и **Up** (**Caps Shift/7**). Получив таким образом нужную форму огибающей закончите редактирование, нажав **Enter**.

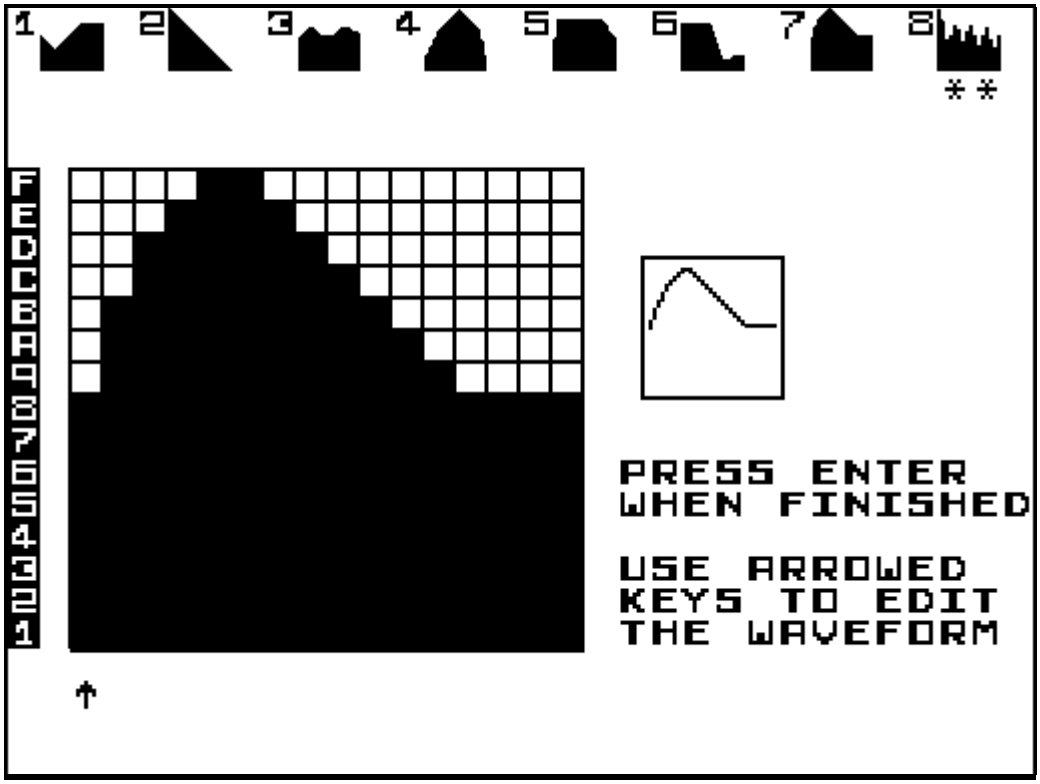


Рис. 10.4. Формирование огибающей

Другая возможность касается «ударных инструментов». Их звучание также можно варьировать в некоторых пределах. Выберите в системном меню опцию SET PRESET NOISE VALUES, нажав клавишу **1**. На экране вы увидите таблицу, показанную на рис. 10.5.
В первой графе указан порядковый номер эффекта, а во второй - клавиша, за которой этот эффект закреплен в режиме редактирования мелодии. Нажав соответствующую цифровую клавишу, вы сможете изменить другие параметры, обозначенные в таблице. Сначала появится запрос об установке нового значения для графы FREQUENCY (частота), на который нужно ввести число от 0 до 31. Затем вводится номер огибающей (клавиши **1...8**), помещаемый в графу ENVELOPE. Если вы хотите задать звук постоянной громкости, введите на этот запрос 0. В этом случае компьютер попросит указать уровень звука (графа VOLUME) вводом шестнадцатеричного числа от 0 до F.

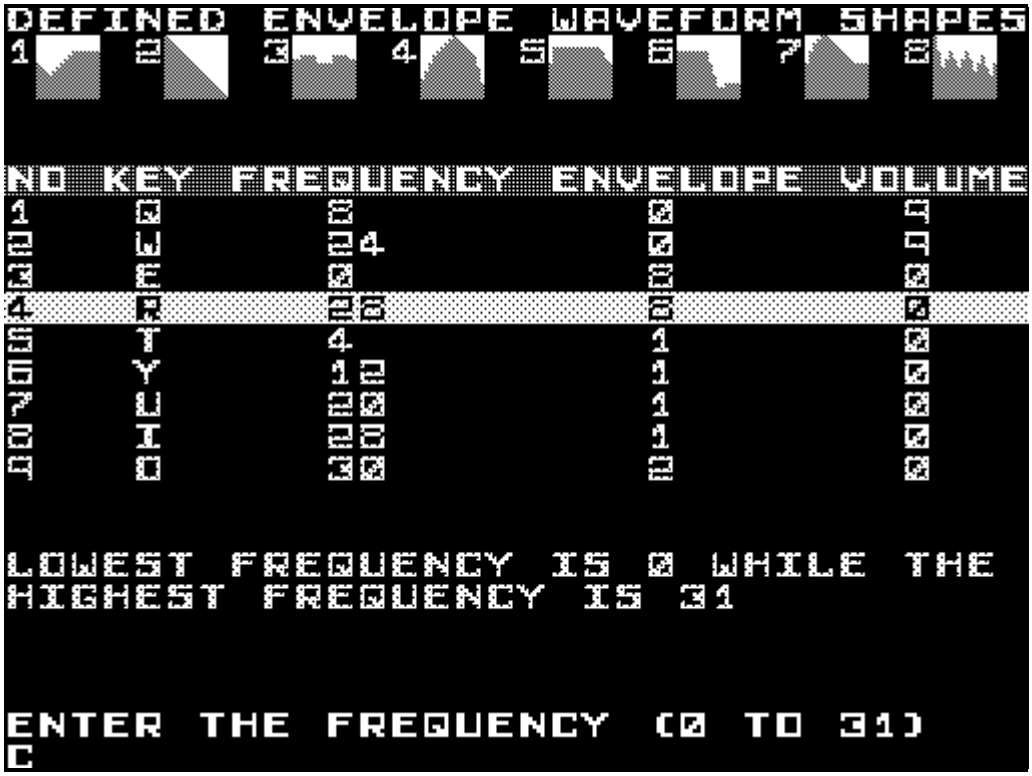


Рис. 10.5. Редактирование шумовых эффектов

Аналогично можно отредактировать любой из девяти эффектов, а для возврата в главное меню нужно нажать клавишу **0**.

В системном меню есть еще одна интересная функция - SET CHANNEL LOOP PARAMETERS, служащая для установки начальных меток циклов в каждом канале.
К услугам этого пункта полезно прибегать в тех пьесах, где мелодия начинается не с сильной доли (то есть не на счет «раз»), а с затакта. Нажмите клавишу **2** и укажите, в каком канале и с какого смещения мелодия в выбранном голосе будет начинаться при повторении. Начальные метки задаются для каждого канала в отдельности (в том числе и для канала эффектов FX), а смещения могут иметь как положительные, так и отрицательные значения. Добавим, что ошибиться в расстановке этих меток не страшно, ибо в любой момент их можно переместить как вперед, так и назад (если ввести отрицательное число).

ГЛАВА ОДИННАДЦАТАЯ,

рассказывающая о некоторых дополнительных возможностях ассемблера GENS4

Во второй главе были перечислены лишь самые необходимые команды редактора GENS. До сих пор этого было вполне достаточно для ввода, редактирования и трансляции процедур и фрагментов программ, предложенных в книге. Но когда вы начнете писать свои собственные игры, они наверняка окажутся значительно больших размеров. Возможно даже, что исходные тексты в несколько раз превзойдут по объему всю имеющуюся в наличии память и их придется разбивать на части. И вот тогда вы почувствуете, что описанных возможностей GENS явно маловато. Поэтому в данной главе мы приводим описание некоторых других

весьма полезных команд редактора, а также способы сокращения исходного текста и придания ему большей наглядности.

