6

O sucesso do Spectrum tem sido espantoso. Este livro é uma introdução prática às suas características mais avançadas, respeitantes tanto ao hardware como ao software. Destina-se ao utilizador do Spectrum que procura uma compreensão mais profunda do aparelho e suas capacidades, começando por analisar o interior do microcomputador, ao que se seguem um guia do BASIC do ZX e uma introdução ao sistema de funcionamento da máquina.

O vídeo do ZX é estudado em detalhe e dedicam-se vários capítulos ao sistema de gravação, à interface RS 232, ao microdrive e às técnicas de programação mais adiantadas. Fornecem-se ao longo do livro projectos e listagens de programas exaustivos, permitindo aos leitores a exploração das mais sofisticadas possibilidades do ZX Spectrum.

Todos os programas deste livro foram verificados e testados pelo Gabinete Verbo de Informática.

BIBLIOTECA VERBO DE INFORMÁTICA

Verbo GUIA AVANCADO PARA O SPECTRUM M.JAMES

GUIA AVANÇADO PARA O SPECTRUM

MIKE JAMES

VERBO

BIBLIOTECA DE INFORMÁTICA

Guia avançado para o Spectrum

MIKE JAMES

Guia avançado para o Spectrum

Verbo

Título original: *An Expert Guide to the Spectrum*
Tradução de Jaime de Oliveira
Revisão científica de Paulo Martel
Capa de Luís Anglin

Editorial VERBO. Lisboa/São Paulo
N.º Ed. 1619
Composto por Fotocompográfica
Impresso por Empresa Litográfica do Sul
em Junho de 1985
Depósito Legal n.º 9034/85

ÍNDICE

Prefácio

O *Spectrum* da Sinclair é um microcomputador extraordinariamente bem sucedido, e merece-o. É sempre surpreendente descobrir o que se pode realizar com um pequeno esforço de programação. Podemos considerá-lo uma máquina revolucionária porque mostra novas formas de conseguir certos efeitos. Por exemplo, o BASIC *ZX* é um novo e excelente dialecto do BASIC, e o *écran* usa atributos paralelos para controlar a cor. O advento da *Interface 1* e das *microdrives* tem resultado em versatilidade e poder ainda maiores.

Muitos dos utilizadores de microcomputadores devem perguntar a si próprios porque tem o *Spectrum* tal sucesso. Este livro explica-o, apontando e explorando várias razões, e permite que se tirem das notáveis características do *Spectrum* o melhor proveito. Em consequência, a obra estuda o *hardware* do *Spectrum*, a sua programação e a interacção vital entre eles.

Depois de um capítulo de introdução, que trata de aspectos gerais da tecnologia dos computadores, os três capítulos seguintes examinam o *Spectrum* normal de 16 k ou 48 k, explorando ambos em termos dos circuitos integrados que o compoēm e também do BASIC *ZX*. O capítulo 5 descreve o complexo sistema I/O escondido dentro do *Spectrum* normal, sistema baseado em canais e *streams*. O *écran* é, obviamente, uma parte importante de quaisquer aplicações, e por isso se lhe devotaram dois capítulos para descrever como é e como funciona, dando exemplos da sua utilização correcta.

Os periféricos *standard* do *Spectrum* — o sistema de gravação, emissor de som e impressora — formam o assunto do capítulo 8, enquanto os capítulos 9 e 10 tratam da *Interface 1* e das *microdrives*.

O capítulo 11 introduz a *Interface RS232* e a rede *Sinclair*, mostrando o potencial de comunicações do *Spectrum*. O capítulo final é uma colecção de exemplos de aplicações para mostrar o tipo

de projectos avançados que o leitor pode abordar por si mesmo. Este livro assume que haja, da parte do leitor, conhecimento da linguagem BASIC a nível de introdução, partindo daí para níveis mais avançados; embora esteja fora da sua finalidade ensinar linguagem assembly, inclui muitos exemplos de aplicações onde esta linguagem é de grande vantagem, e nesses casos são apresentadas rotinas em linguagem máquina e incorporadas em programas BASIC. Para quem já conhece programação em assembly e procura aplicá-la, estes exemplos oferecerão muitas ideias. De qualquer modo, quem ainda não a conhecer pode usar estas rotinas sem outros requisitos — embora elas talvez persuadam das vantagens que a linguagem assembly oferece ou apresentem uma introdução estimulante.

Este livro contém muita matéria, provavelmente mais do que se poderá reter de uma vez só e que por vezes não se conseguirá compreender à primeira leitura. Como muitos assuntos técnicos, a computação não se aprende simplesmente através da leitura. Será necessário cada qual experimentar e tentar resolver problemas por si próprio — antes de realmente chegar a enfrentá-los com sucesso. Não se deve ter temor de explorar ideias próprias — este livro procura dar algumas sugestões, que são apenas uma ponta do icebergue.

Muitas das ideias expostas são interdependentes — verificar-se-á que, à medida que se introduzem novas ideias nos capítulos finais, se ganha um conhecimento mais profundo da matéria apresentada anteriormente. Por causa disto, não é de esperar que se leia a obra de fio a pavio e se assimile toda e cada palavra. Em vez de tal facto, será necessário voltar atrás para reler partes de capítulos anteriores à medida que eles começarem a fazer mais sentido, à luz de novas informações. O autor espera que se considere a obra como de interesse perene, no sentido de que contém matéria suficientemente útil para manter o leitor interessado em muitas áreas e por longo tempo. Acima de tudo, o autor espera ter indicado porque é o *Spectrum* um micro tão sugestivo e que se consiga tirar o máximo proveito do seu enorme potencial.

Mike James

CAPÍTULO 1

Torne-se um especialista

Para se ser especialista em qualquer computador é necessário conhecer razoàvelmente o equipamento propriamente dito *(hardware)* e também os programas que o fazem funcionar *(software)*. De facto não existe uma separação nítida entre equipamento e programas: ninguém se poderá especializar num sem que haja interferência do outro! Quando se usa um computador pessoal, como o *Spectrum*, a maior parte das coisas verdadeiramente interessantes sucedem quando os programas tomam em conta as características inovadoras do equipamento. Afortunadamente isto não significa que cada programador tenha de converter-se num engenheiro electrotécnico. A electrónica é uma carreira difícil, e que requer muito tempo de aprendizagem. Mas em programação de computadores o que realmente importa é um conhecimento da forma como o equipamento afecta o que se consegue com os programas. A maior parte deste livro é dedicada à explicação dos componentes físicos do *Spectrum* sob o ponto de vista do programador criativo. Este capítulo apresenta uma visão global do equipamento de um computador. O capítulo 2 descreve a sua aplicação ao *Spectrum* em particular. Uma parte importante da matéria destes dois primeiros capítulos é usada e desenvolvida nos capítulos posteriores, que tratam de tópicos mais específicos.

Este estudo sobre equipamento pode levar a pensar que o lado da programação ficou esquecido; mas os capítulos 3, 4 e 5, que estudam os pormenores mais recônditos do BASIC *ZX*, demonstrarão que os programas são tão importantes como o equipamento. Os capítulos posteriores descrevem a maneira como funcionam alguns dos periféricos normais do *Spectrum* — o sistema de fita *(cassette)*, a impressora *ZX*, as *interfaces* 1 e 2 e a *microdrive* — e explicam a sua utilização em aplicações como armazenamento de programas e tratamento de redes.

Para compreender os assuntos desenvolvidos neste livro basta ter um conhecimento sólido do BASIC *ZX*. Aos principiantes em BASIC o autor recomenda a leitura de um livro de introdução ao BASIC. Embora se use normalmente o BASIC para ilustrar as ideias expostas, nem sempre é possível alcançar a velocidade

necessária à boa apresentação dos programas. Neste caso não há outra alternativa senão usar a linguagem assembler do *Z80*. Embora este livro não trate sequer superficialmente do assembler do *Z80*, a sua não utilização impediria a apresentação de demasiados assuntos com interesse. A solução adoptada foi, onde necessário, apresentar programas em linguagem assembly do *Z80* que podem ser usados como funções USR dentro de programas em BASIC. Explicar-se-á para que servem tais funções USR e também a maneira geral como elas alcançam o objectivo pretendido, mas sem explicar o código que o realiza. Se se compreender a linguagem assembler do *Z80*, serão suficientes as listagens dos programas, complementadas com os respectivos comentários, para se saber como funcionam esses programas. Não se conhecendo essa linguagem, só se alcancará uma noção geral do funcionamento dos programas, que, de qualquer modo, poderão ser usados a partir do BASIC. Noutras palavras, embora se possa seguir os algoritmos utilizados, não se compreenderão necessariamente os pormenores do código.

É proveitoso observar que, embora muitos livros sobre computadores apresentem uma progressão lógica de ideias desde o primeiro capítulo até ao último, isto não significa que se seja obrigado a ler e compreender inteiramente cada capítulo antes de passar ao capítulo seguinte. É costume dizer que a melhor maneira de ler um manual de computadores é lê-lo uma vez para a frente, outra para trás e só depois tentar começar a comprendê-lo! Esta sugestão, que tem um certo bom senso, muitas vezes paga dividendos: há informações ulteriores que melhoram a compreensão do que já se explicou. Vale a pena ter isto em conta durante a leitura deste livro. Se não se compreende muito bem algum assunto, deve-se resistir à tentação de voltar atrás nesse instante e continuar a ler até ao fim dessa secção. É surpreendente verificar quantas vezes os pormenores se ajustam no seu lugar quando se consegue obter uma visão global da situação. Não se espere compreender toda a obra na primeira leitura. Parte da matéria apresentada só será útil quando posta em prática; só então fará sentido para quem a leu. Assim, este *Guia* é também um livro de referência para o futuro.

OS ELEMENTOS DE UM COMPUTADOR

Todos os computadores têm certas características comuns. Particularizando, todos têm uma UCP (unidade central de processamento),

responsável pela execução das instruções do programa. Todos têm um certo tipo de memória para guardar os programas e respectivos dados, e todos têm certos tipos de unidade de entrada/saída (I/O, do inglês, *input/output*) que permitem comunicar com o programa (ver fig. 1.1).

Esta ideia simplificada vai-se complicando pelo facto de haver tipos diferentes de memória e uma escolha grande de possíveis unidades de I/O. A memória pode dividir-se em dois tipos, primária

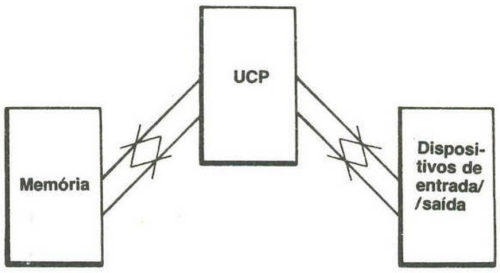


Fig. 1.1. *As diversas partes de um computador*

e secundária ou auxiliar. Usa-se a memória primária para armazenar os programas e dados que a UCP está a processar em cada momento. Usa-se a memória secundária, tal como o armazenamento em fita magnética, para guardar a «biblioteca» de programas do computador e respectivos dados. A memória primária ainda se subdivide em memória de acesso directo (RAM — *random access memory*) e memória só de leitura (ROM — *read only memory*).

A diferença entre RAM e ROM é que a informação armazenada na RAM pode ser alterada por nós, enquanto que o conteúdo da ROM é inalterável, pois é gravado na altura do fabrico. Dito de outra forma, pode-se consultar mas não alterar a informação contida na memória tipo ROM, mas a memória tipo RAM permite, não só ler o seu conteúdo, mas também gravar nova informação para substituir a que existia antes.

Qualquer computador tem uma certa quantidade de RAM, para armazenar programas do utilizador, e uma certa quantidade de ROM com informação inalterável, de que a máquina necessita para executar os nossos programas. No caso do *Spectrum*, e na maioria

dos outros computadores, usa-se a ROM para guardar as regras pelas quais se rege a linguagem BASIC — por outras palavras, é o interpertador de BASIC. Antes de prosseguir no èxame dos componentes físicos do *Spectrum* vale a pena estudar brevemente a forma como a informação é guardada na memória.

ENDEREÇOS E MODELOS DE REPRESENTAÇÃO DA INFORMAÇÃO EM BINÁRIO

Do ponto de vista da UCP, a memória parece-se com uma colecção de locais numerados, cada um capaz de guardar uma certa informação. O número que indentifica cada um dos locais ou posições chama-se o endereço (ver fig. 1.2). A informação, para ser guardada numa posição de memória, toma a forma de uma representação binária. Como um algarismo binário é um «um» ou um «zero», uma representação ou modelo binário é exactamente aquilo que o

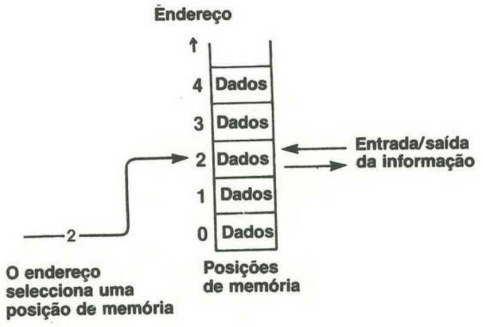


Fig. 1.2. *Utilização de um endereço na busca de uma posição de memória.*

seu nome sugere — uma colecção de «uns» e «zeros». Por exemplo, 01010 é uma representação binária. A maioria dos computadores, incluindo o *Spectrum*, tem memórias que podem armazenar um modelo composto de oito algarismos binários em cada uma das suas posições. Esta medida do modelo binário é tão usada que lhe foi dado um nome (muito conhecido) especial — um *byte*. Pode usar-se um modelo binário para representar as formas mais familiares de dados que se encontram na linguagem BASIC,

mas é importante verificar que em cada posição de memória só pode armazenar-se um modelo, ou representação, binário. Usam-se mais habitualmente estes modelos em «números binários» que representam informação. Por isso convém pormenorizar melhor esta representação. Os oito algarismos binários da posição da memória são geralmente conhecidos por b0 (*bit* zero) até b7 (sétimo bit) como se mostra a seguir:

b7 b6 b5 b4 b3 b2 b1 b0

Cada algarismo deste modelo representante de um número binário está associado com um valor:

b7	b6	b5	b4	b3	b2	b1	b0
124	64	32	16	8	4	2	1

Observando cuidadosamente estes valores verifica-se que, começando pelo 1, associado com b0, todos os outros valores aumentam por um factor de dois para cada posição à esquerda da anterior. Ou, o que é o mesmo, cada valor é igual a 2↑n, onde ↑ significa «elevar à potência de» e n é o número do algarismo binário. Para converter um número binário à notação decimal, mais familiar, basta adicionar todos os valores associados com cada 1 do modelo binário. Por exemplo:

b7	b6	b5	b4	b3	b2	b1	b0
0	0	1	0	1	0	1	0

é 32+8+2, ou 42 em notação decimal.

A maneira mais fácil de converter um número binário em decimal é usar a função BIN do *Spectrum*. Digite-se:

```
PRINT BIN x
```

onde x é o número binário que se deseja converter em decimal. O *Spectrum*, obedientemente, fará a conversão e imprimirá o resultado no *écran*. Recorde-se que se pode sempre usar o computador para evitar cálculos aritméticos difíceis que tantas vezes nos afastam de assuntos simples, como o dos números binários! É mais importante compreender a forma como se usam modelos binários

para representar um número do que ser capaz de realizar milagres de cálculo mental, convertendo números binários em decimais! Infelizmente, o *Spectrum* não tem uma função que converta um número decimal em binário, mas tal conversão não é muito necessária. Para as poucas ocasiões em que o é, a sub-rotina seguinte aceita um número decimal em D e devolve o modelo binário que representa, na variável literal B$.

```
1000 LET B$=""
1010 LET B=D-INT (D/2)*2
1020 IF B=0 THEN LET B$="0"+B$
1030 IF B=1 THEN LET B$="1"+B$
1040 LET D=INT (D/2)
1050 IF D=0 THEN RETURN
1060 GO TO 1010
```

As linguagens de nível elevado, como o BASIC, obrigam-se a artifícios complicados pelo facto de a memória só armazenar modelos binários recebidos de nós. No entanto, os tipos de informação que se encontram em BASIC — números, texto ou vectores — são obtidos a partir deste tipo, mais fundamental, de informação. Uma vez que o leitor conheça algo sobre números e modelos binários ser-lhe-á muito mais fácil compreender como (e porquê) funcionam os computadores. Por exemplo, cada posição de memória só pode guardar oito algarismos binários. Isto significa que o número binário mais pequeno que pode ser armazenado é 00000000, ou zero, e o maior é 11111111 ou, fazendo a conversão para decimal, 255.