ФУНКЦИЯ ПОИСКА/ЗАМЕНЫ

Пока вы имеете дело с небольшими обрывками программ, что-то подправить в них не представляет особого труда. Но попробуйте-ка перетряхнуть огромный текст, состоящий из многих сотен инструкций и найти в нем нужное место. Вряд ли это занятие доставит вам большое удовольствие, а когда вы наконец отыщите заветную строчку, охота заниматься с программой может и вовсе улетучиться. Чтобы уберечь своих пользователей от нервного истощения, вызванного такой работой, фирма HiSoft включила в редактор GENS команду, предназначенную для поиска нужной последовательности символов, да еще и с возможностью замены ее на другую. Эта команда записывается так:

```
F[номер начальной строки],[номер конечной строки],
[текст для поиска][,текст для замены]
```

Как и раньше, квадратные скобки указывают на необязательность параметров. Если какой-то из них не задан, то он берется из последней введенной команды. Вообще же лучше на всякий случай перечислять все элементы команды, особенно при замене текста.

Предположим, вам нужно в тексте программы найти все места, где встречается метка LABEL. Введите в редакторе команду

```
F1,20000,LABEL
```

Как только функция найдет последовательность символов, совпадающую с заданной (LABEL), на экране появится строка текста, содержащая эту последовательность, и GENS перейдет в режим редактирования. При этом курсор для удобства автоматически устанавливается в самое начало найденного текста. После этого у вас есть два варианта дальнейших действий: либо закончить поиск, нажав клавишу **Enter**, либо продолжить его, для чего нет надобности набирать команду заново, а достаточно нажать клавишу **F**. Естественно, вам ничто не мешает сразу же внести в текст какие-то изменения, но если они должны быть везде однотипными, проще задать в команде F также и последний параметр. Например, чтобы заменить все имена LABEL на METKA введите в редакторе строку

```
F1,20000,LABEL,METKA
```

Внешне поведение функции при этом не изменится: при нахождении первого имени строка точно так же будет вызываться на редактирование, а курсор указывать на первый символ слова LABEL. Вы так же можете продолжать вносить изменения вручную и по желанию продолжать поиск без изменений текста или прервать выполнение команды. Но теперь у вас появилась и иная возможность. Если нажать клавишу **S**, то слово LABEL мгновенно заменится словом METKA, а на экране появится следующая найденная строка.

УПРАВЛЕНИЕ ТРАНСЛЯЦИЕЙ

Основная задача ассемблера - трансляция исходных текстов, поэтому о том, как получается исполняемый файл, не помешает знать несколько больше того, что вам уже известно. При обработке небольших программ обычно бывает вполне достаточно просто ввести команду A,

но иногда может потребоваться, например, узнать адреса некоторых меток, вывести листинг ассемблирования на экран или распечатать его на принтере. А как быть, если какая-то часть программы должна размещаться в экранной области памяти или на месте системных переменных? Или размер программы получается настолько большим, что перекрывает не только исходный текст, но и коды самого ассемблера? Оказывается, GENS позволяет справиться и с подобными трудностями! Нужно только дать ему соответствующее задание. Для этого необходимо указать в команде A дополнительные параметры, которые мы раньше пропускали.

Ключи ассемблирования

Первое число, следующее за командой A, называют ключами ассемблирования, потому что в зависимости от него трансляция протекает с теми или иными условиями. Ключи похожи на флаги, так как каждому из них соответствует определенный бит задаваемого значения. При объединении нескольких битов можно получить комбинацию различных условий. Перечислим значения всех используемых ключей при установке соответствующего бита в 1:

бит 0 (1) - заставляет ассемблер вывести таблицу адресов меток и значений констант в конце второго прохода трансляции;

бит 1 (2) - произвести проверку синтаксиса строк программы, не создавая при этом машинного кода;

бит 2 (4) - вывести на втором проходе листинг ассемблирования программы;

бит 3 (8) - на время трансляции назначить вывод вместо экрана на принтер;

бит 4 (16) - реально размещать машинный код сразу за таблицей меток (адрес, указанный в директиве ORG влияет только на создание ссылок в программе);

бит 5 (32) - не делать проверки расположения кода в памяти (обычно же максимальная верхняя граница задается системной переменной UDG и может быть изменена командой редактора U).

Поясним смысл некоторых наиболее важных ключей и покажем, как ими пользоваться на практике. Первый ключ бывает совершенно необходим, по крайней мере, в двух случаях: при отладке достаточно больших программ, чтобы знать, где искать ту или иную процедуру и для определения адресов подпрограмм и переменных при использовании машинного кода совместно с программами на других языках или для связи различных модулей, транслируемых раздельно (что поделаешь, память Speccy весьма ограничена и иногда поневоле приходится изощряться). В таких программах, как БИТВА С НЛО или ГЕНЕРАТОР СПРАЙТОВ мы предложили простой способ выбора необходимой процедуры из пакета, но он не всегда может оказаться пригодным. Более того, в ГЕНЕРАТОРЕ СПРАЙТОВ переменные располагались в буфере принтера, а это в некоторых случаях не допускается. Например, в компьютере ZX Spectrum 128, как мы уже говорили, в этой области содержится важная информация, разрушать которую никак нельзя. Вот здесь-то и поможет ключ 1. Разместите все переменные внутри программы и оттранслируйте ее командой A1. Когда ассемблирование закончится, на экран (или на принтер, если задать режим A9 - 1+8) будет выдана таблица меток, в которой найдите нужное имя, а рядом с ним - адрес. Для обращения к этому адресу из Бейсика вам останется только перевести шестнадцатеричное значение в десятичное.

Ключи 2 и 4 бывают особенно полезны на этапе отладки, а вот без ключа 16 не обойтись при трансляции больших программ или если машинный код должен располагаться в недоступных иными средствами областях памяти. Как уже говорилось, при использовании этого ключа исполняемый код располагается сразу за таблицей символов, создаваемой ассемблером на первом проходе трансляции (которая, в свою очередь, помещается следом за исходным текстом). Это позволяет отвести для исполняемого модуля всю свободную память. Однако после сохранения полученной программы на ленте или диске адрес загрузки ее кодов в заголовке файла окажется равным тому, который вы указали в директиве ORG. Все адреса переходов, естественно, также будут правильными.

Применять последний ключ 32 можно лишь в тех случаях, когда вы абсолютно уверены, что исполняемый код полностью уместится в оперативной памяти, иначе «хвост» программы залезет в начальные адреса ПЗУ, где толку от него не будет никакого. Этот ключ может принести пользу, пожалуй, лишь при создании операционных систем или перемещаемых модулей, но эта тема уже выходит за рамки нашей книги.

Установка размера таблицы меток

Среди сообщений GENS на экране изредка может появиться:

```
No Symbol table space!
```

говорящее о том, что ассемблеру не хватило памяти для размещения таблицы меток. Это может показаться странным, особенно если ваша программа имеет небольшие размеры и свободной памяти на самом деле остается еще килобайт 15-20. А все дело в том, что GENS перед началом трансляции исходного текста выделяет под таблицу не все пространство, а определенных размеров буфер, величина которого рассчитывается исходя из размера текста программы, находящейся в памяти. Чем больше строк вы ввели, тем больше окажется и таблица. В большинстве случаев рассчитанного объема оказывается вполне достаточно, но если при небольших размерах текста он будет насыщен метками и прочими именами, то тогда, скорее всего, и появится указанное выше сообщение.

Бороться с подобной ситуацией поможет второй параметр, задаваемый в команде A. Он соответствует размеру создаваемой таблицы в байтах. Например, если вы считаете, что таблицы в две с половиной тысячи байт будет достаточно, введите для начала ассемблирования команду

```
A,2500
```

При необходимости, конечно, можете указать также и ключи трансляции.

К этой возможности можно прибегать и в тех случаях, когда реальный размер таблицы меток получается намного меньше рассчитанного ассемблером. Таким образом, например, можно высвободить дополнительное пространство для размещения исполняемого кода.

ТРАНСЛЯЦИЯ БОЛЬШИХ ПРОГРАММ

Особенно много сложностей возникает при трансляции больших программ, и в первую очередь это касается тех, кто еще не успел обзавестись дисковой операционной системой TR-DOS и адаптированной к ней версией ассемблера gens4b или GENS4D. Однако это не

означает, что если текст программы, таблица меток, исполняемый файл и сам ассемблер не умещаются в памяти, то мечту о создании серьезных игровых программ можно похоронить. Ассемблер GENS4 предоставляет возможность получения достаточно объемистых исполняемых модулей, сопоставимых с размерами управляющей части фирменных программ. Безусловно, это потребует от вас некоторого терпения, но если есть желание, все остальное приложится.

Конечно, можно транслировать разные модули программы раздельно и определять глобальные адреса, запрашивая вывод таблицы меток (см. ключи ассемблирования). Однако подобный метод вряд ли можно назвать хоть сколько-нибудь удобным и прибегать к нему имеет смысл лишь в исключительных случаях, когда все остальное не помогает. Мы же советуем воспользоваться другой возможностью, имеющейся в ассемблере GENS4.

Идея заключается в том, чтобы не загружать в память исходный текст программы, а транслировать его непосредственно с внешнего носителя. Это позволит высвободить максимальное пространство, так как именно текстовый файл имеет наибольшие размеры по сравнению с таблицей меток и исполняемым файлом. Если ваш компьютер не снабжен дисководом и вы вынуждены работать с лентой, то прежде всего потребуется создать текстовый файл в специальном формате, пригодном для трансляции таким способом. При этом GENS разбивает исходный текст на небольшие фрагменты, а затем считывает и транслирует файл по частям. Считывание происходит в специально предназначенный для этих целей буфер, размер которого необходимо указать в самом начале работы. Введите в редакторе команду C (Change buffers - изменить буферы). На экране появится запрос:

```
Include buffer?
```
на который нужно ввести размер входного буфера для включаемых файлов в байтах. Введите в ответ на него, например, значение 1000. Затем вы увидите еще один запрос - Macro buffer? - который нас пока не интересует, поэтому его можно пропустить, просто нажав **Enter**. После этого загрузите текст программы в память и сохраните его снова, но уже не обычным образом, а с помощью команды T. Ее формат похож на формат команды P для сохранения текста:

```
T[нач.строка],[конечн.строка],[имя_файла]
```
Например, чтобы перевести весь исходный текст во включаемый формат и записать его в файле с именем INCL, нужно ввести команду

```
T1,20000,INCL
```
Теперь очистите память командой Z и напишите одну-единственную строку:

```
10 *F INCL
```
Обратите внимание, что команда ассемблера *F (не путайте с командами редактора) должна находиться в поле меток, а имя файла записывается следом за ней после единичного пробела. Для начала трансляции введите команду редактора A, например:

```
A16,5400
```
Конечно, размер таблицы меток в вашем случае может потребоваться совершенно иной, но важно то, что он обязательно должен быть указан, так как при единственной строке в памяти GENS для этих нужд зарезервирует примерно 100 байт, которых хватит от силы на пару десятков меток. Размер таблицы желательно задать максимально близким к тому, который будет использован, и чтобы узнать его, может понадобиться сначала оттранслировать программу «вхолостую», использовав ключ 2. Для окончательной трансляции желательно применить ключ 16, который отведет максимум памяти для исполняемого модуля.

Таким образом можно обрабатывать исходные тексты программ, превосходящие размеры всей свободной памяти компьютера. Ведь текст может располагаться не в одном, а в нескольких файлах еще до обработки их командой T. Например, программа MOON занимает 3 файла. Создайте для каждого из них три включаемых файла с именами MOON1, MOON2 и MOON3, а затем введите три строки

```
10 *F MOON1
20 *F MOON2
30 *F MOON3
```

которые будут транслироваться уже описанным способом.

При использовании такого метода трансляции нужно помнить еще об одном. Если вы создали включаемые файлы, но не стали их транслировать сразу же, а решили отложить эту работу до следующего раза, то перед началом ассемблирования необходимо будет заново указать размер буфера для включения, причем он должен в точности совпадать с указанным при сохранении программы командой T.

Обладателям дисковой системы TR-DOS повезло несравненно больше. При работе с диском не нужно переводить файлы в специальный формат, достаточно воспользоваться командой *F с указанием имени файла, которое должно начинаться, как и в прочих командах работы с внешней памятью, с указания номера дисковода, например, строка

```
10 *F 1:BATTY
```

оттранслирует непосредственно с диска, находящегося на дисководе A, текстовый файл с именем BATTY. Правда, и в этом случае потребуется задать размер таблицы меток достаточных размеров, но и не слишком большой, чтобы осталось больше места для исполняемого файла. С этой же целью также желательно указать ключ 16.

Еще один существенный плюс использования для трансляции дисковода (помимо скорости считывания) заключается в том, что если исполняемый модуль перекрывает всю свободную память, уже полученный машинный код «сбрасывается» на диск порциями по мере заполнения памяти. Однако в этом случае необходимо указать и третий параметр в команде редактора A - имя выходного файла, то есть того файла, в который будет записана оттранслированная программа. К примеру, это может выглядеть так:

```
A16,8000,2:BATT.EXE
```

После ассемблирования текста исполняемый модуль будет сохранен на дискете в дисководе B под именем BATT.EXE.

МАКРООПРЕДЕЛЕНИЯ

Как вы могли заметить, во многих программах повторяются совершенно однотипные фрагменты текста, отличающиеся только значениями отдельных регистров, а то и вовсе совпадающие. Оказывается, в таких случаях не обязательно каждый раз переписывать одну и ту же последовательность команд. Вы можете обозначить данную последовательность специальными директивами, о которых мы скажем ниже, и присвоить ей какое-нибудь имя. А в дальнейшем, в тех местах текста, где она должна появиться, достаточно записывать

только ее имя. Такие фрагменты текста называются *макроопределениями* или *макросами*, а имя макроса - *макрокомандой*.

Для определения макроса в поле меток записывается его имя, а в поле мнемоник - директива ассемблера MAC. Затем пишется тело макроопределения, состоящее из любых команд, и завершается запись директивой ENDM. В качестве примера можно предложить такой макрос:

```
PRAT    MAC
        LD      A,22
        RST     16
        LD      A,B
        RST     16
        LD      A,C
        RST     16
        ENDM
```

Всякий раз, когда в программу потребуется включить записанную между директивами MAC и ENDM последовательность инструкций, достаточно в поле мнемоник записать макрокоманду PRAT. Это позволит сократить исходный текст программы и сделать его несколько более наглядным, приблизив запись к языкам высокого уровня. Действительно, ведь макрокоманды очень похожи на операторы Бейсика. Например, вместо того чтобы писать

```
        CALL    3435
        LD      A,2
        CALL    5633
```

можно оформить этот фрагмент в виде макроса и присвоить ему имя CLS. Тогда для очистки экрана вы сможете на время забыть адреса соответствующих процедур ПЗУ и записывать в поле мнемоник этот старый знакомый оператор Бейсика.

При задании макросов имеется ряд ограничений, которые необходимо всегда учитывать. Во-первых, имена макроопределений в отличие от имен меток могут состоять не более чем из пяти символов. Причем лишние символы в этом случае не игнорируются, а приводят к появлению ошибки (вообще же лучше обходиться четырьмя символами, тогда вы избавите себя от лишних вопросов). Во-вторых, внутри макроопределения не должно быть строк с метками, так как многократное использование одного и того же макроса приведет к дублированию имен и в результате также появится сообщение об ошибке трансляции. Не допускаются и вложения макросов, то есть внутри макроопределения не могут встречаться ссылки на другие макросы.

Мы уже говорили, что в макросах можно определять не только строго совпадающие фрагменты исходного текста, но и слегка отличающиеся друг от друга. Это становится реальным благодаря возможности использования так называемых формальных параметров. Для каждого макроса допускается задавать до 16 таких параметров. Например, при рисовании точек на экране нужно указывать две координаты. Можно написать макрос, в котором регистры B и C будут загружаться требуемыми значениями и который вызывается командой

```
PLOT    X,Y
```

где X и Y - любые допустимые в одноименном операторе Бейсика значения координат. Формальные параметры в макроопределении задаются знаком равенства (=) и символом, код которого соответствует порядковому номеру фактического параметра в макрокоманде. Для первого параметра этот символ может иметь коды 0, 16, 32, 48 и так далее, второй параметр будут описывать любые символы с кодами 1, 17, 33, 49... Чтобы не запутаться, рекомендуем использовать цифровые символы от **0** до **9** для определения первых десяти параметров, а

остальные 6 можно задавать, например, буквами **K**, **L**, **M**, **N**, **O** и **P**. Тогда макрос PLOT будет записан следующим образом:

```
PLOT   MAC
       LD    C,=0
       LD    B,=1
       CALL  8933
       ENDM
```

После трансляции вышеприведенной макрокоманды PLOT с параметрами 100 для координаты X и 80 для Y получится следующая последовательность команд микропроцессора:

```
       LD    C,100
       LD    B,80
       CALL  8933
```

то есть формальные параметры =0 и =1 заменятся фактическими 100 и 80 соответственно.