Como já se referiu, quando se pretende consultar ou armazenar informação numa posição de memória, usa-se um número chamado «endereço» dessa posição. Isto também tem uma relação muito próxima com modelos e números binários. Do ponto de vista do equipamento, a característica mais importante de um modelo binário é definir cada algarismo só com duas condições ou estados. Habitualmente estas duas condições são representadas por «um» ou por «zero», mas não há nada que impeça de dar-lhes outros nomes, como «aceso» e «apagado», em nada alterando o conceito essencial. Os componentes de um computador usam dois estados de voltagem, baixo e alto, para representar os algarismos que constituem um modelo binário. Por exemplo, os oito dígitos numa posição de memória são armazenados como uma sequência de

estados de voltagens altas e baixas. Da mesma maneira o número usado para seleccionar uma dada posição de memória — o seu endereço — também pode representar-se por um modelo binário de voltagens altas e baixas. Muitos micros, incluindo o *Spectrum*, usam 16 algarismos binários para definir o endereço do local da memória que se está a usar. O número mais pequeno que pode ser representado por um modelo binário de 16 algarismos é 0 e o maior é 65535. Isto determina a quantidade máxima de memória à qual se pode ter acesso, e que dobra de cada vez que se adiciona um algarismo binário *(bit)* ao endereço:

Um endereço de 1 *bit* dá acesso a 2 posições de memória
Um endereço de 2 *bits* dá acesso a 4 posições de memória
Um endereço de 3 *bits* dá acesso a 8 posições de memória

e assim sucessivamente. É portanto mais fácil medir as dimensões da memória de uma forma que tenha este facto em conta. Em vez de usar 1000 posições de memória como unidade básica desta medida, é mais conveniente usar 1024 ou 1 kilobyte. Usando um endereço de 10 *bits* pode ter-se exactamente acesso a 1024 posições de memória ou, simplesmente, 1 k de memória. Assim, usando um endereço de 11 *bits* tem-se acesso a um máximo de 2 k de memória, usando um endereço de 12 *bits* atingem-se 4 k de memória e assim sucessivamente até chegar a um endereço de 16 *bits*, que dará acesso a 64 k de memória. Se a unidade básica de memória fosse 1000 posições, os números associados com cada dimensão do endereço seriam bastante difíceis de memorizar.

MODELOS BINÁRIOS NO EQUIPAMENTO (OU «HARDWARE») – O «BUS»

Números e modelos binários fazem parte, em boa medida, dos programas necessários ao uso de uma máquina. No entanto existe uma correspondência com uma coisa que faz parte do equipamento — o *bus*. Na secção anterior vimos que a representação de um *bit* no equipamento é feita por dois estados diferentes da voltagem — baixo e alto. Será claro então que um grupo de *bits* — ou modelo binário — necessitará de um grupo de voltagens para ser representado em termos de equipamento. Um *bus* é simplesmente um grupo de cabos eléctricos destinados a transportar um modelo

binário, sob a forma de níveis de voltagem, de uma parte do computador para outra. Cada cabo do *bus* leva a condição de um *bit*. Por exemplo a UCP gera endereços que são transportados para a memória através do «*bus* de endereços». Se a UCP usar endereços de 16 *bits*, o *bus* de endereços é formado por 16 cabos eléctricos, e cada cabo leva a condição de um *bit* do endereço. Num sistema de computação real, o *bus* de endereços sai da UCP e está ligado a todos os elementos, memória e unidades de I/O, que necessitam conhecer qual o endereço que a UCP está a usar nesse instante.

Da mesma maneira, a informação é passada através do computador usando um «*bus* de dados» que liga todas as partes do computador que recebem ou transmitem informação. Se a UCP e a memória trabalharem com informação codificada em oito *bits*, então o *bus* de dados consistirá de oito cabos eléctricos. É de notar que o *bus* de dados difere do *bus* de endereços na medida em que pode transportar modelos binários da UCP para outras componentes e dessas componentes para a UCP. Os *bus* de endereços e de dados ligam entre si todas as partes de um computador, constituindo uma única máquina.

Além destes dois *bus* fundamentais do equipamento, existe habitualmente um pequeno grupo de cabos que ligam a UCP ao resto da máquina — o *bus* de controle. Este *bus* transporta um modelo binário que sincroniza o funcionamento de toda a máquina, dando informação acerca do que faz cada elemento da máquina em cada instante. Por exemplo, o *bus* de controle inclui habitualmente um cabo que assinala se a UCP nesse instante está ou não a ler dados exteriores. Do ponto de vista da programação, não há praticamente sinais do *bus* de controle que possam interessar o programador.

Depois desta apresentação generalizada sobre computadores, é tempo de prestar atenção ao que se passa com o *Spectrum* da Sinclair e descobrir o que o torna tão especial.

CAPÍTULO 2

O «Spectrum» por dentro

O *Spectrum* é um computador muito especial. A maior parte daquilo que constitui o computador físico (o *hardware*) está incorporado num único circuito integrado e construído especialmente, chamado ULA (de *Uncommited logic array*). Esta a razão básica do magnífico desempenho do *Spectrum* e do seu baixo preço. No entanto a forma como foi concebido torna difícil alterar o funcionamento da máquina e, neste sentido, o *Spectrum* é uma máquina de «um modo único». Por esta razão, não é muito proveitoso examinar os componentes físicos do *Spectrum* em pormenor, por exemplo com um diagrama completo dos seus circuitos. Um diagrama de circuitos não é muito útil para tentar reparar um *Spectrum*, porque o número de componentes é muito baixo, e um dos maiores — o ULA — só pode ser adquirido ao próprio fabricante, Sinclair! Apesar disto, vale a pena ter uma ideia geral do funcionamento pormenorizado de uma ou duas ligações «externas» importantes, como os circuitos do altifalante e da *cassette*. No entanto, um conhecimento minucioso do equipamento só ajuda se alterar o comportamento dos programas ou se esse conhecimento melhorar ou modificar o próprio funcionamento do *Spectrum*.

A UCP

A UCP do *Spectrum* é o popularíssimo microprocessador *Z80A*. A única diferença entre o circuito integrado *Z80* normal e o *Z80A* é que este trabalha duas vezes mais depressa do que o *Z80*. A velocidade de funcionamento de um microprocessador depende da máxima frequência do relógio que possa aceitar. O relógio é simplesmente um impulso regular que o microprocessador utiliza para sincronizar todas as diferentes operações necessárias para executar uma instrução. O número de impulsos do relógio necessário para cumprir cada instrução depende da complexidade dessa instrução. Em teoria, o *Z80A* pode alcançar os 4 MHz, o que equivale a 1/4 de milionésimo de segundo para a duração de cada impulso do relógio! Na prática, o *Spectrum* usa um relógio de 3,5 MHz, o que não é tão rápido como desejaríamos.

O *Z80* é um microprocessador bastante vulgarizado. Chama-se -lhe «processador de oito *bits*» porque processa a informação de oito em oito *bits*. Tem 16 linhas de endereços, o que lhe permite pesquisar endereços de posições até 64 k, todos eles utilizados no caso do *Spectrum*. Uma característica importante do *Z80* é ter um espaço no endereço que lhe permite endereçar mais outros 64 k que estão dedicados a dispositivos de I/O. Conseguiu-se isto pela adição de mais um *bit*, na realidade mais um *bit* no endereço. Este *bit* adicional, chamado IORQ *(Input Output ReQuest)*, selecciona os 64 k da memória ou os 64 k dos dispositivos de I/O. Parece um poderoso auxílio, e sem dúvida que o é, mas tem uma limitação. Todas as instruções que o *Z80* executa funcionam em qualquer posição da memória, mas os 64 k dos dispositivos de I/O têm o seu próprio conjunto de instruções, muito restrito e especial. As instruções deste conjunto servem essencialmente para ler informação de e escrever informação em quaisquer que sejam os dispositivos de I/O presentes. (Ver IN e OUT mais adiante neste capítulo.) O leitor talvez ache confuso falar de dispositivos de I/O como se fossem posições de memória, mas para o computador isso é, exactamente, o que parecem esses dispositivos. Um dispositivo de I/O envia informação para e recebe informaçao do computador exactamente da mesma maneira que uma posição de memória. A diferença principal é que, dado um dispositivo de I/O qualquer, ele pode corresponder a várias posições de I/O ou «portos». Por exemplo, a impressora *ZX* é um dispositivo de I/O. E usa um porto único no endereço 251 para receber a informação que determina o que irá imprimir, e enviar informação ao *Spectrum* indicando em que «condição» se encontra. As *microdrives* e a *Interface 1* usam três portos de I/O em 254, 247 e 239 para comunicar com o *Spectrum*. O uso de portos e instruções de I/O serão estudados com mais pormenor, com exemplos práticos, em capítulos ulteriores.

A característica mais importante do *Z80*, em relação ao programador, é que determina o código máquina e o assembly que o *Spectrum* usa. O objectivo deste livro não é ensinar o código máquina do *Z80* mas, como se referiu, será usado este código quando não houver alternativa para conseguir a velocidade de processamento necessária a uma demonstração. Se o leitor desejar aprender o código máquina do *Z80* e o assembly, encontra outras obras que abordam essa temática, como *Domínio do Código Máquina no ZX Spectrum*, da colecção a que este livro pertence.

A MEMÓRIA

O espaço reservado para endereços da memória do *Spectrum* está dividido em duas partes, como se pode ver na figura 2.1. Usa-se a ROM de 16 k para guardar o código máquina necessário à implementação das regras do BASIC *ZX*, e as sub-rotinas necessárias ao equipamento normalizado do *Spectrum*. Por exemplo, existe uma sub-rotina que lerá o teclado e outra que produzirá um som usando o altifalante. Se o leitor quiser meter-se numa aventura e tiver o equipamento necessário de programação, pode substituir esta ROM

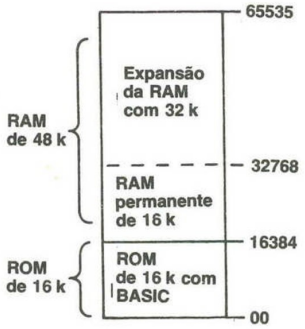


Fig. 2.1. *Estrutura da memória endereçável do* Spectrum

por uma *EPROM 2718* com o seu próprio programa de código máquina. *«EPROM»* é uma palavra formada pelas primeiras letras de *Erasable Programmable Read Only Memory* e é, simplesmente, um tipo de ROM no qual se pode armazenar um programa usando um equipamento razoavelmente barato. Pode rasurar-se completamente uma *PROM* expondo-a por alguns minutos a luz ultravioleta, e por isso pode ser utilizada novamente. Contudo, para ter êxito, o leitor deveria copiar muitas das sub-rotinas que dizem respeito ao equiplamento — tais como o teclado e o *écran* — na sua nova *EPROM*. E é necessário ter em atenção que escrever código máquina que ocupa 16 k não é um projecto que se possa realizar facilmente. A RAM do *Spectrum* está separada em duas secções. Os primeiros 16 k estão sempre presentes, e usam-se para guardar a informação que gera a imagem de *écran* assim como muita

informação própria do sistema e programas dos utentes. Os 32 k finais representam uma opção e adicionam-se aos 16 k básicos do *Spectrum* para chegar ao máximo de memória RAM, isto é, 48 k. Ambas as secções são constituídas por *chips* dinâmicos RAM. O *chip* é um circuito integrado acondicionado em invólucro plástico protector. A secção de 16 k usa 8 *chips 4116* de 16 k *bits* e a de 32 k usa 8 *chips 4532*. Os *4532* são bastante especiais. São fornecidos unicamente pela Texas Instruments, e são circuitos «com defeito». É difícil fabricar circuitos de memória que possam armazenar 64 k de algarismos binários e, para não deitar fora grandes quantidades de circuitos com poucos defeitos, a Texas Instruments projectou o circuito de 64 k de algarismos binários de forma a trabalhar com duas metades independentes de 32 k cada uma. Se os defeitos estiverem todos na mesma metade, não se podem usar como uma pastilha de 64 k algarismos binários, mas não há motivo para não ser usada como uma pastilha de 32 k *bits* — e isto é o que é um 4532. Querendo aumentar o *Spectrum* de 16 k para 48 k, o autor aconselha a comprar o jogo completo de um dos muitos fornecedores; as pastilhas da Texas são um pouco difíceis de conseguir. Os primeiros *Spectrums* (Emissão 1) não se podem aumentar agregando simplesmente as pastilhas que faltam, porque é necessário usar um circuito impresso adicional. Distingue-se um *Spectrum* da Emissão 1 removendo o fundo do invólucro e observando a placa do circuito impresso, à direita das duas tomadas de banana (EAR e MIC). Aí deve estar escrito «ISSUE ONE». A placa correspondente a uma segunda emissão está marcada «ISSUE TWO» na face da frente da placa do circuito impresso, mesmo à dieita da parte central. Todos os números de emissões posteriores estão também nesta posição. A expansão de uma segunda emissão (ISSUE TWO) do *Spectrum* é mais fácil; basta soldar doze *chips* adicionais correctamente e fazer uma ligação de cabo.

É natural a dúvida sobre o que significa a palavra «dinâmico», aplicada a RAMs. Trata-se do seguinte: há duas formas diferentes para implementar uma RAM — estática e dinâmica. Uma RAM estática manterá a informação nela armazenada até que seja alterada ou até que seja cortada a alimentação eléctrica. Neste sentido é simples de usar, de confiança e fácil de testar. O problema é que os produtores não foram capazes de fabricar *chips* RAM estáticas, com capacidades muito grandes. Por outro lado, existem RAMs dinâmicas com dimensões até 64 k dígitos binários por pastilha, a preços

muito aceitáveis. A sua desvantagem é que a informação nelas armazenada tende a desaparecer a não ser que seja lida e tornada a ser escrita de vez em quando. Este acto de leitura e reescrita de informação é conhecido como o «refrescar» de uma memória RAM dinâmica, e na prática há um circuito especial para conseguí-lo sem que nós nos apercebamos. No caso do *Spectrum*, consegue-se este refrescar pelo funcionamento conjunto do *Z80* e do ULA e todo o conjunto dos 48 k é refrescado sem que notemos qualquer perda no seu desempenho ou outro problema de funcionamento.

ACESSO À MEMÓRIA EM BASIC – PEEK E POKE

No dialecto BASIC *ZX* há dois modos de examinar e alterar as posições da memória. O comando.

PEEK (endereço)

devolverá o conteúdo da memória localizada em «endereço». Como o «endereço» deve ser um número binário de 16 algarismos (ver capítulo 1), deve estar entre 0 e 65535. Semelhantemente, como a informação armazenada na memória é um modelo binário de oito algarismos, o valor devolvivo (como um número binário) por PEEK deve, obrigatoriamente, estar entre 0 e 255. O comando

POKE endereço, dados

armazenará o modelo binário correspondente à representação binária de «informação» na memória localizada em «endereço». Uma vez mais, o «endereço» deve estar entre 0 e 65535 e «informação» deve estar entre 0 e 255.

Note-se que embora PEEK e POKE funcionem em termos de números decimais, muitas vezes o que importa realmente é o modelo binário subjacente. Por exemplo, na definição de caracteres novos (ver capítulo 6), cada *pixel* é representado por um *bit* único: 1 se o *pixel* se refere a tinta e 0 se o *pixel* se refere ao fundo. Para alterar uma posição da memória utilizando POKE e escrever nessa posição um modelo binário representando *pixels* tinta/fundo será necessário tratar o modelo binário como um número binário em decimal. Felizmente, o BASIC ZX inclui o comando BIN, que torna desnecessária esta conversão. Se o leitor desejar introduzir um modelo binário numa posição de memória, usando POKE, poderá usar:

POKE endereço, BIN x

onde x é o modelo binário. No entanto, este método não resulta se x for uma variável (BIN não funciona com variáveis) e PEEK devolve sempre um valor decimal. Por esta razão, serão dados métodos, em capítulos posteriores, para usar o BASIC na manipulação de valores decimais como se fossem modelos binários.

O «ÉCRAN» DE VÍDEO

O *écran* de vídeo usa um sistema muito engenhoso de atributos paralelos para obter um *écran* de oito cores (com algumas restrições) num pouco mais de memória do que a necessária para um *écran* a preto e branco. Quase todo o labor posto na geração da imagem é da responsabilidade do *chip* ULA. É um pouco frustrante que o *chip* ULA não seja programável para produzir modos diferentes de imagem. Desde o momento em que se liga o *Spectrum*, o ULA expõe a informação armazenada numa área fixa da memória — a RAM do vídeo — para produzir um *écran* a cores de formato constante (256 × 192 pontos). Este modo único de operação do expositor oferece pouco campo para experimentação. Contudo, 256 por 200 pontos a cores é uma resolução mais do que adequada.