При написании макрокоманд нужно помнить, что если имя макроса состоит из пяти символов (напомним еще раз, что это максимальная длина имен макросов), то фактические параметры обязательно нужно заключать в круглые скобки, например:

```
       PRINT (TEXT)
```

Прежде чем привести пример использования макросов в реальной программе, добавим, что в качестве параметров могут выступать только непосредственные числовые значения. Использование символьных строк (за исключением имен меток и констант) не разрешается.

Во время трансляции текст макроопределения не переводится сразу в машинные коды, а помещается в специальный буфер, из которого затем извлекается по мере необходимости. Поэтому перед вводом команды A необходимо указать размер этого буфера с помощью команды C. Помните, при вводе этой команды сначала запрашивается размер входного буфера Include buffer?, а затем появляется еще один запрос - Macro buffer? На него нужно ввести количество байт, достаточное для размещения текста всех макроопределений, заданных в программе. Если задать слишком маленькое число, то во время первого прохода ассемблирования появится сообщение **No Macro Space**. В этом случае нужно повторить ввод с большим числом. В приведенном ниже примере для размещения макросов достаточно 300 байт.

```
       ORG   60000
       ENT   $
; Печать ASCIIZ-строки в позиции экрана, задаваемой первыми двумя параметрами
PRN    MAC
       LD    B,=0
       LD    C,=1
       LD    HL,=2
       CALL  PRNZ
       ENDM
; Позиционирование печати
PRAT   MAC
       LD    A,22
       RST   16
       LD    A,B
       RST   16
       LD    A,C
       RST   16
       ENDM
; Установка цветов INK и PAPER, а также цвета бордюра
COLOR  MAC
```

```
        LD    A,=1*8+=0
        LD    (23693),A
        LD    A,=1
        CALL  8859
        ENDM
; Очистка экрана и назначение вывода на основной экран
CLS     MAC
        CALL  3435
        LD    A,2
        CALL  5633
        ENDM
; Установка PLOT-позиции без рисования точки
PSET    MAC
        LD    L,=0
        LD    H,=1
        LD    (23677),HL
        ENDM
; Черчение линии из текущей PLOT-позиции
DRAW    MAC
        EXX
        PUSH  HL
        LD    DE,=0
        LD    C,=1
        LD    B,=2
        CALL  9402
        POP   HL
        EXX
        ENDM
; Направления рисования линий
UP_RT   EQU   #0101       ;вверх и вправо
DN_RT   EQU   #FF01       ;вниз и вправо
DN_LF   EQU   #FFFF       ;вниз и влево
UP_LF   EQU   #01FF       ;вверх и влево
; ------
BEGIN   COLOR (5,0)
        CLS
        PRN   5,8,TEXT1
        PRN   7,7,TEXT2
        PSET  48,144
        DRAW  UP_RT,131,0
        DRAW  DN_RT,0,39
        DRAW  UP_LF,131,0
        DRAW  UP_RT,0,39
        PSET  50,142
        DRAW  UP_RT,127,0
        DRAW  DN_RT,0,35
        DRAW  UP_LF,127,0
        DRAW  UP_RT,0,35
        RET
; Подпрограмма печати ASCIIZ-строки, вызываемая макросом PRN
PRNZ    PUSH  HL
        PRAT
PRNZ1   LD    A,(HL)
        INC   HL
        AND   A
        JR    Z,PRNZ2
        RST   16
        JR    PRNZ1
PRNZ2   POP   HL
        RET
; ------
TEXT1   DEFB  16,2,19,1
        DEFM  "*** DEMO ***"
        DEFB  0
TEXT2   DEFB  16,6,19,1
```

```
        DEFM  "### MACROS ###"
        DEFB  16,5,0
```

Имея возможность работать с дисководом, очень удобно собрать все макросы в одном или нескольких файлах (например, по родству выполняемых функций) и затем при необходимости включать их в исходный текст с помощью команды ассемблера *F. Так как макросы сразу не транслируются, то это никак не повлияет на размер исполняемого кода, даже если среди включаемых макросов есть такие, которые ни разу не используются в программе. Они, конечно, займут некоторый объем памяти, но так и останутся в буфере невостребованными.

В заключение хочется предостеречь вас от чрезмерного увлечения макроопределениями. Во всем нужно знать меру. Учтите, что макросы могут запросто свести все преимущества ассемблера на нет, снизив эффективность программы, в лучшем случае, до уровня компиляторов.

ДИРЕКТИВЫ УСЛОВНОЙ ТРАНСЛЯЦИИ

Работая с GENS4, у вас есть возможность получать различные варианты исполняемого кода в зависимости от выполнения тех или иных условий. Достигается это включением в программу команд условной трансляции IF, ELSE и END, которые записываются в поле мнемоник (эти слова не относятся к зарезервированным и поэтому их можно использовать в качестве меток, но не макрокоманд). Общий вид текста программы при этом будет таким:

```
        .........
        IF    выражение
        команды_1
        [ELSE
        команды_2]
        END
        .........
```

Команда ELSE и следующий за ней блок инструкций «команды_2» являются необязательной частью условной конструкции, поэтому в данном примере они заключены в квадратные скобки. Если значение выражения после команды IF истинно (то есть не равно нулю), то транслируется блок команд «команды_1» до ELSE или, если его нет, до END. В противном случае (если значение выражения равно нулю) ассемблируются «команды_2» после ELSE, конечно, если эта команда указана. После END трансляция текста протекает как обычно.

Часто эти команды используются для получения различных версий одной и той же программы, одна из которых, например, предназначена для работы на «обычном» Speccy, другая на ZX Spectrum 128 и т. п. Но, на наш взгляд, наиболее полезными они оказываются при написании макроопределений. В этом случае макрос можно составить таким образом, чтобы в зависимости от задаваемых в макрокоманде параметров получался максимально компактный код. Рассмотрим такой пример:

```
CHAN    MAC
        IF    =0
        LD    A,=0          ;если первый параметр не 0
        ELSE
        XOR   A             ;если параметр равен 0
        END
        CALL  5633
        ENDM
```

Встретив в тексте макрокоманду CHAN, ассемблер обратится к одноименному макросу и в первую очередь проверит значение первого параметра =0. Если его величина отлична от 0 (условие истинно), то транслируется команда LD A,N, затем ассемблирование продолжается после команды END. В противном же случае, то есть если заданный параметр равен 0 (условие ложно), то обрабатываются команды после ELSE, в данном случае - XOR A и далее текст транслируется, как и в предыдущем варианте. Поэтому после трансляции макрокоманды

```
        CHAN  2
```

получится последовательность инструкций

```
        LD    A,2
        CALL  5633
```

а если задать

```
        CHAN  0
```

то такая макрокоманда оттранслируется иначе:

```
        XOR   A
        CALL  5633
```

Приведем другой, более серьезный пример применения команд условной трансляции в макросах:

```
        ORG   60000
UP      EQU   1
DN      EQU   %10
RT      EQU   %100
LF      EQU   %1000
SCRL    MAC
        PUSH  BC
        LD    HL,=1*256+=0
        LD    (COL),HL
        LD    HL,=3*256+=2
        LD    (LEN),HL
        IF    =4 & UP ;если 5-й параметр = UP
        CALL  SCR_UP
        END
        IF    =4 & DN ;если 5-й параметр = DN
        CALL  SCR_DN
        END
        IF    =4 & RT ;если 5-й параметр = RT
        CALL  SCR_RT
        END
        IF    =4 & LF ;если 5-й параметр = LF
        CALL  SCR_LF
        END
        POP   BC
        ENDM
; ------
        LD    B,16
SCRL1   SCRL  10,4,5,7,UP
        DJNZ  SCRL1
        LD    B,16
SCRL2   SCRL  10,4,5,7,RT
        DJNZ  SCRL2
        LD    B,16
SCRL3   SCRL  10,4,5,7,DN
        DJNZ  SCRL3
        LD    B,16
SCRL4   SCRL  10,4,5,7,LF
        DJNZ  SCRL4
        RET
```

```
SCR_UP .........
SCR_DN .........
SCR_RT .........
SCR_LF .........
COL     DEFB  0
ROW     DEFB  0
LEN     DEFB  0
HGT     DEFB  0
```

В выражениях ассемблера GENS отсутствует знак равенства, но из этого затруднения можно выйти, если употребить поразрядную операцию «И» - AND, обозначаемую символом «амперсанд» (**&**), а в соответствующем параметре использовать отдельные биты, указывающие на различные действия. В приведенном макросе после определения графических переменных COL, ROW, LEN и HGT в зависимости от последнего параметра вызывается одна из четырех процедур скроллингов (напомним, что сами процедуры были описаны в 6-й главе). Как видите, благодаря командам условной трансляции стало возможно объединить их в одном макросе. В результате и текст программы заметно сократился и стал значительно более удобочитаемым. Правда, при этом несколько возрос размер исполняемого модуля, но этот недостаток также можно устранить, слегка доработав макрос. Например, можно добавить еще один условный блок в самом начале, в котором проверяется значение самого первого параметра и только если он не равен 0, транслируются команды определения переменных, а в противном случае они будут пропускаться.

ПРИЛОЖЕНИЕ I

КОМАНДЫ АССЕМБЛЕРА

Это приложение содержит таблицу кодировки команд микропроцессора Z80. В первой графе указывается мнемоника, во второй - соответствующий ей машинный код, а в последней - время исполнения инструкции, измеряемое в тактах микропроцессора. Для условных и циклических команд (таких как LDIR) даются два значения времени, так как при соблюдении условия или завершении цикла на их выполнение требуется больше тактов. Для некоторых команд приведены два варианта машинных кодов, но время исполнения относится только к более короткому варианту. Следует помнить, что ассемблер при трансляции мнемоник также выбирает более короткий код. Вообще же для многих команд существуют дополнительные варианты кодировки, но они уже относятся к набору так называемых *недокументированных команд*, некоторые из них приведены в следующем приложении. Обозначения: ADDR - двухбайтовый адрес, N - однобайтовое число, NN - двухбайтовое число, PORT - младший байт адреса порта, XX - шестнадцатеричный байт-аргумент, Д - байт смещения (-128...+127).

Команда	Коды	T
ADC A,(HL)	8E	7
ADC A,(IX+Д)	DD8EXX	19
ADC A,(IY+Д)	FD8EXX	19
ADC A,A	8F	4
ADC A,B	88	4
ADC A,C	89	4
ADC A,D	8A	4

Команда	Коды	T
LD L,(IY+Д)	FD6EXX	19
LD L,A	6F	4
LD L,B	68	4
LD L,C	69	4
LD L,D	6A	4
LD L,E	6B	4
LD L,H	6C	4

ADC A,E	8B	4
ADC A,H	8C	4
ADC A,L	8D	4
ADC A,N	CEXX	7
ADC HL,BC	ED4A	15
ADC HL,DE	ED5A	15
ADC HL,HL	ED6A	15
ADC HL,SP	ED7A	15
ADD A,(HL)	86	7
ADD A,(IX+Д)	DD86XX	19
ADD A,(IY+Д)	FD86XX	19
ADD A,A	87	4
ADD A,B	80	4
ADD A,C	81	4
ADD A,D	82	4
ADD A,E	83	4
ADD A,H	84	4
ADD A,L	85	4
ADD A,N	C6XX	7
ADD HL,BC	09	11
ADD HL,DE	19	11
ADD HL,HL	29	11
ADD HL,SP	39	11
ADD IX,BC	DD09	15
ADD IX,DE	DD19	15
ADD IX,IX	DD29	15
ADD IX,SP	DD39	15
ADD IY,BC	FD09	15
ADD IY,DE	FD19	15
ADD IY,IY	FD29	15
ADD IY,SP	FD39	15
AND (HL)	A6	7
AND (IX+Д)	DDA6XX	19
AND (IY+Д)	FDA6XX	19

LD L,L	6D	4
LD L,N	2EXX	7
LD R,A	ED4F	9
LD SP,(ADDR)	ED7BXXXX	20
LD SP,IX	DDF9	10
LD SP,IY	FDF9	10
LD SP,HL	F9	6
LD SP,NN	31XXXX	10
LDD	EDA8	16
LDDR	EDB8	21/16
LDI	EDA0	16
LDIR	EDB0	21/16
NEG	ED44	8
NOP	00	4
OR (HL)	B6	7
OR (IX+Д)	DDB6XX	19
OR (IY+Д)	FDB6XX	19
OR A	B7	4
OR B	B0	4
OR C	B1	4
OR D	B2	4
OR E	B3	4
OR H	B4	4
OR L	B5	4
OR N	F6XX	7
OTDR	EDBB	21/15
OTIR	EDB3	21/15
OUT (C),A	ED79	12
OUT (C),B	ED41	12
OUT (C),C	ED49	12
OUT (C),D	ED51	12
OUT (C),E	ED59	12
OUT (C),H	ED61	12
OUT (C),L	ED69	12

AND A	A7	4
AND B	A0	4
AND C	A1	4
AND D	A2	4
AND E	A3	4
AND H	A4	4
AND L	A5	4
AND N	E6XX	7
BIT 0,(HL)	CB46	12
BIT 0,(IX+Д)	DDCBXX46	20
BIT 0,(IY+Д)	FDCBXX46	20
BIT 0,A	CB47	8
BIT 0,B	CB40	8
BIT 0,C	CB41	8
BIT 0,D	CB42	8
BIT 0,E	CB43	8
BIT 0,H	CB44	8
BIT 0,L	CB45	8
BIT 1,(HL)	CB4E	12
BIT 1,(IX+Д)	DDCBXX4E	20
BIT 1,(IY+Д)	FDCBXX4E	20
BIT 1,A	CB4F	8
BIT 1,B	CB48	8
BIT 1,C	CB49	8
BIT 1,D	CB4A	8
BIT 1,E	CB4B	8
BIT 1,H	CB4C	8
BIT 1,L	CB4D	8
BIT 2,(HL)	CB56	12
BIT 2,(IX+Д)	DDCBXX56	20
BIT 2,(IY+Д)	FDCBXX56	20
BIT 2,A	CB57	8
BIT 2,B	CB50	8
BIT 2,C	CB51	8

OUT (PORT),A	D3XX	12
OUTD	EDAB	16
OUTI	EDA3	16
POP AF	F1	10
POP BC	C1	10
POP DE	D1	10
POP HL	E1	10
POP IX	DDE1	14
POP IY	FDE1	14
PUSH AF	F5	11
PUSH BC	C5	11
PUSH DE	D5	11
PUSH HL	E5	11
PUSH IX	DDE5	15
PUSH IY	FDE5	15
RES 0,(HL)	CB86	15
RES 0,(IX+Д)	DDCBXX86	23
RES 0,(IY+Д)	FDCBXX86	23
RES 0,A	CB87	8
RES 0,B	CB80	8
RES 0,C	CB81	8
RES 0,D	CB82	8
RES 0,E	CB83	8
RES 0,H	CB84	8
RES 0,L	CB85	8
RES 1,(HL)	CB8E	15
RES 1,(IX+Д)	DDCBXX8E	23
RES 1,(IY+Д)	FDCBXX8E	23
RES 1,A	CB8F	8
RES 1,B	CB88	8
RES 1,C	CB89	8
RES 1,D	CB8A	8
RES 1,E	CB8B	8
RES 1,H	CB8C	8