O único aspecto útil dos componentes do sistema de geração da imagem é a forma por que a RAM do vídeo determina aquilo que será exposto no *écran*; este assunto é tratado no capítulo 6. No entanto, é bom ter uma ideia geral do modo como isto funciona, e por tal motivo explicaremos, a seguir, os princípios em que se baseia a geração da imagem de vídeo. A RAM do vídeo está sempre nos primeiros 6912 *bytes* dos 16 k inferiores da RAM. Enquanto está a ser exposta uma imagem de TV, o ULA pesquisa esta área da RAM e a informação que contém usa-se para determinar a cor de cada ponto ou *pixel (picture element)* do *écran*. Uma imagem de TV (pelo menos na Grã-Bretanha e em Portugal), é composta por 625 linhas expostas cada cinquenta avos de segundo.

Para produzir uma imagem estável, o ULA deve não só gerar os sinais de sincronismo que marcam o início de cada linha e de cada imagem mas, também, recuperar informação da RAM do vídeo suficientemente depressa para determinar a cor de cada *pixel* daquela linha. O ULA deve ainda recuperar cada elemento de

informação imediatamente antes de os *pixels*, que são controlados por ele, serem expostos na linha. Por outras palavras, para produzir uma imagem estável, o ULA deve ser capaz de ter acesso à RAM do vídeo a qualquer momento em que necessite. Mas há uma dificuldade: a UCP deve também ter acesso à RAM do vídeo, de vez em quando. Se assim não fosse, como poderia modificar-se a informação que controla o *écran*? Isto significa que a informação que está na RAM do vídeo e o *bus* de endereços devem ser compartilhados pelo ULA, que gera a imagem, e a UCP, que opera com ela (ver figura 2.2). No entanto, só um dos dois pode, em cada instante, usar a memória do vídeo. Se é necessário que ambos a usem, tem de estabelecer-se uma prioridade que decida qual terá que aguardar. Fazer esperar o ULA para que a UCP use a RAM do vídeo, enquanto o ULA está a gerar a imagem do vídeo, resultará em espaços vazios (pontos brancos) na imagem. Acontece isto mesmo em algumas outras máquinas. É melhor que a UCP espere até que o ULA termine a utilização da RAM do vídeo, e é assim que procede o *Spectrum*. No entanto, há um problema oculto neste esquema. O ULA e a UCP são obrigadas a compartilhar os *bus* de dados e de

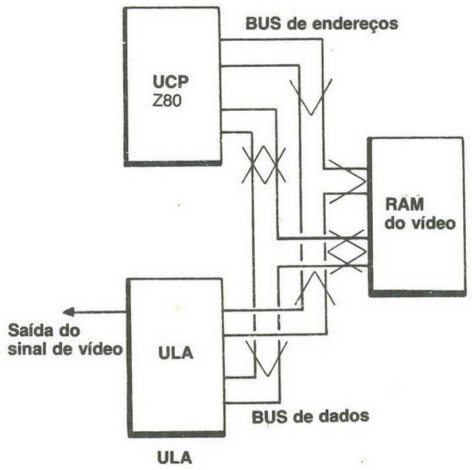


Fig. 2.2. *Ligações compartilhadas entre a RAM do vídeo, o ULA e a UCP.*

endereços para ter acesso à RAM do vídeo. Se a UCP não pode utilizar a RAM do vídeo enquanto o ULA o faz, não poderá, igualmente, usar qualquer outra memória, RAM ou ROM, do sistema, porque os *bus* de dados e de endereços estão também a ser solicitados pelo ULA. Se esta limitação fosse aceite, a máquina resultante trabalharia demasiado devagar. Cada acesso à memória pela UCP teria de esperar até que o ULA já não necessitasse da memória. A solução adoptada pelo *Spectrum* foi proporcionar *bus* de dados e de endereços separados para a UCP e para o ULA. Isto significa que o ULA pode usar os 16 k de RAM que contêm a RAM do vídeo enquanto, simultaneamente, a UCP usa qualquer outra memória não constituída por estes 16 k. Observe-se isto na figura 2.3, onde se verifica que o ULA tem uma ligação directa aos 16 k de RAM, onde está a RAM do vídeo, enquanto a UCP tem uma ligação directa à memória restante. Quando o ULA detecta que a UCP quer utilizar os seus 16 k, pára o relógio da UCP até que possa deixar a UCP ter acesso ao seu *bus* de endereços e de dados. Isto causa uma ligeira demora na operação da UCP quando utiliza os 16 k mais baixos da memória. Não se nota, habitualmente, quando se está a executar uma linguagem lenta como o BASIC, mas obriga programas em código máquina, armazenados nos 16 k mais baixos, a serem executados a ritmos diferentes. Isto só constitui um problema real se é crítica a medição de intervalos de tempo ou se estão incluídos no programa ciclos de medição de intervalos de tempo.

Embora seja de pouco valor prático, interessa notar que o *Spectrum* não usa dispendiosos *chips* múltiplos para controlar o acesso ao *bus* da RAM do vídeo. Em vez dele, utiliza o mais simples de todos os componentes electrónicos — a resistência. Quando a UCP não tenta utilizar os 16 k mais baixos da RAM, os dois *bus* funcionam independentemente, com os sinais num dos *bus* aparecendo a um nível muito mais reduzido do que no outro por causa da acção de «abaixamento» da voltagem das resistências. Esta redução da voltagem é tal que os sinais do outro *bus* aparecem como «ruído de fundo» e não influenciam os acontecimentos. No entanto, quando o ULA deixa a UCP ter acesso à sua parte dos *bus* de endereços e de dados, pára a «condução» dos *bus*. A redução da voltagem produzida pelas resistências é agora muito atenuada; por isso os sinais da UCP ganham o controle. É uma solução notavelmente brilhante, simples e barata para um problema muito comum

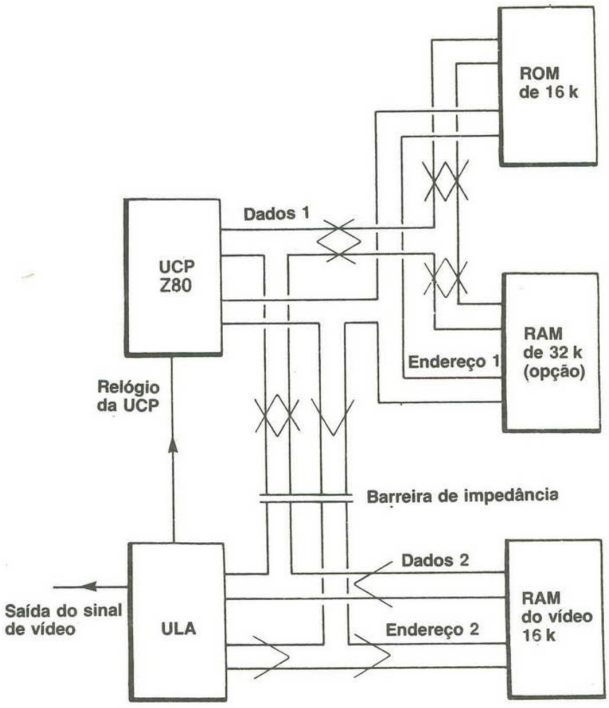


Fig. 2.3. *Ligações entre as áreas da memória principal, a UCP e o ULA, mostrando como se compartilha o acesso à RAM do vídeo.*

em projectos de equipamentos electrónicos digitais e é típico da engenharia inventiva da Sinclair.

O CIRCUITO DE SAÍDA DO VÍDEO

O ULA é responsável pela obtenção dos dados da RAM do vídeo e utiliza-os para transformar a informação de cor em três sinais vídeo. No entanto, a transformação destes três sinais a cor num único sinal vídeo de cor PAL é da responsabilidade de um *chip* de codificação

PAL, *LM1889 N*. Os três sinais produzidos pelo ULA são:

Y = sinais de luminescência e sincronização
U = sinal azul-verde
V = sinal encarnado-amarelo

Esta utilização de sinais de diferença de cor é um problema para quem quiser usar um monitor de vídeo que só tenha uma entrada RGB *(Red, Green, Blue)*, mas simplifica os circuitos de vídeo do *Spectrum*. O codificador PAL gera, a partir dos sinais U e V, um sinal a cor ou cromático que é misturado com o sinal Y por um misturador de dois transistores, resultando, finalmente, no sinal vídeo PAL, que vai alimentar o modulador UHF.

Pode ajustar-se a cor e a qualidade do expositor usando o condensador variável e resistência posicionadas numa linha do lado esquerdo da placa do circuito impresso (ver figura 2.4). (Note-se que as versões mais recentes do *Spectrum* já não possuem estes controles.) O ajuste cuidadoso da TC1 melhora a nitidez da imagem, eliminando quaisquer interferências.

VR1 e VR2 servem para ajustar o equilíbrio relativo da cor do *écran*. VR1 altera o equilíbrio encarnado-amarelo e VR2 modifica o equilíbrio azul-amarelo. Na prática é melhor ajustar o equilíbrio de cor do aparelho de TV ligado ao *Spectrum*, em vez de «brincar» com VR1 e VR2. A frequência dos impulsos do relógio à UCP é

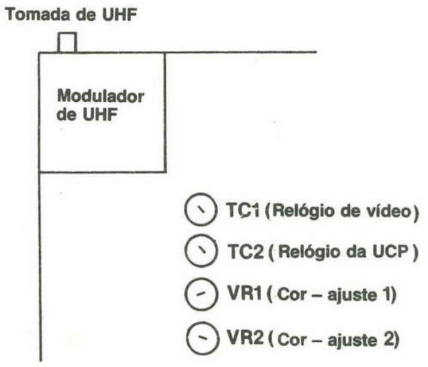


Fig. 2.4. *Ajustes do sinal de saída do vídeo.*

regulada por TC2 e não deve ser modificada. Os sinais U, V, Y e o composto estão todos disponíveis no conector da face traseira, que será descrito mais adiante. Com um pouco de esforço, estes sinais podem alimentar um monitor a cor RGB normalizado, ou um preto e branco ou a cor desde que aceite um sinal de vídeo composto. Este assunto é estudado com mais profundidade depois da secção que trata dos sinais disponíveis no conector traseiro do *Spectrum*.

I/O EM BASIC – IN E OUT

O *Z80* fornece um espaço adicional de endereços com 64 k de capacidade para dispositivos de I/O. Tanto os endereços de I/O como a informação são transportados pelos *bus* de endereços e de dados que ligam a UCP ao resto do computador. Como se descreveu anteriormente, os endereços da memória e de I/O distinguem-se pela condição em que estiver uma linha do *bus* de controle chamada IORQ *(I/O ReQuest)*. O BASIC *ZX* tem dois comandos que permitem o acesso aos dispositivos de I/O, à semelhança do que acontece com PEEK e POKE, que permitem acesso directo à memória. O comando

IN endereço

devolve um valor vindo do dispositivo localizado no «endereço» I/O. A diferença principal entre PEEK/POKE e IN/OUT é que todas as posições da memória se comportam aproximadamente da mesma forma, mas o dispostivo localizado em «endereço» pode comportar-se de muitas maneiras dependentes do seu tipo. Note-se também que, num comando OUT, não está implicado nenhum armazenamento de informação. Por exemplo, se foi contruída uma *interface* especial para ligar ao *Spectrum* uma impressora não normalizada, a *interface* poderá configurar-se para aceitar dados, por exemplo, do endereço 56 de I/O. Para o conseguir, a *interface* teria que ser capaz de detectar, através do exame das linhas de endereço e IORQ, a ocorrência de um modelo binário correspondendo ao endereço 56, de I/O. Uma vez detectada esta ocorrência, leria a informação presente no *bus* de dados e passá-la-ia à impressora, onde seria interpretada como o código de um carácter. Assim, OUT 56, code («A») enviaria o código ASCII da letra A à impressora, mas IN 56 seria ignorado totalmente pela *interface* da impressora e não devolveria nenhuma informação útil. Alguns

endereços de I/O correspondem a portos I/O que só aceitam informação, tais como *interfaces* de impressoras. Neste caso só faz sentido usar OUT. Alguns endereços I/O correspondem a portos I/O que somente fornecem informação, e neste caso só faz sentido usar IN. Contudo, alguns endereços I/O correspondem a portos que podem receber ou dar informação. Uma *interface* para *cassette,* por exemplo, tanto pode ler como escrever informação. Em conclusão, para usar correctamente IN e OUT, além de termos que saber o endereço que ocupa o dispositivo, deveremos também conhecer o seu funcionamento com pormenor.

OS DISPOSITIVOS DE I/O PRÓPRIOS DO «SPECTRUM»

Os dispositivos de I/O próprios do *Spectrum* são o altifalante, a *interface* para a *cassette* e o teclado. Todos são controlados pelo ULA. A *interface* para a *cassette* e o altifalante são controlados por uma ligação única ao ULA, e neste sentido são um único dispositivo de I/O.

Como já se mencionou, o *Z80* tem um espaço separado de endereços com a capacidade de 64 k que podem ser usados na selecção de dispositivos de I/O. Contudo, em vez de atribuir um endereço próprio a cada dispositivo de I/O (ou grupo de endereços) o *Spectrum* atribui a cada dispositivo um algarismo binário *(bit)* particular do endereço. Por exemplo, o primeiro *bit,* b0, selecciona os dispositivos internos de I/O ligados ao ULA. A acção desde *bit* é tal que, quando é zero, o ULA fica seleccionado. Assim, qualquer endereço de I/O que tenha b0 igual a zero seleccionará o ULA. Do mesmo modo, b2 selecciona a impressora *ZX* quando é zero. Se continuarmos a atribuir *bits* do endereço a outros dispositivos veremos que o número máximo de dispositivos que podem ser controlados é 16. Na realidade, o *Spectrum* usa somente os *bits* b0 a b4 do endereço para escolher um de seis possíveis dispositivos, como se apresenta na tabela seguinte:

b0	ULA teclado/altifalante/*interface* para a *cassette*
b1	não é utilizado
b2	impressora *ZX*
b3	*Microdrives* e *Interface 1*
b4	*Microdrives* e *Interface 1*

Assim, para usos especiais somente restam b5, b6 e b7: os *bits* de b8 a b15 são utilizados para escolher qual das colunas de teclas, que constituem o teclado, está a ser lida (ver mais adiante). Está claro que a situação seria muito confusa se fossem seleccionados, ao mesmo tempo, vários dispositivos de I/O, por isso só são válidos os endereços I/O que tiverem um dos *bits* de b0 a b7 igual a zero. Como o ULA é seleccionado por um *bit* único de endereço, parece impossível que possa tomar conta de tantos dispositivos diferentes de I/O. Na realidade, o ULA tem um comportamento diferente consoante está a ser lido (usando IN) ou escrito (usando OUT).

O ULA COMO DISPOSITIVO DE SAÍDA

Quando a informação se envia ao ULA para que funcione como dispositivo de saída, ele passa a controlar o altifalante e a ligação de saída para a *cassette,* MIC. Embora, no sentido estrito da palavra, não tenha nada a ver com I/O, a cor da zona de moldura do *écran* da TV é também controlada pelo ULA, actuando como dispositivo de saída. Cada um destes dispositivos internos de saída é controlado pelo modelo binário enviado ao ULA, de acordo com o esquema seguinte:

```
b7 b6 b5 b4     b3     b2 b1 b0
*  *  *  A/F    MIC    (  c o r )
```

onde * significa que o *bit* correspondente não é usado e A/F é o altifalante. Assim, o altifalante é controlado por b4, o MIC por b3 e a cor da zona de moldura pelo número binário representado pelos *bits* b2, b1 e b0. Para se usar esta informação, basta saber qual o endereço de I/O para enviar dados ao ULA. Como o ULA é seleccionado quando b0 é zero e b1 a b7 são iguais a um, haverá somente que determinar os valores de b8 a b15. Como se mencionou anteriormente, as linhas b8 a b15 do endereço são utilizadas para pesquisar o teclado: por isso, em saída de dados, podem ser postas a zero. E por isso o modelo binário do endereço de saída para o ULA é o seguinte:

```
b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
0   0   0   0   0   0   0  0  1  1  1  1  1  1  1  0
```

ou 254 decimal.