BIT 2,D	CB52	8	RES 1,L	CB8D	8
BIT 2,E	CB53	8	RES 2,(HL)	CB96	15
BIT 2,H	CB54	8	RES 2,(IX+Д)	DDCBXX96	23
BIT 2,L	CB55	8	RES 2,(IY+Д)	FDCBXX96	23
BIT 3,(HL)	CB5E	12	RES 2,A	CB97	8
BIT 3,(IX+Д)	DDCBXX5E	20	RES 2,B	CB90	8
BIT 3,(IY+Д)	FDCBXX5E	8	RES 2,C	CB91	8
BIT 3,A	CB5F	8	RES 2,D	CB92	8
BIT 3,B	CB58	8	RES 2,E	CB93	8
BIT 3,C	CB59	8	RES 2,H	CB94	8
BIT 3,D	CB5A	8	RES 2,L	CB95	8
BIT 3,E	CB5B	8	RES 3,(HL)	CB9E	15
BIT 3,H	CB5C	8	RES 3,(IX+Д)	DDCBXX9E	23
BIT 3,L	CB5D	8	RES 3,(IY+Д)	FDCBXX9E	23
BIT 4,(HL)	CB66	12	RES 3,A	CB9F	8
BIT 4,(IX+Д)	DDCBXX66	20	RES 3,B	CB98	8
BIT 4,(IY+Д)	FDCBXX66	20	RES 3,C	CB99	8
BIT 4,A	CB67	8	RES 3,D	CB9A	8
BIT 4,B	CB60	8	RES 3,E	CB9B	8
BIT 4,C	CB61	8	RES 3,H	CB9C	8
BIT 4,D	CB62	8	RES 3,L	CB9D	8
BIT 4,E	CB63	8	RES 4,(HL)	CBA6	15
BIT 4,H	CB64	8	RES 4,(IX+Д)	DDCBXXA6	23
BIT 4,L	CB65	8	RES 4,(IY+Д)	FDCBXXA6	23
BIT 5,(HL)	CB6E	12	RES 4,A	CBA7	8
BIT 5,(IX+Д)	DDCBXX6E	20	RES 4,B	CBA0	8
BIT 5,(IY+Д)	FDCBXX6E	20	RES 4,C	CBA1	8
BIT 5,A	CB6F	8	RES 4,D	CBA2	8
BIT 5,B	CB68	8	RES 4,E	CBA3	8
BIT 5,C	CB69	8	RES 4,H	CBA4	8
BIT 5,D	CB6A	8	RES 4,L	CBA5	8
BIT 5,E	CB6B	8	RES 5,(HL)	CBAE	15
BIT 5,H	CB6C	8	RES 5,(IX+Д)	DDCBXXAE	23
BIT 5,L	CB6D	8	RES 5,(IY+Д)	FDCBXXAE	23

BIT 6,(HL)	CB76	12
BIT 6,(IX+Д)	DDCBXX76	20
BIT 6,(IY+Д)	FDCBXX76	20
BIT 6,A	CB77	8
BIT 6,B	CB70	8
BIT 6,C	CB71	8
BIT 6,D	CB72	8
BIT 6,E	CB73	8
BIT 6,H	CB74	8
BIT 6,L	CB75	8
BIT 7,(HL)	CB7E	12
BIT 7,(IX+Д)	DDCBXX7E	20
BIT 7,(IY+Д)	FDCBXX7E	20
BIT 7,A	CB7F	8
BIT 7,B	CB78	8
BIT 7,C	CB79	8
BIT 7,D	CB7A	8
BIT 7,E	CB7B	8
BIT 7,H	CB7C	8
BIT 7,L	CB7D	8
CALL ADDR	CDXXXX	17
CALL C,ADDR	DCXXXX	10/17
CALL M,ADDR	FCXXXX	10/17
CALL NC,ADDR	D4XXXX	10/17
CALL NZ,ADDR	C4XXXX	10/17
CALL P,ADDR	F4XXXX	10/17
CALL PE,ADDR	ECXXXX	10/17
CALL PO,ADDR	E4XXXX	10/17
CALL Z,ADDR	CCXXXX	10/17
CCF	3F	4
CP (HL)	BE	7
CP (IX+Д)	DDBEXX	19
CP (IY+Д)	FDBEXX	19
CP A	BF	4

RES 5,A	CBAF	8
RES 5,B	CBA8	8
RES 5,C	CBA9	8
RES 5,D	CBAA	8
RES 5,E	CBAB	8
RES 5,H	CBAC	8
RES 5,L	CBAD	8
RES 6,(HL)	CBB6	15
RES 6,(IX+Д)	DDCBXXB6	23
RES 6,(IY+Д)	FDCBXXB6	23
RES 6,A	CBB7	8
RES 6,B	CBB0	8
RES 6,C	CBB1	8
RES 6,D	CBB2	8
RES 6,E	CBB3	8
RES 6,H	CBB4	8
RES 6,L	CBB5	8
RES 7,(HL)	CBBE	15
RES 7,(IX+Д)	DDCBXXBE	23
RES 7,(IY+Д)	FDCBXXBE	23
RES 7,A	CBBF	8
RES 7,B	CBB8	8
RES 7,C	CBB9	8
RES 7,D	CBBA	8
RES 7,E	CBBB	8
RES 7,H	CBBC	8
RES 7,L	CBBD	8
RET	C9	10
RET C	D8	11/5
RET M	F8	11/5
RET NC	D0	11/5
RET NZ	C0	11/5
RET P	F0	11/5
RET PE	E8	11/5

CP B	B8	4
CP C	B9	4
CP D	BA	4
CP E	BB	4
CP H	BC	4
CP L	BD	4
CP N	FEXX	7
CPD	EDA9	16
CPDR	EDB9	21/16
CPI	EDA1	16
CPIR	EDB1	21/16
CPL	2F	4
DAA	27	4
DEC (HL)	35	11
DEC (IX+Д)	DD35XX	23
DEC (IY+Д)	FD35XX	23
DEC A	3D	4
DEC B	05	4
DEC BC	0B	6
DEC C	0D	4
DEC D	15	4
DEC DE	1B	6
DEC E	1D	4
DEC H	25	4
DEC HL	2B	6
DEC IX	DD2B	10
DEC IY	FD2B	10
DEC L	2D	4
DEC SP	3B	6
DI	F3	4
DJNZ Д	10XX	13/8
EI	FB	4
EX (SP),HL	E3	19
EX (SP),IX	DDE3	23

RET PO	E0	11/5
RET Z	C8	11/5
RETI	ED4D	14
RETN	ED45	14
RL (HL)	CB16	15
RL (IX+Д)	DDCBXX16	23
RL (IY+Д)	FDCBXX16	23
RL A	CB17	8
RL B	CB10	8
RL C	CB11	8
RL D	CB12	8
RL E	CB13	8
RL H	CB14	8
RL L	CB15	8
RLA	17	4
RLC (HL)	CB06	15
RLC (IX+Д)	DDCBXX06	23
RLC (IY+Д)	FDCBXX06	23
RLC A	CB07	8
RLC B	CB00	8
RLC C	CB01	8
RLC D	CB02	8
RLC E	CB03	8
RLC H	CB04	8
RLC L	CB05	8
RLCA	07	4
RLD	ED6F	18
RR (HL)	CB1E	15
RR (IX+Д)	DDCBXX1E	23
RR (IY+Д)	FDCBXX1E	23
RR A	CB1F	8
RR B	CB18	8
RR C	CB19	8
RR D	CB1A	8

EX (SP),IY	FDE3	23
EX AF,AF'	08	4
EX DE,HL	EB	4
EXX	D9	4
HALT	76	4
IM 0	ED46	8
IM 1	ED56	8
IM 2	ED5E	8
IN (HL),(C)	ED70	12
IN A,(C)	ED78	12
IN A,(PORT)	DBXX	11
IN B,(C)	ED40	12
IN C,(C)	ED48	12
IN D,(C)	ED50	12
IN E,(C)	ED58	12
IN H,(C)	ED60	12
IN L,(C)	ED68	12
INC (HL)	34	11
INC (IX+Д)	DD34XX	23
INC (IY+Д)	FD34XX	23
INC A	3C	4
INC B	04	4
INC BC	03	6
INC C	0C	4
INC D	14	4
INC DE	13	6
INC E	1C	4
INC H	24	4
INC HL	23	6
INC IX	DD23	10
INC IY	FD23	10
INC L	2C	4
INC SP	33	6
IND	EDAA	16

RR E	CB1B	8
RR H	CB1C	8
RR L	CB1D	8
RRA	1F	4
RRC (HL)	CB0E	15
RRC (IX+Д)	DDCBXX0E	23
RRC (IY+Д)	FDCBXX0E	23
RRC A	CB0F	8
RRC B	CB08	8
RRC C	CB09	8
RRC D	CB0A	8
RRC E	CB0B	8
RRC H	CB0C	8
RRC L	CB0D	8
RRCA	0F	4
RRD	ED67	18
RST 00	C7	11
RST 08	CF	11
RST 10	D7	11
RST 18	DF	11
RST 20	E7	11
RST 28	EF	11
RST 30	F7	11
RST 38	FF	11
SBC A,(HL)	9E	7
SBC A,(IX+Д)	DD9EXX	19
SBC A,(IY+Д)	FD9EXX	19
SBC A,A	9F	4
SBC A,B	98	4
SBC A,C	99	4
SBC A,D	9A	4
SBC A,E	9B	4
SBC A,H	9C	4
SBC A,L	9D	4