Como exemplo da utilização do ULA como dispositivo de saída, experimente-se o seguinte programa:

```
10 INPUT B
20 OUT 254,B
30 GO TO 10
```

Se dermos a B valores entre 0 e 7, veremos a cor da moldura mudar. Embora também se possa usar a mesma técnica para o altifalante ou MIC, o BASIC é tão lento que o melhor que se conseguirá é um bzzz, de tom grave. No exemplo seguinte:

```
10 OUT 254,16
20 OUT 254,0
30 GO TO 10
```

a linha 10 envia ao ULA o modelo binário 00010000 e a linha 20 envia-lhe 00000000. Reconhece-se neste programa um ciclo cujo resultado é mudar continuamente b4, que controla o altifalante, entre 0 e 1. O resultado é um som grave emitido pelo altifalante. No capítulo 8, descrevemos mais minuciosamente a utilização do porto I/O 254 para controlar o alfitalante. Como este e a saída MIC para o gravador de *cassettes* são controlados pelo mesmo terminal do ULA o sinal emitido pelo altifalante também está presente na saída do MIC. Isto significa que, gravando-se o som, bastante baixo, produzido pelo altifalante, se poderá tocar a *cassette* mais tarde e reproduzir esse som com o volume bastante mais alto. Do mesmo modo, se se ligar um amplificador com um altifalante à saída MIC, aumentam-se os sons emitidos pelo *Spectrum* até qualquer altura desejada. Por outras palavras, MIC não é somente uma «saída para a *cassette*», mas também uma ligação de «saída de som».

O ULA COMO DISPOSITIVO DE ENTRADA

O ULA, ao ser utilizado como dispositivo de entrada, envia à UCP informação respeitante à condição em que se encontram a entrada EAR e o teclado. Como este último é o mais complicado, descrevemo-lo em primeiro lugar.

Na figura 2.5 apresenta-se um diagrama do teclado do *Spectrum*. Observe-se que tem a forma de uma matriz de ligações. Quando se prime uma tecla, estabelece-se uma ligação entre um cabo horizontal e um vertical. É evidente que, para identificar a tecla que foi carregada, bastará saber quais os cabos, horizontal e vertical, que entraram em contacto. Os oito cabos horizontais estão ligados aos *bits* b8 a b15 do *bus* de endereços, de forma a estes serem postos a níveis diferentes de voltagem, de acordo com o modelo binário do endereço que está em uso. Os cinco cabos verticais estão ligados a cinco terminais de entrada do ULA e, quando este é utilizado como dispositivo de entrada, o seu estado é enviado à UCP como *bits* b0 a b4 dos dados. Por outras palavras, a instrução IN 254 «lê o estado» em que se encontram os cinco cabos verticais do teclado, e devolve o equivalente decimal do número binário representado pelos *bits* b0 a b4. Como as linhas verticais de entrada estão ligadas a +5 volts

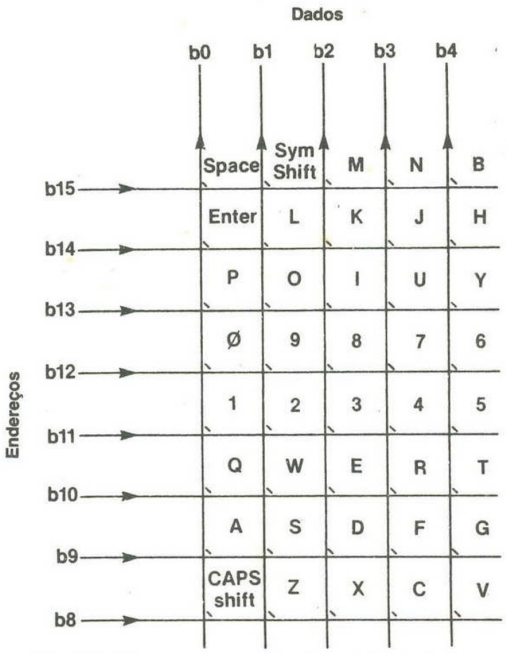


Fig. 2.5. *Diagrama esquemático do teclado do* Spectrum.

(alto), devolvem o valor 1 quando não se carrega em nenhuma tecla. No instante em que IN 254 lê as linhas de entrada, o endereço presente no *bus* (ou seja, 254) é tal que as linhas de b8 a b15 estão todas a 0, isto é, a baixa voltagem. Se se premir uma tecla, esta fará entrar em contacto uma linha vertical e uma das linhas de endereço. A linha de endereço ficará a baixa voltagem e retomará o valor zero. Isto é, IN 254 devolverá uma representação binária com um zero correspondendo a cada linha vertical ligada a uma linha de endereço. Isto permite saber se foi premida ou não uma tecla, mas como saber qual das 8 ligada a uma linha vertical foi a seleccionada? Se as linhas de b8 a b16 estiverem todas em 0, não há modo de saber. A solução é pôr a zero apenas uma das linhas de endereço; assim, haverá apenas uma única fila de teclas, capaz de ligar as linhas de entrada à baixa voltagem. Deste modo, em vez de usar o endereço de I/O 254 para ler o teclado, terá de se pôr a zero *uma só* das linhas de b8 a b15. Por exemplo, para ler a fila de teclas ligadas à linha b15 do endereço, haverá que usar, para o endereço de I/O, o seguinte modelo binário:

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

que é igual a 32766 em decimal. Isto é, a instrução IN 32766 devolverá um valor que, representado em binário, tem os bits b0 a b4 postos de acordo com o estado da primeira fila de teclas — espaço a B — com - representando uma tecla premida.

Seguindo este raciocínio obtêm-se os valores decimais seguintes para os endereços de I/O que lêm cada uma das filas da matriz representativa do teclado:

linha do endereço posta a zero	endereço de I/O	teclas
b15	32766	desde SPACE a B
b14	49150	desde ENTER a H
b13	57324	de P a Y
b12	61438	de 0 a 6
b11	63486	de 1 a 5
b10	64510	de Q a T
b9	65022	de A a G
b8	65278	de CAPS SHIFT a V

Usando esta informação é possível escrever programas que detectem quando se carrega em várias teclas simultaneamente. Por exemplo, o programa seguinte analisará os dois grupos de cinco teclas que formam a linha de cima do teclado; e imprimirá no *écran* o modelo binário resultante:

```
10 LET D=IN 63486
20 GO SUB 1000
30 PRINT AT 5,10;
40 FOR I=8 TO 4 STEP -1
50 PRINT B$(I);
60 NEXT I
70 LET D=IN 61438
80 GO SUB 1000
90 PRINT B$(4 TO 8)
100 GO TO 10
```

Note-se que, para que este programa funcione correctamente, deve incluir-se a sub-rotina 1000, dada no capítulo 1, que converte números decimais em binários. Como a fila superior do teclado inclui as teclas das setas, o programa anterior pode ser usado em jogos e outros programas que necessitem controlar movimentos. No entanto, é de notar ainda que as duas meias filas de teclas «interactuam», de forma que a pressão de mais que uma tecla em cada metade, simultaneamente, poderá falsear os resultados.

Agora que se explicaram os componentes físicos do teclado e os seus princípios de funcionamento, é fácil reconhecer que todas as elaboradíssimas características do teclado se devem aos programas incorporados no *Spectrum*. Existem rotinas em código máquina na ROM do SPECTRUM que inspeccionam o estado do teclado. Tomando em conta qual das teclas se premiu e também qualquer pressão das teclas SHIFT, as rotinas convertem este conhecimento em código que representa uma das cinco possíveis legendas sobre a tecla ou acima e abaixo da mesma. Os referidos programas são ainda responsáveis pela repetição automática de qualquer tecla premida continuamente e por verificar (cada 1/50 de segundo) se a tecla BREAK foi ou não premida.

Quando se usa o ULA como dispositivo de entrada, ele devolve o estado em que se encontra a tomada EAR, no *bit* b6 do modelo binário. Se a informação vinda do gravador de *cassettes* foi um estado de voltagem alta, então b6 é 1. Se não, é zero. O ULA devolverá sempre o estado da tomada EAR em b6, não importando como estejam os *bits* de b8 a b15. Assim, se quisermos ler o teclado

e a tomada EAR, utilizemos um dos endereços dados acima. Se quisermos ler a tomada EAR independentemente do teclado, todos os *bits* b8 a b15 deverão ser postos a 1; por isso o endereço de I/O a usar será 65534. Habitualmente o estado de b6 usa-se para descodificar os tons do som produzido pelo gravador de *cassettes*. No entanto, pode usar-se o estado de b6 para outras tarefas simples de entrada de dados. Por exemplo, o programa seguinte detecta onde começa a gravação na fita:

```
10 PRINT "LIGUE O GRAVADOR"
20 IF IN 65534=255 THEN PRINT
AT 2,0;"Silencio": GO TO 20
30 PRINT "Comecou o som"
```

Se pusermos uma *cassette* a funcionar, este programa imprimirá «silêncio» até detectar o primeiro ruído na fita. Note-se, todavia, que a entrada EAR não se pode usar para verificar voltagens que variam devagar, porque EAR está ligada ao ULA através de um condensador de baixa capacidade.

O CONECTOR DE EXPANSÃO

A maioria dos sinais usados no interior do *Spectrum* está disponível no conector na face de trás do *Spectrum*. O conector é habitualmente usado para ligar a impressora *ZX*, *microdrives* e *Interfaces 1 e 2*. Contudo, também aceita a ligação de periféricos de construção caseira, e os sinais vídeo podem usar-se para controlar um monitor. Como o manual do *Spectrum* fornece pouquíssima informação sobre a natureza desses sinais, vale a pena incluir uma lista deles com breves comentários. As ligações do lado A estão no lado da placa que contém as componentes, e as ligações do lado B estão no outro lado.

Os sinais vídeo disponíveis em 15B, 16B, 17B e 18B admitem a ligação de um monitor, melhorando assim a qualidade da imagem produzida pelo *Spectrum*. O sinal vídeo composto em 15B é uma ligação directa à saída de dois transistores (emissor-seguidor) que alimentam o modulador UHF, tendo, por isso, potência suficiente para controlar, directamente, um monitor. O único problema é que esta saída vídeo, com 75 ohm de impedância, não é a normalizada, e a maioria dos monitores não trabalha muito bem com ela. Os três sinais de cor em 16B, 17B e 18B são todos (sem *buffer*) do ULA, e não têm potência suficiente para guiar um monitor directamente.

1A	b15 do *bus* de endereços	
2A	b13 do *bus* de endereços	
3A	b7 do *bus* de dados	
4A	não está ligado	
5A	SLOT	
6A	b0 do *bus* de dados	
7A	b1 do *bus* de dados	
8A	b2 do *bus* de dados	
9A	b6 do *bus* de dados	
10A	b5 do *bus* de dados	
11A	b3 do *bus* de dados	
12A	b4 do *bus* de dados	
13A	INT	linha de interrupção do *Z80;* ligando-a a +5 pararão as interrupções geradas pelo ULA
14A	NMI	linha de interrupções não mascaráveis do *Z80;* esta interrupção não é utilizada pelo *Spectrum*. Um impulso «baixo» causará o reiniciar do sistema.
15A	HALT	linha de paragem do *Z80* que assinala que foi executada uma instrução *halt* em código máquina.
16A	MREQ	linha do *bus* de controle normalizado do *Z80*
17A	IORQ	linha do *bus* de controle normalizado do *Z80*
18A	RD	linha do *bus* de controle normalizado do *Z80*
19A	WR	linha do *bus* de controle normalizado do *Z80*
20A	−5V	alimentação de corrente «baixa» − 5V .
21A	WAIT	linha de espera do *Z80* que mantida «baixa» parará, temporariamente, o *Z80*. A espera não deve ser maior do que cerca de 1 ms, se não, a memória dinâmica alterar-se-á!
22A	+ 12V	alimentação suavizada de 12 V
23A	+ 12V	alimentação não suavizada de 12 V
24A	M1	linha *bus* de controle do *Z80* normal
25A	RFSH	sinal para refrescar a memória do *Z80*
26A	b8 do *bus* de endereços	
27A	b10 do *bus* de endereços	
28A	não está ligada	
1B	b14 do *bus* de endereços	
2B	b12 do *bus* de endereços	
3B	alimentação de 5 V	

4B alimentação de 9 V
5B SLOT
6B 0 volts
7B 0 volts
8B CK relógio do sistema *Z80* — 3,5 MHZ
9B bl do *bus* de endereços
10B bl do *bus* de endereços
11B b2 do *bus* de endereços
12B b3 do *bus* de endereços
13B IORQGE mantendo «alta» esta linha (ou seja, +5 V) fará com que o ULA deixe de responder a pedidos de entrada/saída. Com circuitos apropriados poderia utilizar-se para expandir o número de dispositivos I/O seleccionáveis pelo *Spectrum*
14B OV terra do vídeo
15B sinal composto de cor (vídeo)
16B sinal vídeo Y
17B sinal vídeo V
18B sinal vídeo U
19B BUSRQ linha *bus* de controle normalizado do *Z80*
20B RESET ligando momentaneamente esta linha a 0 V reiniciará a máquina como se tivesse sido desligada e ligada logo a seguir.
21B b7 do *bus* de endereços
22B b6 do *bus* de endereços
23B b5 do *bus* de endereços
24B b4 do *bus* de endereços
25B ROMCS ligando esta linha a +5 V eliminar-se-á a ROM do BASIC do mapa da memória do *Spectrum*.
26B BUSACK linha *bus* de controle do *Z80* normal
27B b9 do *bus* de endereços
28B b11 do *bus* de endereços

Este facto torna essencial o uso de um amplificador de *buffer*, e a derivação de um sinal RGB normalizado necessita de um circuito subtractivo bastante complicado. No fim de tudo, é muito mais fácil usar o sinal vídeo composto, presente em 15B! Em muitos *Spectrums*, alguns destes sinais vídeo não estão ligados, e este simples facto explica por que falharam muitas tentativas para ligar monitores! A solução é simples — no interior do *Spectrum*, próximo dos circuitos vídeo, na zona mais à esquerda do cartão onde está o circuito impresso, estão quatro uniões marcadas por U, V, Y e VID. Se os terminais dessas uniões estiverem ligadas por fio eléctrico, os sinais vídeo aparecerão no conector de expansão. Contudo, se os terminais estiverem ligados apenas por uma linha branca, será preciso soldar um fio eléctrico para que os sinais se apresentem no conector.

O DIAGRAMA DO SISTEMA

Descritas todas as secções do *Spectrum*, podemos apresentar agora um diagrama completo do sistema. Na figura 2.6 reconhecem-se os

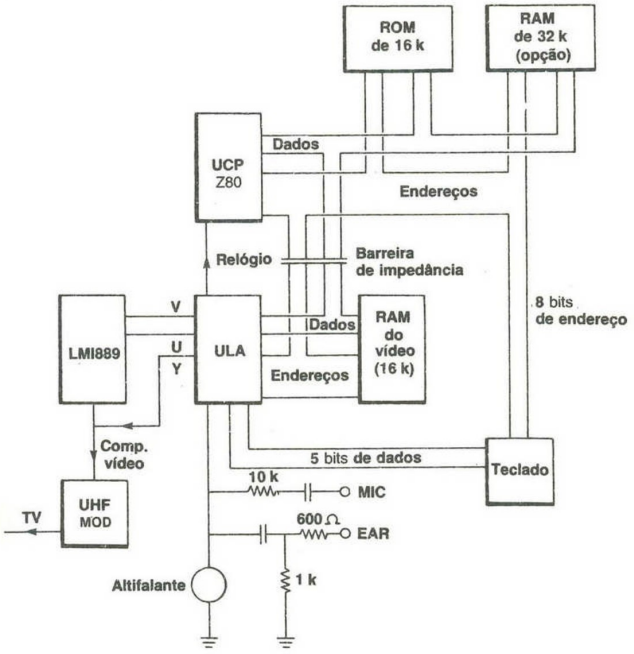


Fig. 2.6. *Diagrama geral do* Spectrum.

pormenores estudados nas secções anteriores. Embora tenham interesse os princípios em que se baseia a operação do *Spectrum*, as características mais importantes do equipamento, sob o prisma do programador, são os dispositivos de I/O. Os conhecimentos adquiridos sobre o teclado, altifalante e *interface* da *cassette* serão usados em capítulos posteriores, para se obterem resultados mais eficientes com um *Spectrum* que não esteja modificado.

CAPÍTULO 3

No interior do BASIC «ZX»

Este capítulo e os dois seguintes tratam do funcionamento interno do BASIC *ZX*. Não haveria grande interesse na explicação pormenorizada do modo como funciona o *hardware* do *Spectrum*. Da mesma forma, não há grande interesse em dar uma lista completa do BASIC existente na ROM. Essa lista daria sem dúvida toda e qualquer informação que o leitor desejasse alguma vez, mas a maior parte dela seria irrelevante. Se se escrevem programas em código máquina, então, sim, será útil saber quais as sub-rotinas presentes na ROM do BASIC, para se poder utilizá-las. Mas, se se programar em BASIC, é mais importante conhecer o modo como o BASIC organiza a memória que utiliza. Conhecendo os métodos gerais que o BASIC segue na execução dos seus comandos, podem também imaginar-se soluções mais criativas e económicas.

A primeira parte deste capítulo descreve a divisão da RAM pelo BASIC e a finalidade de cada uma dessas diferentes áreas. E dá uma visão global das várias secções, a maioria das quais é tratada subsequentemente com mais pormenor. Depois descreve os muitos e variados usos da área da memória onde estão as variáveis do sistema. O capítulo 4 trata da organização, na memória, das linhas do programa BASIC e respectivas variáveis, e no capítulo 5 considera-se o método usado pelo BASIC *ZX* para alargar os comandos PRINT e INPUT a outros dispositivos de I/O que não sejam o *écran* e o teclado.

O MAPA DA MEMÓRIA

Sempre que se liga o *Spectrum*, ele segue um processo de inicialização que determina quanta memória está disponível (normalmente 16 k ou 48 k) e divide-a em várias áreas, que se apresentam no «mapa da memória» da figura 3.1. Note-se que algumas das fronteiras entre áreas contíguas da memoria são fixas e outras variáveis. Por exemplo, a zona da memória que contém a imagem do *écran* começa sempre em 16384 e termina em 22528, mas a localização de um dado programa BASIC na memória depende do espaço ocupado pelo mapa da *microdrive* e pela

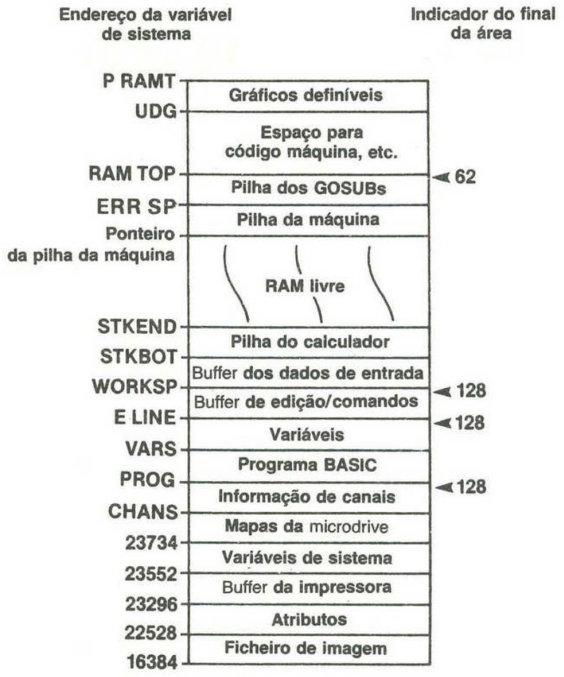


Fig. 3.1. *Mapa da memória do* ZX Spectrum.

informação relativa aos canais. Semelhantemente, o começo da área onde se guardam variáveis depende da área ocupada pelo programa. Os endereços dos locais onde começam estas áreas móveis (e ocasionalmente onde terminam) são guardados na área das variáveis de sistema juntamente com outra informação acerca da condição actual em que se encontra o *Spectrum*. Este conceito de guardar um endereço na própria memória não é difícil, depois de o leitor praticar com ele muitas vezes. Por exemplo, CHANS é uma variável de sistema onde está o endereço do início da área dos canais. Esta variável de sistema ocupa duas posições na memória em 23631 e 23632 (lembremo-nos de que uma posição de memória

só tem oito *bits*, e que um endereço possui 16 *bits*). Assim, se se quiser saber o endereço onde começa a área com a informação dos canais dever-se-á «espreitar» (PEEK) qual o conteúdo das posições 23631 e 23632. É importante saber que o BASIC *ZX* não reconhece nomes como CHANS e outros. Querendo ter acesso à informação guardada em CHANS tem de utilizar-se o seu endereço. A área das variáveis de sistema é uma zona da RAM do *Spectrum* com tanto interesse que se lhe dedica toda uma secção. Seguidamente, descrevem-se outras áreas:

Ficheiro de écran (16384 a 22527)
Utilizada para guardar informação acerca dos *pixels* (ou seja, tinta e fundo) de todo o *écran* (24 linhas de 32 colunas). O seu formato de armazenamento da informação é estudado mais amplamente no capítulo 6.

Atributos (22528 a 23295)
Utilizados para guardar os atributos de cada posição contendo um caracter do *écran* de 24 linhas e 32 colunas. Também se estuda este tópico, com mais pormenor, no capítulo 6.

«Buffer» da impressora (23296 a 23551)
Usado para guardar uma única linha de 32 caracteres, que será enviada à impressora *ZX*. Note-se que os caracteres se guardam em modelos de 8 por 8 pontos em vez de códigos ASCII. Consequentemente, a impressão do conteúdo deste *buffer* consiste simplesmente na transferência de cada fila completa de pontos para a impressora. Se não se utilizar a impressora *ZX*, pode usar-se esta área da memória para guardar funções USR em código máquina. Consulte-se o capítulo 7 para maior informação.

Variáveis de sistema (23552 a 23733)
Usadas para guardar uma gama muito variada de valores, que reflectem a condição em que se encontra o *Spectrum* num dado instante. Esta área estuda-se com profundidade mais adiante, neste capítulo.

Mapas de «microdrive» (23734 a CHANS)
Os mapas de *microdrive* servem para informar quais são os sectores

da *cartridge* livres e quais estão a ser utilizados num momento dado. É óbvio que, no caso de não se utilizarem *microdrives*, esta área não existe e CHANS é posto em 23734.

Informação dos canais (CHANS a PROG-2)
Esta área serve para guardar informação relativa aos caudais *(streams)* associados a cada canal. O capítulo 5 dedica-se ao estudo sobre caudais e canais.

Programa BASIC (PROG a VARS-1)
Usada para guardar as linhas de texto que constituem um programa BASIC. O programa não é guardado na mesma forma em que aparece no *écran*. Algumas das suas partes estão em código, para poupar memória ou tempo de execução. Este assunto é tratado no capítulo 4.

Variáveis (VARS a E LINE-2)
Esta área serve para guardar as variáveis criadas durante a execução do programa. Convém notar que esta área só fica limpa logo após ser dado o comando RUN; por isso, as variáveis criadas por um dado programa existem até se executar de novo esse programa ou outro programa qualquer, ou até se dar um dos comandos NEW ou CLEAR. Também este assunto será tratado mais amplamente no capítulo 4.

«Buffer» de edição (E LINE a WORKSP-1)
Usada para guardar um comando ou uma linha do programa enquanto está a ser editada.

«Buffer» de entrada de dados (WORKSP a STKBOT-1)
Usada para guardar os dados lidos em comandos INPUT e para outras aplicações mistas de armazenamento da informação.

Pilha de cálculo (STKBOT a STKEND-1)
Usada durante os cálculos de qualquer expressão numérica ou de texto para guardar resultados intermédios. O modo de funcionamento de uma pilha é explicado no próximo capítulo.

Pilha da máquina (ponteiro da pilha a ERRSP)
Pilha da máquina usada pelo *Z80* para guardar dados temporários, endereços de retorno e outros. A partir do BASIC, é impossível encontrar o endereço mais baixo utilizado pela pilha da máquina, porque o endereço do fim da pilha está armazenado permanentemente num registo do *Z80*, chamado «ponteiro da pilha» (*stack pointer*, abrev. sp).

Pilha de GOSUB (ERRSP+1 a RAMTOP)
Esta área serve para guardar os números das linhas utilizadas por instruções RETURN. A operação desta pilha confunde-se com a operação da pilha da máquina, por isso não é fácil alterar, através dos POKES, os endereços de retorno. Explica-se o funcionamento pormenorizado desta pilha no capítulo 4.

Gráficos definidos pelo utilizador (UDG a PRAMT)
Esta área serve para guardar os modelos por pontos, associados com os caracteres definidos pelo utilizador. Este tópico será tratado, novamente, no capítulo 6.

Note-se que o *Spectrum* usa a memória de «ambos os lados». O armazenamento do programa e das variáveis começa do lado dos endereços menores da RAM e expande-se para valores superiores, se assim for necessário. Por outro lado, as pilhas da máquina e o de GOSUB começam em endereços altos da RAM e vão diminuindo. Isto significa que a RAM livre está situada entre STKEND e o endereço interno do *Z80*, chamado «ponteiro de pilha».

VARIÁVEIS DE SISTEMA

Já foi mencionada a utilização das variáveis de sistema para guardar os endereços das separações entre as diferentes áreas da memória. Na realidade, a área das variáveis de sistema serve para guardar muita informação diferente que pode ser muito útil ao programador de BASIC. No capítulo 25 do manual do *Spectrum* encontra-se uma lista completa das variáveis ordenadas pelo número de endereço. No entanto, a melhor classificação é de acordo com a sua finalidade e não de acordo com a sua posição na memória. Há cinco grupos de variáveis de sistema:

(1) as variáveis de separação da RAM
(2) as variáveis da condição do teclado

(3) as variáveis do estado em que se encontra o sistema
(4) outras variáveis de I/O
(5) as variáveis que definem o *écran* de vídeo

A fim de evitar repetições, as outras variáveis de I/O serão descritas no capítulo 4 e as variáveis do *écran* de vídeo no capítulo 5. Descreve-se, a seguir, a utilização dos outros três grupos.

Antes de continuar deve primeiro observar-se um problema comum a todos os grupos: como se guarda um endereço de 16 *bits* num par de posições de memória com oito *bits* cada uma. A um nível, a resposta é óbvia. Listando-se os *bits* do número com 16 *bits* (chamando-lhes b0. b1.....b15), o problema soluciona-se pelo uso de uma posição de memória para guardar os *bits* de b0 a b7, e outra posição para guardar os b8 até b15. A posição de memória associada aos b0 a b7 chama-se «byte menos significativo» e a posição de memória associada aos b8 até b15 recebe o nome de «byte mais significativo». No *Spectrum,* com muito poucas excepções, o *byte* menos significativo guarda-se na posição de memória cujo endereço é menor. Assim, se a posição N tem os *bits* de b0 a b7, a posição N+1 terá os *bits* b8 até b15. É fácil calcular o equivalente decimal de um número de 16 *bits* a partir das duas metades com oito *bits*. Se obtivermos, com PEEK, o *byte* menos significativo, chegaremos ao valor decimal correcto, mas o valor obtido pela leitura do *byte* mais significativo terá que ser multiplicado por 256. Porquê? Porque os pesos dados a cada *bit* do *byte* menos significativo são 128, 64, 32, 16, 8, 4, 2, 1, e estes estão correctos para os *bits* b7 até b0, seja de um número binário com 16 *bits* ou com oito. No entanto, esses mesmos pesos são utilizados para o *byte* mais significativo, isto é, para os *bits* de b15 a b8, que deveriam ser 32768, 16384, 8192, 4096, 2048, 1024, 512, 256. E estes diferem dos do primeiro conjunto por um factor de 256. Assim, se se guardar um número de 16 *bits* nas duas posições da memória N e N+1, a função seguinte, definida pelo utilizador, calculará o seu equivalente decimal:

```
DEF FND (N)=PEEK(N)+256*PEEK(N+1)
```

Semelhantemente, se desejarmos introduzir um número de 16 *bits,* com POKE, que corresponda ao número decimal V, nas posições da memória N e N+1, faremos:

```
POKE N, V-256*INT (V/256)
POKE N+1,INT (V/256)
```

As expressões V-256*INT (V/256) e INT (V/256) aparecem tantas vezes neste tipo de aplicação que vale a pena o leitor definir uma função para cada uma. A expressão V-256*INT (V/256) calcula o resto da divisão entre V e 256, e INT (V/256) é simplesmente o número inteiro de vezes que 256 cabe em V. A função

```
DEF FNH (V)=INT (V/256)
```

devolve o equivalente decimal do *byte* mais significativo de V e

```
DEF FNL (V)=V-INT (V/256)*256
```

devolve o equivalente decimal do *byte* menos significativo de V.

UTILIZAÇÃO DAS VARIÁVEIS DE SEPARAÇÃO DA RAM

Todas as variáveis de separação da RAM foram descritas em relação ao mapa da memória dado anteriormente. Nesta secção explicaremos algumas das suas aplicações para o programador de BASIC *ZX*. A aplicação mais evidente é achar quanta memória se está a usar e com que finalidade. Por exemplo, a diferença entre o endereço guardado em PROG e o que está em VARS dirá quanta memória está a ser ocupada por um programa BASIC. A sub-rotina seguinte imprimirá quanta memória foi atribuída ao programa e às variáveis e quanta memória está livre:

```
9000 DEF FN p(N)=PEEK N+256*PEEK
 (N+1)
9010 PRINT "Programa: ";FN p(236
27)-FN p(23635)
9020 PRINT "Variaveis: ";FN p(23
641)-FN p(23627)
9030 PRINT "Livre: ";FN p(23613)
-FN p(23653)
9040 RETURN
```

Usaram-se as seguintes variáveis do sistema:

23627 VARS
23635 PROG
23641 E LINE

23613 ERR SP
23653 STKEND

A memória livre estimada por este programa não inclui a memória usada pela pilha da máquina: por isso está sobrestimada em cerca de 10 posições de memória. A ROM do *Spectrum* contém uma rotina em código máquina que devolve a quantidade exacta da memória disponível, mas não existe uma garantia de que a sua localização seja mantida em futuras versões da ROM. No entanto, talvez se queira substituir a linha 9030 por:

```
9030 PRINT "Livre: ";65536-USR 7
962
```

que deverá imprimir um resultado muito aproximado ao anterior.

Um outro uso das variáveis de separação reside em achar o começo da área do programa ou da área das variáveis, para as examinar. Um caso destes será exemplificado no próximo capítulo.

AS VARIÁVEIS DE ESTADO DO TECLADO

As variáveis que representam o estado em que se encontra o teclado servem para controlar o comportamento do teclado. Isto torna-se muito útil em programas de aplicação que necessitem de condicionar a resposta do teclado à informação introduzida. As variáveis do sistema a tomar em conta são

KSTATE — 8 posições de memória de 23552 até 23559
Regista as teclas que foram premidas, a fim de controlar a sua repetição automática, e, de facto, não interessa ao programador de BASIC.

LAST K — 1 posição de memória em 23560
Guarda o código da última tecla premida. Em cada 1/50 de segundo, verifica-se se mudou, a não ser que o *Spectrum* esteja a carregar ou gravar um programa ou a emitir um som. A rotina INPUT utiliza o valor de LAST K para certificar-se de que não perdeu nenhuma tecla que haja sido premida. Neste sentido actua como um *buffer* de um único carácter, premido com demasiada rapidez! Para observar o funcionamento de LAST K pode ensaiar-se:


```
10 PRINT CHR$ PEEK 23560
20 GO TO 10
```

que imprime o carácter correspondente ao código guardado em LAST K. Note-se que INKEY$ lê o teclado directamente e, por isso, ignora LAST K.

REPDEL — 1 posição de memória em 23561 — e REPPER — 1 posição de memória em 23562
Estas duas variáveis do teclado devem ser consideradas em conjunto porque ambas controlam aspectos da repetição automática. REPDEL define o intervalo de tempo que se pode manter carregada uma tecla antes que comece a repetição, a REPPER define o ritmo de repetição da tecla. Com POKE, podemos introduzir valores nestas variáveis para alterar o modo de comportamento do teclado. Por exemplo, para que o teclado responda com um ritmo de repetição quase intantâneo usaremos:

POKE 23561,1: POKE 23562,1

Estes ritmos muito altos são úteis quando, por exemplo, se utiliza o teclado para controlar o movimento de gráficos nos jogos. Pondo ambas as variáveis a zero, verificaremos que o ritmo de repetição é muito lento, o que será útil para principiantes.

RASP — 1 POSIÇÃO DE MEMÓRIA EM 23608; E PIP — 1 posição de memória em 23609
Estas duas posições controlam a duração de sons característicos associados com o teclado. O valor de RASP altera a duração do som de aviso que acompanha um erro, como, por exemplo, a introdução de uma linha demasiado comprida. O valor de PIP altera a duração do som produzido quando se prime uma tecla. Habitualmente, esta duração é tão pequena que o som se reduz a um «clique». Experimentando diferentes valores de PIP, conseguem-se vários sons.

AS VARIÁVEIS DE ESTADO DO SISTEMA

O BASIC *ZX* utiliza as variáveis que traduzem o estado do sistema

para seguir, em cada instante, a condição da máquina. A maioria destas variáveis não tem interesse para o programador de BASIC e não se podem alterar. Incluído neste grupo, contudo, está o contador de tempo, constituído por três posições da memória, que começa em 23672. E conta o número de imagens de TV que foram expostas desde que se ligou o *Spectrum*. A função

```
DEF FN t()=(PEEK 23672+256*PEEK
23673+65536*PEEK 23674)/50
```

devolve o tempo, em segundos, desde que o *Spectrum* foi ligado. Para o pôr a zero utiliza-se

```
POKE 23674,0:POKE 23675,0:POKE 23676,0
```

Existem duas outras variáveis incluídas neste grupo com utilidade para os programadores de assembly do *Z80*.