INDR	EDBA	21/16
INI	EDA2	16
INIR	EDB2	21/16
JP (HL)	E9	4
JP (IX)	DDE9	8
JP (IY)	FDE9	8
JP ADDR	C3XXXX	10
JP C,ADDR	DAXXXX	10
JP M,ADDR	FAXXXX	10
JP NC,ADDR	D2XXXX	10
JP NZ,ADDR	C2XXXX	10
JP P,ADDR	F2XXXX	10
JP PE,ADDR	EAXXXX	10
JP PO,ADDR	E2XXXX	10
JP Z,ADDR	CAXXXX	10
JR Д	18XX	12
JR C,Д	38XX	12/7
JR NC,Д	30XX	12/7
JR NZ,Д	20XX	12/7
JR Z,Д	28XX	12/7
LD (ADDR),A	32XXXX	13
LD (ADDR),BC	ED43XXXX	20
LD (ADDR),DE	ED53XXXX	20
LD (ADDR),HL	22XXXX ED63XXXX	16
LD (ADDR),IX	DD22XXXX	20
LD (ADDR),IY	FD22XXXX	20
LD (ADDR),SP	ED73XXXX	20
LD (BC),A	02	7
LD (DE),A	12	7
LD (HL),A	77	7
LD (HL),B	70	7
LD (HL),C	71	7
LD (HL),D	72	7

SBC A,N	DEXX	7
SBC HL,BC	ED42	15
SBC HL,DE	ED52	15
SBC HL,HL	ED62	15
SBC HL,SP	ED72	15
SCF	37	4
SET 0,(HL)	CBC6	15
SET 0,(IX+Д)	DDCBXXC6	23
SET 0,(IY+Д)	FDCBXXC6	23
SET 0,A	CBC7	8
SET 0,B	CBC0	8
SET 0,C	CBC1	8
SET 0,D	CBC2	8
SET 0,E	CBC3	8
SET 0,H	CBC4	8
SET 0,L	CBC5	8
SET 1,(HL)	CBCE	15
SET 1,(IX+Д)	DDCBXXCE	23
SET 1,(IY+Д)	FDCBXXCE	23
SET 1,A	CBCF	8
SET 1,B	CBC8	8
SET 1,C	CBC9	8
SET 1,D	CBCA	8
SET 1,E	CBCB	8
SET 1,H	CBCC	8
SET 1,L	CBCD	8
SET 2,(HL)	CBD6	15
SET 2,(IX+Д)	DDCBXXD6	23
SET 2,(IY+Д)	FDCBXXD6	23
SET 2,A	CBD7	8
SET 2,B	CBD0	8
SET 2,C	CBD1	8
SET 2,D	CBD2	8
SET 2,E	CBD3	8

LD (HL),E	73	7
LD (HL),H	74	7
LD (HL),L	75	7
LD (HL),N	36XX	10
LD (IX+Д),A	DD77XX	19
LD (IX+Д),B	DD70XX	19
LD (IX+Д),C	DD71XX	19
LD (IX+Д),D	DD72XX	19
LD (IX+Д),E	DD73XX	19
LD (IX+Д),H	DD74XX	19
LD (IX+Д),L	DD75XX	19
LD (IX+Д),N	DD36XXXX	19
LD (IY+Д),A	FD77XX	19
LD (IY+Д),B	FD70XX	19
LD (IY+Д),C	FD71XX	19
LD (IY+Д),D	FD72XX	19
LD (IY+Д),E	FD73XX	19
LD (IY+Д),H	FD74XX	19
LD (IY+Д),L	FD75XX	19
LD (IY+Д),N	FD36XXXX	19
LD A,(ADDR)	3AXXXX	13
LD A,(BC)	0A	7
LD A,(DE)	1A	7
LD A,(HL)	7E	7
LD A,(IX+Д)	DD7EXX	19
LD A,(IY+Д)	FD7EXX	19
LD A,A	7F	4
LD A,B	78	4
LD A,C	79	4
LD A,D	7A	4
LD A,E	7B	4
LD A,H	7C	4
LD A,I	ED57	9
LD A,L	7D	4

SET 2,H	CBD4	8
SET 2,L	CBD5	8
SET 3,(HL)	CBDE	15
SET 3,(IX+Д)	DDCBXXDE	23
SET 3,(IY+Д)	FDCBXXDE	23
SET 3,A	CBDF	8
SET 3,B	CBD8	8
SET 3,C	CBD9	8
SET 3,D	CBDA	8
SET 3,E	CBDB	8
SET 3,H	CBDC	8
SET 3,L	CBDD	8
SET 4,(HL)	CBE6	15
SET 4,(IX+Д)	DDCBXXE6	23
SET 4,(IY+Д)	FDCBXXE6	23
SET 4,A	CBE7	8
SET 4,B	CBE0	8
SET 4,C	CBE1	8
SET 4,D	CBE2	8
SET 4,E	CBE3	8
SET 4,H	CBE4	8
SET 4,L	CBE5	8
SET 5,(HL)	CBEE	15
SET 5,(IX+Д)	DDCBXXEE	23
SET 5,(IY+Д)	FDCBXXEE	23
SET 5,A	CBEF	8
SET 5,B	CBE8	8
SET 5,C	CBE9	8
SET 5,D	CBEA	8
SET 5,E	CBEB	8
SET 5,H	CBEC	8
SET 5,L	CBED	8
SET 6,(HL)	CBF6	15
SET 6,(IX+Д)	DDCBXXF6	23

LD A,R	ED5F	9
LD A,N	3EXX	7
LD B,(HL)	46	7
LD B,(IX+Д)	DD46XX	19
LD B,(IY+Д)	FD46XX	19
LD B,A	47	4
LD B,B	40	4
LD B,C	41	4
LD B,D	42	4
LD B,E	43	4
LD B,H	44	4
LD B,L	45	4
LD B,N	06XX	7
LD BC,(ADDR)	ED4BXXXX	20
LD BC,NN	01XXXX	10
LD C,(HL)	4E	7
LD C,(IX+Д)	DD4EXX	19
LD C,(IY+Д)	FD4EXX	19
LD C,A	4F	4
LD C,B	48	4
LD C,C	49	4
LD C,D	4A	4
LD C,E	4B	4
LD C,H	4C	4
LD C,L	4D	4
LD C,N	0EXX	7
LD D,(HL)	56	7
LD D,(IX+Д)	DD56XX	19
LD D,(IY+Д)	FD56XX	19
LD D,A	57	4
LD D,B	50	4
LD D,C	51	4
LD D,D	52	4
LD D,E	53	4

SET 6,(IY+Д)	FDCBXXF6	23
SET 6,A	CBF7	8
SET 6,B	CBF0	8
SET 6,C	CBF1	8
SET 6,D	CBF2	8
SET 6,E	CBF3	8
SET 6,H	CBF4	8
SET 6,L	CBF5	8
SET 7,(HL)	CBFE	15
SET 7,(IX+Д)	DDCBXXFE	23
SET 7,(IY+Д)	FDCBXXFE	23
SET 7,A	CBFF	8
SET 7,B	CBF8	8
SET 7,C	CBF9	8
SET 7,D	CBFA	8
SET 7,E	CBFB	8
SET 7,H	CBFC	8
SET 7,L	CBFD	8
SLA (HL)	CB26	15
SLA (IX+Д)	DDCBXX26	23
SLA (IY+Д)	FDCBXX26	23
SLA A	CB27	8
SLA B	CB20	8
SLA C	CB21	8
SLA D	CB22	8
SLA E	CB23	8
SLA H	CB24	8
SLA L	CB25	8
SRA (HL)	CB2E	15
SRA (IX+Д)	DDCBXX2E	23
SRA (IY+Д)	FDCBXX2E	23
SRA A	CB2F	8
SRA B	CB28	8
SRA C	CB29	8