ERR NR — 1 POSIÇÃO DE MEMÓRIA EM 23610

Contém um valor igual ao código de erro menos um. Poderia usar-se para implementar um tipo de instrução ON ERROR GOTO que alargasse o BASIC *ZX*, mas não é um projecto fácil.

ERR SP — 2 POSIÇÕES DE MEMÓRIA EM 23613

Contém o endereço de um par de posições de memória na pilha da máquina. Estas posições contêm o endereço da rotina em código máquina, na ROM do BASIC *ZX*, que será executada se ocorrer algum erro. Se diminuirmos o conteúdo deste par de posições em duas unidades, notaremos que a tecla BREAK foi desactivada, e que se ocorrer um erro qualquer a máquina deixará de funcionar normalmente. Um programador de assembly do *Z80* poderá alterar o endereço de retorno do erro, de forma a substituir o tratamento normal desse erro por uma rotina de erros diferente. No entanto, isto não é tão fácil como parece, porque o *Spectrum*, ao funcionr, altera o endereço de retorno do erro, para permitir o tratamento de diferentes tipos de erros. Por exemplo, um erro na entrada de dados durante um comando INPUT não causa qualquer problema à máquina; faz simplesmente com que o editor de INPUT peça nova entrada de dados. Este projecto constitui um desafio possível, embora difícil!

A MEMÓRIA MÓVEL

Como se disse anteriormente, à medida que se introduz ou executa um programa, muitas das áreas da memória variam. Por exemplo, de cada vez que se introduz uma linha de BASIC, a área do programa aumenta. O que não é tão evidente é que, de cada vez que varia uma área da memória, há que deslocar todas as áreas depois dessa, sendo também alteradas todas as variáveis do sistema, que marcam as respectivas separações. Por exemplo, ao conseguir mais espaço na área dos dados de entrada, a pilha de cálculo tem de deslocar-se para cima. Todo este complicado processo é executado, automaticamente, pelo BASIC ZX, mas é proveitoso saber um pouco mais acerca dele.

Sempre que é necessário arranjar um espaço de x *bytes* na área de mamória, toda a memória acima da área e abaixo do STKEND se desloca para cima. Depois examinam-se, uma por uma, as quinze variáveis do sistema, começando em VARS (23627) e acabando em STKEND (23653). Se a variável contiver um endereço que esteja acima da área de memória que estava a ser dilatada, o endereço aumenta de x. Segue-se o processo oposto quando se reduz uma área de memória em x *bytes*. Por outras palavras, move-se x *bytes* para baixo toda a memória que esteja acima da área, e as quinze variáveis do sistema, que contenham endereços acima da área, diminuem de x.

Esta mudança e ajuste das variáveis do sistema tem de ser tomada em conta por quaisquer programas em assembly do *Z80* que modifiquem a posição habitual de qualquer área da memória ou o valor de qualquer variável do sistema. Por exemplo, no capítulo 5, as deslocações da memória causam problemas se a área da memória indicada por CURCHL for posicionada acima da área de edição da entrada de dados e acima de STKEND. Embora a área da memória fique na mesma posição, porque está acima de STKEND, a sua variável de sistema contém um endereço que está acima da referida área de edição; por isso se ajustou como se a «área» se tivesse movido. O resultado é um *crash* irrecuperável do sistema! Outra consequência dos movimentos da memória é o não se ter a certeza de que o que estiver guardado acima de 23734 permaneça na mesma posição de memória: talvez se soubesse onde estava no início do programa, mas isso não significa que aí permaneça durante toda a respectiva execução. Como consequência, é necessário localizar cada artigo variável, linha de programa, etc., de cada vez que se

precisa dele, a não ser que haja a certeza de que não se moveram. No capítulo seguinte, damos alguns exemplos de localização de elementos na memória.

CONCLUSÃO

Neste capítulo, descreve-se com bastante pormenor a organização global da memória do *Spectrum*. No entanto, só em capítulos seguintes serão estudados e exemplificados muitos dos conceitos introduzidos, para explorar a sua relação com outros tópicos. Se, nos capítulos seguintes, não se conseguir situar alguma ou mais áreas de memória, deverá usar-se como guia o mapa da memória, fornecido neste capítulo.

CAPÍTULO 4

A estrutura do BASIC «ZX»

O BASIC *ZX* é uma implementação, completamente nova, de BASIC, com muitas características próprias. Particularizando, a forma como trata as variáveis alfanuméricas é totalmente diferente dos métodos utilizados pela norma, mais antiga e complicada, da Microsoft. Neste capítulo, o tópico considerado não é a aparência exterior do BASIC *ZX*, mas, sim, a forma como organiza e utiliza a memória para implementar algumas das vantagens mais importantes que fornece. A descrição mais completa do funcionamento do BASIC *ZX* consistirá, sem dúvida, numa listagem da ROM com o BASIC *ZX*. No entanto, a informação desta listagem é, em grande parte, excessiva. A maior parte da ROM dedica-se a implementação minuciosa de aritmética, funções, etc., com algum interesse, mas geralmente de pouca utilização.

A melhor maneira de compreender o funcionamento do BASIC *ZX* é estudar o modo como ele organiza e usa a memória e os princípios que determinam a forma como implementa os GOTOs, GOSUBs, ciclos FOR NEXT, etc. Este conhecimento torna mais fácil a compreensão do esquema global de uma listagem da ROM: na maioria dos casos, também basta para tornar desnecessária a consulta da listagem da ROM. Para o demonstrar, damos alguns exemplos práticos de manipulação da área do programa e das variáveis e, ainda, de alteração do funcionamento do BASIC *ZX*. Todos os exemplos estão escritos em BASIC *ZX* para terem o máximo de acessibilidade; mas para quem conhecer e estudar assembly do *Z80* todos esses exemplos beneficiarão do aumento da velocidade de execução resultante da sua reconversão para essa linguagem.

O FORMATO DAS VARIÁVEIS – UM PROGRAMA PARA UM BLOCO DE DADOS DAS VARIÁVEIS

O capítulo 24 do manual do *Spectrum* dá grande parte da informação sobre o formato de diferentes tipos de variáveis, criadas pelo BASIC *ZX*, na área da memória reservada a variáveis. Apesar disso, vale a pena resumir essa informação para mostrar o sistema em que se baseiam todos os formatos.

Todos os seis diferentes tipos de variáveis do BASIC *ZX* são armazenados na área de memória reservada a variáveis, começando

com um *byte* único que serve não só para identificar o tipo de variável mas também para guardar o primeiro (e talvez único) carácter do seu nome. Não é difícil compreender a forma como se juntam estas duas informações num *byte* único. O BASIC *ZX* interpreta, da mesma forma, as letras maiúsculas ou minúsculas que formam o nome de uma variável; por isso a primeira letra do nome de uma variável pode ser sempre guardado em letra minúscula. Poderia utilizar-se o código ASCII de cada uma das 26 minúsculas mas isso ocuparia uma posição de memória (8 *bits*) quando de outra forma bastam 5 *bits*. É muito mais eficiente armazenar um número de 0 a 25 para indicar por qual das 26 letras começa o nome da variável. Isto também liberta três *bits* da posição de memória para guardar o «código do tipo da variável». Assim, o formato do primeiro *byte* de cada variável é o seguinte:

b7 b6 b5 b4 b3 b2 b1 b0

código de tipo	código de letra

Os códigos usados para o tipo da variável são:

2 variável de texto
3 variável numérica denominada por uma única letra
4 vector numérico
5 variável numérica denominada por mais de uma letra
6 vector de texto (isto é, uma variável de texto dimensionada)
7 índice (isto é, uma variável usada como índice de um ciclo FOR NEXT)

Note-se que os códigos do tipo são convertidos num número binário de três algarismos, e o modelo binário resultante associa-se aos *bits* b7 a b5. Por exemplo, o código do tipo 5 é equivalente ao binário 101 e por isso b7=1, b6=0 e b5=1. O código ASCII da primeira letra do nome da variável pode ser calculado a partir do valor de b4 a b0, adicionando-lhe simplesmente 96. Assim, se A contém o endereço da primeira posição de memória onde se guarda uma variável, a função seguinte

```
DEF FNt(A)=INT(PEEK(A)/32)
```

devolverá o tipo do código dessa variável, de acordo com a tabela anterior, e

```
DEF FNc$(A)=CHR$(PEEK(A)-FNt(A)*32+95)
```

devolverá a primeira letra do nome dessa variável. Estas duas funções utilizam técnicas BASIC de tratamento de *bits*, descritas no capítulo anterior.

O que se segue ao primeiro *byte* de cada variável depende do tipo dessa variável. Estes formatos são descritos pormenorizadamente

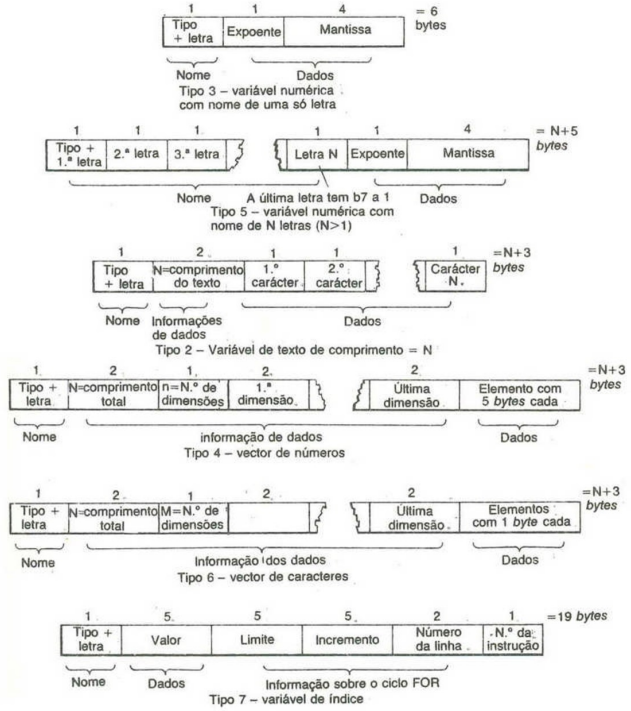


Fig. 4.1. *Modo de armazenamento das variáveis no BASIC ZX.*

no capítulo 25 do manual do *Spectrum* e reproduzem-se, com comentários adicionais, na figura 4.1. Embora seja importante para o programador de assembler o conhecimento destes formatos, o programador de BASIC *ZX* pode utilizar as funções VAL e VAL$ para saber o que contém uma dada variável. Por exemplo, se N$ contém o nome de uma variável não vectorizada,

PRINT VAL(N$)

imprimirá o seu conteúdo se for uma variável numérica, e

PRINT VAL$(N$)

imprimirá o seu conteúdo se for uma variável de texto; expressões semelhantes servem para imprimir qualquer elemento de uma variável que designe um vector. Por exemplo, se N$ contiver o nome de uma única variável numérica dimensionada,

PRINT VAL(N$+"("+STR$(I)+")")

imprimirá o conteúdo do elemento I. O modo de conseguir esta impressão de elementos de vectores é formar o nome completo do elemento numa variável de texto e depois utilizar VAL ou VAL$ para avaliá-lo.

Conjugando este método para descobrir o conteúdo de uma variável com as duas funções dadas anteriormente e ainda com o conhecimento do número de posições de memória ocupadas por cada tipo de variável, consegue-se listar todas as variáveis usadas por um dado programa. Apresentamos a seguir este programa (um *dump* das variáveis):

```
9100 DEF FN t(A)=INT (PEEK (A)/3
2)
9110 DEF FN c$(A)=CHR$ (PEEK (A)
-FN t(A)*32+96)
9120 DEF FN v()=PEEK (23627)+256
*PEEK (23628)

9130 LET V0=FN v()
9140 PRINT "Variavel";TAB (15);"
Tipo";TAB (25);"Valor"
9150 DIM N$(15): DIM T$(10)

9200 IF PEEK (V0)=128 THEN STOP
9210 LET I0=1: LET N$="": LET T$
="Numerico": LET N$=""
9220 LET N$(I0)=FN c$(V0)
9230 IF FN t(V0)=3 THEN GO TO 92
60
9240 IF FN t(V0)<>5 THEN GO TO 9
300
9250 LET V0=V0+1: LET I0=I0+1
9260 LET N$(I0)=CHR$ (PEEK (V0)-
INT (PEEK (V0)/128)*128)
9270 IF INT (PEEK (V0)/128)*128=
0 THEN GO TO 9250
9280 LET V0=V0+6
9290 PRINT N$;TAB (15);T$;TAB (2
5);VAL (N$)
9295 GO TO 9200

9300 IF FN t(V0)<>7 THEN GO TO 9
350
9310 LET T$="Indice"
9320 LET V0=V0+19
9330 GO TO 9290

9350 IF FN t(V0)<>2 THEN GO TO 9
400
9360 LET T$="literal": LET N$(2)
="$"
9370 LET V0=V0+PEEK (V0+1)+256*P
EEK (V0+2)+3
9380 PRINT N$;TAB (15);T$;TAB (2
5);VAL$ (N$)
9390 GO TO 9200

9400 IF FN t(V0)=6 THEN LET N$(2
)="$"
9410 LET T$="Vector"
9420 LET I0=0
9430 PRINT N$;TAB (15);T$;TAB (2
5);"DIM(";
9440 PRINT PEEK (V0+4+I0*2)+256*
PEEK (V0+5+I0*2);
9450 LET I0=I0+1
9460 IF I0<>PEEK (V0+3) THEN PRI
NT ",";: GO TO 9440
9470 PRINT ")"
9480 LET V0=V0+3+PEEK (V0+1)+256
*PEEK (V0+2)
9490 GO TO 9200
```

A primeira parte do programa define três funções úteis. As funções FNt e FNc$ já foram dadas e FNv devolve o endereço onde começa a área das variáveis, na memória. As linhas de 9130 a 9150 imprimem um título, inicializam a variável V0, que indica a posição

actual na memória, e dimensionam dois vectores utilizados no programa. N$ serve para ir construindo o nome de cada variável e T$ para guardar uma descrição do seu tipo. O resto do programa está formado por um grande ciclo que começa na linha 9200. Na linha 9200 verifica-se se foi alcançado o valor 128, que marca o fim da área de variáveis. Da linha 9210 à 9295 constrói-se o nome, em N$, de uma variável numérica e imprime-se o seu valor na linha 9290. Se a variável for do tipo 3, o nome da variável tem uma só letra, já guardada em N$, e a linha 9230 transfere o controle para a linha 9290, que imprime os pormenores relativos à variável. Se for do tipo 5, ao primeiro carácter segue-se uma sequência de letras, constituindo o nome completo da variável (ver figura 4.1). As linhas de 9250 a 9270 vão calculando cada carácter, um de cada vez, e guardam-no em N$. O valor de b7, que é 0 para todos os caracteres, exceptuando o último, marca o final do nome da variável. A linha 9280 adiciona 6 a V0 para obrigá-lo a apontar para o começo da variável seguinte.

Se a variável é do tipo 7, o controle é transferido para a linha 9300. Depois, a linha 9320 ajusta V0 para que aponte à variável seguinte. A linha 9290 imprime os pormenores da variável «índice». Se a variável for do tipo 2, transfere-se o controle para a linha 9360. A linha 9370 põe V0 apontando para o começo da variável seguinte, adicionando-lhe o comprimento da variável de texto (ver figura 4.1). A linha 9380 imprime a informação guardada, nesse momento, na variável de texto usando a função VAL$ como se descreveu anteriormente.

Finalmente, se a variável for do tipo 4, a variável é um vector. Neste caso não se tenta imprimir a informação do vector, porque esta poderia ser imensa! Em seu lugar, imprimem-se as dimensões do vector. A linha 9400 junta um «$» ao nome do vector se este for literal. À parte isto, pode tratar-se os dois tipos de vectores da mesma maneira, porque a informação das suas dimensões também é guardada da mesma forma. O número de dimensões é dado na quarta posição do vector e obtém-se com um PEEK, na linha 9460, para verificar se se imprimiram os valores de todas as dimensões. A linha 9440 imprimirá o valor de uma única dimensão, e I0 é utilizada para contar quantos valores de imprimiram até esse momento. Finalmente, a linha 9480 utiliza o comprimento total do vector para actualizar V0 de modo a que aponte para o começo da variável seguinte.

Se se juntar este programa ao fim de um dos nossos próprios programas, com um GOTO 9100 imprimir-se-á uma lista de todas as variáveis do nosso programa e ainda I0, V0, N$ e T$, que são utilizadas no programa que acabámos de analisar. Usaram-se os vectores N$ e T$, de preferência a variáveis de texto, porque a área das variáveis modifica-se à medida que aumenta ou diminui o número de caracteres numa variável de texto. Se uma variável de texto variasse de extensão enquanto se executasse o programa de *dump* da memória, variaria também a posição de todas as variáveis acima dela, e V0 já não apontaria, necessariamente, para o começo de outra variável. Contudo, um vector de caracteres tem tamanho fixo, e por isso a sua utilização não obriga à variação da área das variáveis. Podem-se agregar mais opções a este programa, como por exemplo imprimir quanta memória ocupa; mas é necessário ter a certeza de não utilizar variáveis de texto dentro do programa de *dump;* doutra forma as coisas correrão mal!

OS FORMATOS DA INFORMAÇÃO NUMÉRICA

A forma como se guardam números dentro de um computador é um assunto bastante teórico. Existem dois métodos principais — armazenamento inteiro e armazenamento decimal. O armazenamento inteiro trata de números num intervalo limitado, mas é rápido e de fácil operação quando se usa aritmética. É, essencialmente, a representação binária de números, que temos utilizado desde o capítulo 1, alargada de forma a incluir tanto números positivos como números negativos. O armazenamento decimal pode usar-se para representar um intervalo imenso de números, mas a aritmética de números decimais é bastante lenta comparada com a dos inteiros. O armazenamento decimal baseia-se num equivalente binário da notação exponencial decimal utilizada em muitas máquinas de calcular.

Muitas versões do BASIC fornecem dois tipos diferentes de variáveis numéricas — inteiras para números inteiros e reais para números decimais. Deixa-se ao programador a escolha do tipo de variável, e quando deve utilizar um tipo ou outro. O BASIC *ZX* também tem estes dois tipos de armazenamento, mas é o BASIC *ZX* que decide quando e qual o tipo que deve utilizar. O número, se está dentro do intervalo possível do armazenamento inteiro, é guardado como um inteiro. Se não, é guardado como um número decimal.

Com este mecanismo, o programador obtém o melhor dos dois tipos de armazenamento, e nunca necessita de preocupar-se com o modo de guardar os valores. No capítulo 24 do manual do *Spectrum* descrevem-se claramente os pormenores dos dois tipos de armazenamento.

A GESTÃO DINÂMICA DAS VARIÁVEIS

Descreveu-se, nas secções anteriores, o formato utilizado no armazenamento de variáveis. No entanto, existe um outro aspecto com interesse. A área das variáveis modifica-se à medida que se criam novas variáveis ou se alteram variáveis de texto. O modo de conseguir esta reordenação pode afectar a eficiência dos programas; por isso são importantes os pormenores da gestão dinâmica da área das variáveis.

A área das variáveis limpa-se com um dos comandos RUN ou CLEAR. As variáveis são criadas à medida que vão sendo encontradas. Para evitar deslocações a mais, as novas variáveis juntam-se à parte superior da área das variáveis. Assim, pelo menos de início, as variáveis vão sendo armazenadas na respectiva área, pela ordem da sua criação. Imagine-se a dificuldade de adicionar um carácter no fim duma variável de texto já existente. Se esta foi criada no começo do programa, cada carácter que se lhe agrega implica o deslocamento, para a posição de memória seguinte, de todas as outras variáveis armazenadas acima dela. Este facto sugere que o seguinte programa

```
10 LET A$=""
20 DIM M(1000)
30 LET A$=A$+"X"
40 GO TO 30
```

obteria maior rapidez se o vector fosse dimensionado (ou seja, criado) antes da variável literal A$ (trocando a ordem das duas primeiras linhas do programa). No programa apresentado, cada vez que se agrega um «x» à variável A$, deslocam-se, aproximadamente, 5000 posições de memória. Se se definir o vector antes da variável, não será necessário deslocar qualquer variável ao agregar um carácter a A$. Este tipo de problema faz com que muitas versões do BASIC baixem de velocidade quando tratam vectores e textos extensos, mas não o BASIC *ZX*! Assim, ambas as versões do programa teriam a mesma velocidade, aproximada, no *Spectrum!*

Isto resulta de o BASIC *ZX* utilizar um método interessante na gestão da área das variáveis. De cada vez que aparece uma variável do texto no lado esquerdo de um comando LET, o seu conteúdo anterior é destruído pela deslocação, para baixo, da área das variáveis para «fechar» o espaço da memória que ela ocupava. E depois é redefinida no cimo da área das variáveis como se fosse uma variável inteiramente nova. Resumindo, as variáveis de texto são redefinidas cada vez que ocorrem no lado esquerdo de um comando LET. Esta redefinição tem dois efeitos. Primeiramente, não há versões antigas de uma dada variável de texto na área das variáveis; por isso não é necessário parar a execução para fazer a «recolha de desperdícios» de vez em quando, como sucede noutros sistemas. Em segundo lugar, a variável de texto usada mais recentemente está sempre no cimo da área das variáveis, e as variáveis de texto mais utilizadas tendem a estar perto do cimo da área das variáveis. Isto minimiza o número de movimentos feitos e o número de posições afectadas por cada movimento devido ao manuseio de variáveis de texto. O leitor deverá agora reconhecer que no programa anterior, agregando uma única letra à variável A$, o vector só será movido uma vez para a variável A$ passar para o cimo da área das variáveis.

Usa-se este mesmo sistema de gestão quando se define um vector. Ao dimensionar um vector e se existir uma versão anterior, o espaço ocupado por esta desaparece pela deslocação, para baixo, das variáveis que estavam acima dessa versão. Depois cria-se um novo vector ao cimo da área das variáveis. Isto significa que é possível redimensionar vectores em BASIC *ZX*, ao passo que outras versões do BASIC tratam os vectores como variáveis de comprimento fixo.

COMO É ARMAZENADO O BASIC «ZX»

O formato de armazenamento de cada linha em BASIC *ZX* é o seguinte (ver figura 4.2): os dois primeiros *bytes* contêm o número

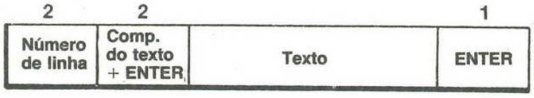


Fig. 4.2. *Formato de uma linha em BASIC.*

da linha, guardado na ordem inversa da maioria dos outros números, isto é, com o *byte* mais significativo em primeiro lugar. O número da linha é utilizado para determinar para onde será transferido o controlo nos GOTOs e GOSUBs e para determinar onde se inserirão novas linhas no programa. As linhas guardam-se por ordem crescente dos seus números de linha. Os dois *bytes* seguintes representam o comprimento do texto, incluindo o carácter ENTER que marca o final de cada linha. Estes dois *bytes* armazenam-se na ordem normal, e são utilizados para achar a posição de memória onde começa a linha seguinte.

Se A for o endereço do início de uma linha em BASIC, a função

```
DEF FNL(A)=256*PEEK(A)+PEEK(A+1)
```

devolve o seu número de linha. A função

```
DEF FNn(A)=PEEK (A+2)+256*PEEK (A+3)+A
```

devolverá o endereço do número da linha seguinte. O fim dum programa ocorre quando FNn(A) é igual ao conteúdo da variável do sistema VARS. Além de examinarmos números de linha, podemos modificá-los, dando-lhes novos valores com POKE. Embora o BASIC *ZX* somente aceite números de linha entre 1 e 9999, poderá funcionar com números entre 0 e 61439. Funcionará no sentido em que os GOSUBs e GOTOs transferirão o controle para números de linha no intervalo maior, mas o editor somente permitirá a edição de linhas cujos números se encontrem dentro do intervalo menor. Pode tirar-se proveito desta anomalia mudando o número da primeira linha para 0, tornando-a assim impossível de apagar. Há métodos mais complexos de utilizar estes números de linha semilegais na protecção de programas, mas, uma vez conhecidos, é muito fácil ultrapassá-los.

A parte de uma linha de programa em forma de texto é armazenada exactamente como foi dactilografada no teclado, com poucas excepções. Em primeiro lugar, todos os comandos da linha são armazenados em *bytes* únicos, correspondendo aos seus códigos, referidos no apêndice A do manual do *Spectrum*. Assim, o comando GOTO não é guardado letra a letra «G», «O», «T» e «O» mas, sim, com o código 236, que ocupa um único *byte*. Em segundo lugar, todas as constantes numéricas são guardadas de duas formas diferentes: como a sequência de dígitos dactilografados no teclado, ou com o formato de cinco *bytes*, utilizado para as

variáveis numéricas. Usa-se a primeira forma quando se lista, e a segunda quando o programa está a ser executado, para poupar tempo de conversão das constantes para o formato interno utilizado em todos os cálculos do *ZX*. O código 14 indica que irá seguir-se um número decimal de cinco *bytes*, e é usado pela rotina LIST, contida na ROM do BASIC *ZX*, para não considerar, nas listagens, números formatados internamente. Além de se seguirem a uma constante numérica, os números decimais de cinco *bytes* poderão ocorrer noutras situações; por isso há que ter sempre em vista a aparição do código 14 quando se inspecciona uma linha em BASIC.

UM DETECTOR DE COMANDOS

Para exemplificar a utilização dos formatos internos do BASIC *ZX*, apresenta-se o programa seguinte, que inspecciona a área onde se encontra o programa e imprime o número de todas as linhas contendo o comando especificado em C$.

```
  10 INPUT C$
  20 GO SUB 9500
  30 STOP

9500 DEF FN p()=PEEK 23635+256*P
EEK 23636
9510 DEF FN v()=PEEK 23627+256*P
EEK 23628
9520 DEF FN L(A)=256*PEEK A+PEEK
 (A+1)
9530 PRINT C$;" EM "
9540 LET S=FN p()
9550 LET F=FN v()
9560 LET L=FN L(S)
9570 LET S=S+4
9580 IF S>=F THEN RETURN
9585 LET C=PEEK S
9590 IF C=13 THEN LET S=S+1: GO
TO 9560
9600 IF C=14 THEN LET S=S+5: GO
TO 9580
9610 IF C<>CODE (C$(1)) THEN LET
 S=S+1: GO TO 9580
9620 PRINT "linha ";L
9630 LET S=S+1
9640 GO TO 9580
```

A procura das linhas com o comando que está em C$ é feita pela sub-rotina que começa em 9500. As linhas de 9500 a 9520 definem

três funções úteis. FNp devolve o endereço onde começa a área do programa, FNv devolve o endereço onde começa a área das variáveis e FNL devolve o número da linha que começa no endereço A. As linhas de 9580 a 9640 formam um ciclo que percorre a área do programa, linha a linha e carácter a carácter, à procura de códigos de caracteres iguais a CODE (C$(1)). A linha 9590 detecta os caracteres ENTER que assinalam o final de cada linha do programa, e a linha 9600 detecta o código 14, indicando que os cinco *bytes* seguintes são a forma interna de uma constante numérica, e por isso não devem ser considerados.

Este simples programa tem grande aplicação prática na verificação de todos os GOSUBs e GOTOs. (Note-se que para introduzir um comando como LET é necessário introduzir primeiro THEN para o cursor passar ao modo K, depois introduzir o comando desejado, e depois apagar aquele THEN.) Podem-se mesmo procurar todas as variáveis do programa começando por uma dada letra, para o que se deve introduzir essa letra única em vez do comando. No entanto, se se procura uma variável cujo nome tenha mais do que uma letra, haverá que modificar o programa para comparar cada letra com o conteúdo da memória. Outro exemplo da aplicação deste programa de procura de comandos pode ser dado alterando as linhas seguintes

```
9620>POKE 23625,L-INT (L/256)*25
6
9630 POKE 23626,INT (L/256)
9640 STOP
```

Estas linhas introduzem, com POKE, na variável de sistema E PPC, o número da primeira linha que contém o comando guardado em C$. Como se usa E PPC para guardar a posição do cursor de edição, esta rotina moverá o referido cursor para a primeira linha onde exista o comando guardado em C$. Agregando uma opção de «procura a partir da última posição», pode modificar-se a rotina de modo a posicionar o cursor de edição, rapidamente, em qualquer parte dum programa.

UM PROGRAMA DE RENUMERAÇÃO DAS LINHAS

À primeira vista parece fácil renumerar um programa em BASIC *ZX*. Tudo o que haveria a fazer era inspeccionar a área do programa alterando os dois *bytes*, no início de cada linha, que guardam o seu número. O mal é que esta estratégia ignora completamente as alterações que devem ser feitas aos números das linhas que fazem parte dos GOTOs e GOSUBs. Não é difícil imaginar alguns algoritmos que ajustem os números dos GOTOs e GOSUBs, mas todos eles obrigam à inspecção da área completa do programa e à procura de cada ocorrência dos comandos GOTO e GOSUB. Tal algoritmo, aplicado a um programa em BASIC *ZX*, seria muito vagaroso.

Como compromisso, a sub-rotina seguinte renumera todas as linhas de um programa, ignorando o problema dos GOTOs e GOSUBS, mas imprime uma listagem que mostra a correspondência entre os novos números de linha e os anteriores, de modo a poderem ser emendados manualmente.

```
9700>LET P=FN p()
9710 INPUT "Linha inicial ";S0
9720 INPUT "Passo ";I0
9730 PRINT "ANTES";TAB (10);"AGO
RA"
9740 LET L=FN L(P)
9750 IF L>9000 THEN STOP
9760 PRINT L;TAB (10);S0
9770 POKE P,INT (S0/256)
9780 POKE P+1,S0-INT (S0/256)*25
6
9790 LET S0=S0+I0
9800 LET P=P+4+PEEK (P+2)+256*PE
EK (P+3)
9810 GO TO 9740
```

A linha 9700, usando a função FNp, definida na secção anterior, atribui a P o início da área do programa. As linhas seguintes, 9710 e 9720, obtêm o número pelo qual deverá começar o programa renumerado, e o incremento de uma linha à seguinte. Na linha 9740 atribui-se a L o número da linha anterior utilizando a função FNL definida na secção anterior. Nas linhas 9770 e 9780 introduz-se, com POKE, o novo número de linha no lugar correcto da linha. Agora na linha 9790, adiciona-se o incremento ao novo número de linha, e a linha 9800 ajusta P de forma a apontar para o início da linha seguinte, adicionando o conteúdo das duas posições de memória que guardam o comprimento da parte da linha contendo texto. A renumeração termina quando o número de linha do programa dado chega a 9000, para evitar renumerar o próprio

programa de renumeração ou qualquer dos outros programas referidos neste capítulo ou em capítulos anteriores.

GOTO

De modo geral, o comando GOTO do BASIC *ZX* funciona como se esperaria, mas há algumas características especiais merecedoras de atenção. Quando se encontra um GOTO, procura-se na área do programa pela primeira linha cujo número seja maior ou igual ao número de linha utilizado no GOTO. Se a busca tiver êxito, passa-se o controle para aquela linha. Se não, o programa termina imprimindo um código normal, daqueles que aparecem na última linha do *écran* sempre que termina a execução de um programa BASIC. Isto significa que é improvável cometer um erro com uma instrução GOTO em BASIC *ZX*, o mesmo não acontecendo em outras versões de BASIC. Em certos aspectos esta forma de GOTO é vantajosa, noutros pode ser desastrosa. Por exemplo, suponhamos que GOTO 4000 aparece num programa sem linha 4000 e que a primeira linha com número maior que 4000 é a linha 5000. Nesta situação, o GOTO 4000 transfere o controlo para a linha 5000, e talvez o programa funcione de acordo com o que o programador deseja. Mas suponhamos que, algum tempo depois, o programador agrega, inocentemente, ao programa uma nova secção começando na linha 4500. Na execução do programa, o GOTO transferiria o controle para 4500 e o programa talvez não funcionasse. A localização deste tipo de erro é muito difícil, porque a origem do problema — o GOTO incorrecto — pertence à parte do programa que não foi alterada! A conclusão a tirar é que se deve sempre assegurar que os GOTOs (e GOSUBs) transfiram o controlo para linhas cujos números existam. Uma segunda característica, não habitual, do BASIC *ZX* é que a linha referida num GOTO (ou GOSUB) pode também ser dada por uma expressão numérica. Por exemplo, em BASIC *ZX*.

```
GOTO 200+10*4
```

tem o mesmo efeito do que GOTO 240. Esta característica é vantajosa em certo número de casos. Por exemplo, pode seleccionar-se uma de várias rotinas, de acordo com valor guardado numa variável, usando

```
GOTO L (I)
```


onde L é um vector que contém os números de linha do início de cada rotina, e o valor de I determina para qual delas se passará o controle. Isto é, se I contiver 1 será executada a rotina começando em L(1), e semelhantemente para outros valores de I. Este artifício também funcionará com GOSUB. Noutras versões de BASIC, chama-se a esta característica um «GOTO calculado» e escreve-se habitualmente

```
ON I GOTO L1, L2, L3,...
```

onde L1, L2, etc., são os números de linha a alcançar quando I é 1, 2, etc., respectivamente.