LD D,H	54	4
LD D,L	55	4
LD D,N	16XX	7
LD DE,(ADDR)	ED5BXXXX	20
LD DE,NN	11XXXX	10
LD E,(HL)	5E	7
LD E,(IX+Д)	DD5EXX	19
LD E,(IY+Д)	FD5EXX	19
LD E,A	5F	4
LD E,B	58	4
LD E,C	59	4
LD E,D	5A	4
LD E,E	5B	4
LD E,H	5C	4
LD E,L	5D	4
LD E,N	1EXX	7
LD H,(HL)	66	7
LD H,(IX+Д)	DD66XX	19
LD H,(IY+Д)	FD66XX	19
LD H,A	67	4
LD H,B	60	4
LD H,C	61	4
LD H,D	62	4
LD H,E	63	4
LD H,H	64	4
LD H,L	65	4
LD H,N	26XX	7
LD HL,(ADDR)	2AXXXX ED6BXXXX	16
LD HL,NN	21XXXX	10
LD I,A	ED47	9
LD IX,(ADDR)	DD2AXXXX	20
LD IX,NN	DD21XXXX	14
LD IY,(ADDR)	FD2AXXXX	20

SRA D	CB2A	8
SRA E	CB2B	8
SRA H	CB2C	8
SRA L	CB2D	8
SRL (HL)	CB3E	15
SRL (IX+Д)	DDCBXX3E	23
SRL (IY+Д)	FDCBXX3E	23
SRL A	CB3F	8
SRL B	CB38	8
SRL C	CB39	8
SRL D	CB3A	8
SRL E	CB3B	8
SRL H	CB3C	8
SRL L	CB3D	8
SUB (HL)	96	7
SUB (IX+Д)	DD96XX	19
SUB (IY+Д)	FD96XX	19
SUB A	97	4
SUB B	90	4
SUB C	91	4
SUB D	92	4
SUB E	93	4
SUB H	94	4
SUB L	95	4
SUB N	D6XX	7
XOR (HL)	AE	7
XOR (IX+Д)	DDAEXX	19
XOR (IY+Д)	FDAEXX	19
XOR A	AF	4
XOR B	A8	4
XOR C	A9	4
XOR D	AA	4
XOR E	AB	4
XOR H	AC	4

LD IY,NN	FD21XXXX	14
LD L,(HL)	6E	7
LD L,(IX+Д)	DD6EXX	19

XOR L	AD	4
XOR N	EEXX	7

ПРИЛОЖЕНИЕ II

НЕКОТОРЫЕ НЕДОКУМЕНТИРОВАННЫЕ КОМАНДЫ

К недокументированным относятся те команды, которые не были описаны фирмой-разработчиком микропроцессора Z80A CPU. Вполне возможно, они даже не были запланированы, а получились, если так можно выразиться, как издержки производства. В связи с этим каждый программист «открывает» их для себя, пользуясь методом «научного тыка». Вы также можете поэкспериментировать, используя некоторые правила построения системы команд, о которой мы и хотим здесь рассказать. Внимательно изучив коды, приведенные в предыдущем приложении, вы сможете заметить определенную закономерность. Сравните, например, команды LD HL,NN, LD IX,NN и LD IY,NN. Не правда ли, кодировки очень похожи друг на друга? Команды, использующие индексные регистры, состоят из тех же кодов, что и LD HL,NN, но предваряются специальным префиксом #DD для IX или #FD для IY. Среди «стандартных» мнемоник отсутствуют команды обработки половинок индексных регистров, но воспользовавшись правилами кодировки, не трудно получить такие команды. Они будут соответствовать кодам обработки регистров H и L, перед которыми стоит один из указанных выше префиксов. Например, для загрузки младшей половинки регистра IX числом 32 можно написать следующую последовательность:

```
DEFB   #DD
LD     L,32
```

В мнемоническом обозначении такая команда обычно записывается как LD IXL,32.
С префиксами #DD и #FD аналогичным образом могут быть получены следующие команды:

```
ADD    A,s
ADC    A,s
AND    s
CP     s
DEC    s
INC    s
LD     r,s
LD     s,n
LD     s,r
OR     s
SBC    A,s
SUB    s
XOR    s
```

где **s** - IXH, IXL, IYH или IYL; **r** - A, B, C, D или E; **n** - однобайтовое числовое значение.
Существует еще целый ряд интересных команд, получаемых с префиксами #DD и #FD, которые могут выполнять сразу два действия - устанавливать или сбрасывать биты одновременно в регистре и в памяти, адресуемой индексными регистрами. Например, команда

```
SET    1,A(IX+3)
```

сначала установит бит 1 в ячейке, адресуемой IX со смещением в 3 байта, а затем поместит полученное значение в регистр A. То есть выполняются как бы две команды:

```
SET   1,(IX+3)
LD    A,(IX+3)
```

Приведем способы кодировки некоторых команд этой группы, а остальные предлагаем вам составить самостоятельно по аналогии с приведенными. Описанная выше команда получается из стандартной

```
SET   1,A
```

Эта команда кодируется двумя байтами #CB и #CF (см. Приложение I). Для получения новой команды необходимо вначале поставить код префикса, а между байтами исходной команды вставить байт величины смещения. Таким образом, приведенная выше мнемоника должна кодироваться последовательностью #DD, #CB, #03, #CF. Команда

```
RES   5,H(IY-5)
```

получается, исходя из кодировки команды

```
RES   5,H
```

В результате у вас должна получиться последовательность кодов #FD, #CB, #FB, #AC. Конечно, все предложенные мнемоники не поддерживаются ассемблером GENS, поэтому использовать их можно одним лишь способом - записывая коды непосредственно в директиве DEFB. То есть последняя команда в ассемблерном тексте будет выглядеть так:

```
DEFB  #FD,#CB,#FB,#AC
```

Несколько сложнее дела обстоят с другими префиксами: #CB и #ED. С ними также можно получить ряд новых команд, хотя часто они лишь повторяют «стандартные» инструкции (особенно это относится к префиксу #ED) и практической пользы поэтому от них мало. Такие команды чаще используются для защиты коммерческих программ, чтобы текст невозможно было прочитать (ни один из известных дизассемблеров недокументированные команды не распознает). Для поисков лучше всего использовать отладчик MONS, так как он в режиме трассировки тупо выполняет машинные инструкции, совершенно не вникая в их смысл. Но в данном случае вам это как раз и нужно. Любой код, который вы ему подсунете, будет выполнен абсолютно так же, как и в программе (заметим, что другой трассировщик - Mon2 - перед выполнением очередной команды справляется о ней в таблице, поэтому, если вы решите использовать для экспериментов именно этот дизассемблер, стоит оформлять искомые коды в виде подпрограммы и трассировать их, нажимая клавишу **S**). Поскольку «пропущенных» команд, получаемых с использованием префикса #CB немного, приведем их полный список (кстати, дизассемблер Mon2, в отличие от MONS, прекрасно «справляется» с этой группой команд).

```
SLL   (HL)        ; CB36
SLL   (IX+Д)      ; DDCBXX36
SLL   (IY+Д)      ; FDCBXX36
SLL   A           ; CB37
SLL   B           ; CB30
SLL   C           ; CB31
SLL   D           ; CB32
SLL   E           ; CB33
SLL   H           ; CB34
SLL   L           ; CB35
```

Эти команды дополняют перечень команд сдвига. При их выполнении содержимое регистра или ячейки памяти сдвигается влево на один разряд. Старший бит переходит во флаг CY, а младший в любом случае заполняется единицей. Например, после сдвига числа 00101110 командой SLL (в такой мнемонике данные команды показываются дизассемблером Mon2, однако в литературе часто предлагается другое обозначение - SLI; тем не менее, это те же самые команды) получится значение 01011101. Команда воздействует на флаги CY, Z, P/V

и S. Флаги H и N сбрасываются в 0. Если вы решите использовать в своей программе приведенные выше инструкции, то помните, что ассемблер GENS понятия не имеет о существовании мнемоники SLL. Поэтому все команды придется кодировать исключительно с помощью директивы DEFB.

ЛИТЕРАТУРА

1. А. Капульцевич, И. Капульцевич, А. Евдокимов. Как написать игру для ZX Spectrum - книга первая, СПб.: Питер, 1994.
2. Ларченко А. А., Родионов Н. Ю. ZX Spectrum и TR-DOS для пользователей и программистов - 3-е изд., СПб.: Питер, 1994.
3. Системные программы для ZX Spectrum. Сборник описаний, 1-й выпуск (под редакцией Родионова Н. Ю.) - СПб.: Питер, 1993.
4. Диалекты Бейсика для ZX Spectrum (под редакцией Родионова Н. Ю., Ларченко А. А.) - СПб.: Питер, 1992.
5. Компьютерные миры ZX Spectrum. Сборник описаний игровых программ, 1-й выпуск - СПб.: Питер, 1994.