Outra utilização de expressões que façam parte de GOTOs (ou GOSUBs) é tornar os programas ligeiramente mais fáceis de compreender. Se, por exemplo, uma parte de um programa, começando em 3123, ler informação para tratamento posterior, uma instrução como esta

```
60 GO TO 3123
```

consegue fazer ler os dados, mas não diz nada, a quem ler o programa, do que se passa na realidade. No entanto, se definirmos uma variável, com um nome apropriado, que contenha o número da linha inicial da rotina, os GOTOs (e GOSUBs) tornam-se mais compreensíveis. Por exemplo

```
10 LET LERDADOS=3123
    :  :  :
60 GO TO LERDADOS
```

O BASIC *ZX* pode tratar mais do que uma instrução por linha de programa, utilizando dois pontos para separar essas instruções. Esta possibilidade é muito útil, mas o comando GOTO só pode transferir o controle para o início de uma linha com instruções múltiplas e não, directamente, para uma dessas instruções. Na realidade, o BASIC *ZX* funciona com um número de linha, e ainda com o número de instrução que pertence a essa linha, o que pode ser utilizado para localizar directamente qualquer instrução dum programa, mesmo que ela pertença a uma linha com instruções múltiplas. Por exemplo.

```
1203 PRINT "1": PRINT "2": PRINT
 "3"
```

é uma linha com instruções múltiplas. O comando PRINT «1» é a instrução 1 da linha 1203, PRINT «2» é a instrução 2 da linha 1202, e assim sucessivamente. Embora não exista o comando «GOTO número de linha, número de instrução» que transfira o controle para dentro de uma linha com instruções múltiplas, não é difícil realizá-lo. O par de variáveis do sistema NEWPCC e NSPCC é utilizado para guardar o número da linha e o número da instrução dessa linha para onde se transfere o controle. Pode obrigar-se uma transferência introduzindo, com POKE, o número da linha desejada em NEWPCC e depois introduzir (POKE) em NSPCC o número da instrução. Por exemplo:

```
  10 PRINT 1;: PRINT 2;: PRINT 3
;
  20 LET L=10: LET S=2: GO TO 98
00

9800 POKE 23618,L-INT (L/256)*25
6
9810 POKE 23619,INT (L/256)
9820 POKE 23620,S
```

Executando este programa, aparecerá 123 impresso no *écran* seguido por 23, uma e outra vez, até se carregar em BREAK. A rotina 9800 transferirá o controle para a linha L, instrução S, pelo que a linha 20 é equivalente a GOTO linha 10, instrução 2. Note-se que não se deve saltar para a rotina 9800 com um GOSUB porque a transferência de controle criada pela própria rotina impossibilita a execução de um RETURN!

GOSUB E A PILHA

O comando GOSUB do BASIC *ZX* funciona exactamente da mesma maneira que o comando GOTO, mas armazena informação na pilha dos GOSUBs. Esta é utilizada pelo comando RETURN para transferir o controle de volta à instrução imediatamente a seguir ao GOSUB. Para compreender os comandos GOSUB e RETURN torna-se necessário conhecer um pouco do funcionamento da pilha.

Uma pilha ou, mais propriamente, um «Last In First Out (LIFO) *stack*» é um conjunto de posições de memória com um ponteiro, utilizado para indicar a primeira posição livre. Por exemplo, pode usar-se, como pilha, um simples vector que esteja associado com uma variável, ou ponteiro da pilha, que guarde o índice do primeiro elementos livre desse vector. Introduz-se a informação numa pilha tipo LIFO com uma operação *push*. Esta consiste no armazenamento da informação na posição livre, indicada pelo ponteiro da pilha, e no deslocamento automático do ponteiro para indicar a posição livre seguinte. Semelhantemente, recupera-se informação de uma pilha tipo LIFO com uma operação *pop*. Esta desloca o ponteiro da pilha para a posição previamente utilizada e depois devolve os dados nela armazenados. Se o vector S estiver a ser utilizado como pilha, sendo P o ponteiro da pilha inicializado para apontar ao primeiro elemento de S, uma operação *push* seria

```
LET S(P)=D:LET S=S+1
```

e uma operação *pop* seria

```
LET S=S-1:LET D=S(P)
```

onde a variável D contém a informação a guardar, em ambos os casos. Note-se que nenhuma das rotinas verifica se os limites do vector são excedidos ou não. Uma pilha pode aumentar, como no exemplo dado, ou diminuir, como na maioria das pilhas do *Z80*, começando por ocupar primeiro posições altas da memória.

A característica mais importante de uma pilha LIFO é dada pelo seu próprio nome. A última informação entrada na pilha é a primeira a sair dele. Por exemplo, se A, B e C forem introduzidos na pilha, o primeiro *pop* devolverá C, o segundo B e o terceiro A. Este comportamento é exactamente o que se necessita para implementar o armazenamento de números de linha que seguem um GOSUB e onde o controle deve regressar depois de executada a rotina. Estes números de linha designar-se-ão por "números de linha de retorno". Cada GOSUB introduz um número de linha de retorno na pilha e cada RETURN tira da pilha um número de linha de retorno. Assim, se se executar GOSUB A, GOSUB B e GOSUB C, por esta ordem, o primeiro RETURN transferirá o controle para a linha seguinte a GOSUB C, o segundo RETURN transferirá o controle para a linha seguinte a GOSUB B e o terceiro RETURN passará o controle para a linha imediatamente depois de GOSUB A.

Desta forma uma pilha serve para não esquecer os endereços de retorno e, também, para fornecê-los, na ordem correcta, à medida que vão sendo necessários.

A pilha de GOSUB utilizada no *Spectrum* é um pouco fora do

comum: está confundida com outra pilha, utilizada pelo *Z80* para armazenar o endereço de retorno para o código máquina equivalente a um GOSUB. Para ser mais preciso, a pilha de GOSUB faz parte da pilha do *Z80*. No entanto, a maioria das vezes a variável de sistema ERRSP contém o endereço do primeiro item que está na pilha do *Z80*, e o conteúdo de ERRSP mais dois é, portanto, o endereço do primeiro item que está na pilha de GOSUB.

É interessante notar que, tanto o número de linha como o número da instrução dentro da linha estão armazenados na pilha de GOSUB. Isto significa que um GOSUB pertencente a uma linha de instruções múltiplas regressará (RETURN) à instrução que se lhe siga nessa mesma linha. Por exemplo,

```
  10 GO SUB 1000: PRINT "linha 1
0 sentença 2"
  20 PRINT "linha 20 sentença 1"
```

executará os dois PRINTs porque a sub-rotina 1000 devolve o controle à instrução 2 da linha 10. A informação exacta guardada na pilha GOSUB é, em primeiro lugar, um número de dois *bytes* representando o número da instrução, executada nesse instante, mais um, e em segundo lugar vem o número de linha onde se encontra essa instrução.

Com esta informação consegue-se escrever um programa que obrigará um RETURN a transferir controle para qualquer número de linha e de instrução desejados. Este tipo de saltos desordenados num programa não deve ser encorajado, mas é útil algumas vezes para implementar, a partir das sub-rotinas, RETURNS de erros especiais.

```
10 GO SUB 200
20 PRINT "linha 20"
30 PRINT "linha 30"
40 STOP

100 LET G=2+PEEK 23613+256*PEEK
23614
110 POKE G,L-INT (L/256)*256
120 POKE G+1,INT (L/256)
130 POKE G+2,S
140 RETURN

200 PRINT "linha 200"
210 LET L=30: LET S=1
220 GO TO 100
```


Primeiro, a sub-rotina 100 vai buscar (PEEK) ERR SP para indicar a posição do primeiro item que está na pilha GOSUB. Depois, as linhas 110 e 120 introduzem, com POKE, um novo valor para o número de linha, e a linha 130 introduz, com POKE, um novo valor para o número da instrução. Desta forma LET L=x:LET S=y:GOTO 100 fará com que o RETURN seguinte transfira o controle para a instrução y da linha x. É fácil verificá-lo: a sub-rotina 200, acima dada, devolverá (RETURN) o controle à instrução 1 da linha 30 em vez de o devolver à instrução 1 da linha 20.

O CICLO FOR

A implementação do ciclo FOR do BASIC *ZX* é muito versátil e hábil. Difere das da maioria das versões de BASIC. O método habitual para se usarem ciclos FOR dentro uns dos outros é utilizar uma pilha FOR para armazenar os números de linha para os quais os comandos NEXT transferirão o controle (as chamadas linhas de ciclo). Armazenam-se numa pilha as linhas de ciclo pela mesma razão que se usa uma pilha GOSUB para armazenar os números de linha de retorno. Cada ciclo FOR introduz o número da sua linha de ciclo na pilha FOR, o que significa que um comando NEXT transferirá, sempre, o controle da linha de ciclo do último, ou mais interior, ciclo FOR. No entanto, o BASIC *ZX* não utiliza uma pilha FOR, dando como resultado comportar-se de maneira diferente da maioria das outras versões de BASIC.

Cada vez que se encontra uma instrução FOR, procura-se, na área das variáveis, aquela que tenha o mesmo nome da variável usada no índice. Se se encontra alguma, elimina-se. Depois cria-se uma nova variável, com o mesmo nome, de tipo 7, ou seja, de índice. O formato de uma variável de índice já foi dado neste capítulo, mas repete-se na figura 4.3. É de notar que contém toda a

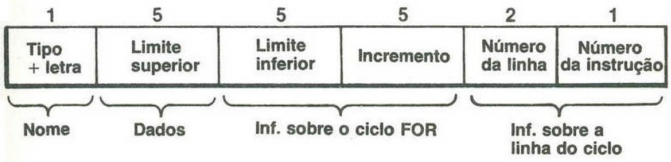


Fig. 4.3. *Formato da informação para variáveis indexadas em BASIC* ZX.

informação necessária à implementação do ciclo FOR, bem como o valor habitual de cinco *bytes* associado com cada variável numérica. O «limite» e o «incremento» são o valor final e o incremento do ciclo FOR, respectivamente. A linha de ciclo é armazenada como um número de linha de dois *bytes* e um número de instrução de um *byte*, o que define a instrução para a qual o comando NEXT (referente ao índice utilizado) transferirá o controle.

O único efeito real de uma instrução FOR é criar uma nova variável de índice. Todo o verdadeiro trabalho, num ciclo FOR, é feito pela instrução NEXT. Quando se encontra uma instrução NEXT adiciona-se o «incremento» ao «valor» e compara-se o resultado com o «limite». Se o resultado exceder o «limite», o ciclo terminará. Se assim não for, passa-se o controle à linha de ciclo. À parte da sua utilização numa instrução NEXT, a variável de índice pode manipular-se e utilizar-se como se fosse outra variável numérica qualquer. Assim, para terminar um ciclo FOR prematuramente, pode-se simplesmente introduzir na variável de índice um valor maior do que o «limite».

Há duas consequências importantes pelo facto do BASIC *ZX* não utilizar uma pilha FOR. Em primeiro lugar, e diversamente de muitas versões de BASIC, pode sair-se de um ciclo FOR antes de ser completado, sem nenhuns problemas. Se o fizermos num BASIC que use uma pilha FOR, a entrada que está na pilha nunca é eliminada, e por isso a pilha vai-se enchendo até, finalmente, dar uma mensagem de erro. A única penalidade em BASIC *ZX* é que se deixa a variável de índice na área, mas pode ser usada como uma variável numérica, e um novo ciclo FOR para o mesmo índice também tornará a utilizá-la. Embora não cause qualquer problema em BASIC *ZX* o saltar fora de um ciclo FOR, não se deve adquirir tal hábito, porque os programas serão mais difíceis de converter a outras versões de BASIC.

O segundo efeito de não se utilizar uma pilha FOR é notável. Esta pilha produz um efeito inesperado no que respeita a ciclos FOR encadeados uns nos outros, pois detecta automaticamente um erro nessa ligação e emite uma mensagem de erro. Em BASIC *ZX*, contudo, funcionam quase todos os tipos de ligação entre ciclos FOR. Por exemplo:

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 PRINT J,I
40 NEXT I
50 NEXT J
```

A maioria das versões de BASIC, e também a maioria dos programadores habituados a essas versões, rejeitaria este programa por estar incorrecto (as linhas 40 e 50 devem ser trocadas entre si para constituir uma ligação correcta entre os dois ciclos FOR apresentados). Se se executar este programa em BASIC *ZX* verificar-se-á que não só funciona, mas que também tem a sua utilidade! A tentativa de entender esta ligação anormal entre ciclos FOR deve levar-nos a evitar o seu uso: ela funciona porque cada uma das instruções NEXT, nas linhas 40 e 50, é executada sem qualquer relação com o resto do programa. Assim a linha 40 transfere o controle, dez vezes, para a linha 20 para valores de I de 1 a 10. De cada vez, a linha 20 cria a variável de índice J e iguala-a a 1. Depois o controle passa à linha 50 que faz com que o ciclo FOR respeitante a J (isto é, linhas 20 a 50) seja executado dez vezes consecutivas. Em cada uma dessas vezes, o NEXT I da linha 40 não faz repetir o ciclo FOR respeitante a I, porque o «valor» armazenado em I já excede o «limite». No entanto, aumenta o valor armazenado no índice, adicionando-lhe o «incremento». Compreende-se agora a sequência de números que estes dois ciclos FOR imprimem no *écran*!

CONCLUSAO

A matéria tratada neste capítulo ajuda a compreender o funcionamento interno do BASIC *ZX*. Muitos dos exemplos de programas dados não só ilustram as ideias subjacentes mas também formam a base de uma colecção de utilitários com interesse. Para pôr à prova a compreensão de BASIC *ZX* não há melhor maneira do que tentar melhorar e desenvolver alguns dos muitos projectos originados por aqueles exemplos. A maior parte do trabalho pode ser feito em BASIC *ZX*, mas quem estiver a aprender assembly do *Z80* encontrará muitos problemas não demasiado difíceis para resolver e que são uma recompensa.

CAPÍTULO 5

I/O – Canais e «streams»

O *Spectrum* tem um método muito geral e complexo para tratar de diferentes dispositivos de I/O, método esse que se baseia em *streams* (caudais) e canais. O *Spectrum* normal tem um número muito limitado de dispositivos de I/O; como consequência, bastam poucos comandos para fazer accionar cada um desses dispositivos. Por exemplo, para enviar informação ao *écran* utiliza-se a instrução PRINT, mas, para enviar dados à impressora *ZX*, emprega-se o comando LPRINT. Agregadas *microdrives* ao sistema, torna-se rapidamente inconveniente inventar comandos especiais. Mesmo sem as *microdrives* há vantagem na utilização do método geral do *Spectrum* para definir um dispositivo para uma operação I/O. Surpreendentemente, o manual do *Spectrum* não menciona, nem sequer sugere, o método de tratamento de I/O utilizando *streams* e canais.

«STREAMS» – INPUT E PRINT

O modo de visualizar o I/O é separá-lo em duas partes, uma correspondendo aos programas do sistema que recebem ou criam informação, e a outra correspondendo ao equipamento que recebe e cria informação. Em BASIC *ZX* a componente de I/O, relativa aos programas, é chamada *stream* (caudal) e a componente relativa ao equipamento tem o nome de «canal». A diferença principal é que o *stream* é um fluxo, incaracterístico, de informação que entra e sai de um programa, mas um canal corresponde a um dispositivo particular de I/O como a impressora *ZX*. Um *stream* é como um conjunto de itens de informação entrando e saindo de um dado equipamento. Os *streams* identificam-se por um número entre 0 e 15, e as suas operações básicas são a leitura e a escrita de informação. A instrução

```
INPUT # s; «lista de variáveis»
```

lerá informação do caudal «s» e armazená-la-á nas variáveis que constem da «lista de variáveis». Por exemplo


```
INPUT # 0; A;B;A$
```

lerá informação do *stream* 0 para as variáveis A, B e A$.

Do mesmo modo o comando

```
PRINT # s, «lista de variáveis»
```

enviará para o *stream* «s» a informação que estiver armazenada nas variáveis que constem na «lista de variáveis». Por exemplo

```
PRINT # 0; TOTAL; A$
```

enviará, para o *stream* 0, o conteúdo das variáveis TOTAL e A$. Note-se que tanto o INPUT# como PRINT# podem utilizar-se, exactamente, da mesma maneira que os comandos INPUT e PRINT. Qualquer item que se possa usar numa «lista de variáveis», quer de entrada quer de saída, pode também incluir-se em instruções de I/O em *streams*. Por exemplo, ambas as instruções seguintes

```
PRINT #2; «OLA»; TAB(10);«AMIGOS»
```

e

```
INPUT # 0;«ESCREVA O SEU NOME»;N$
```

são válidas. A instrução PRINT envia, ao *stream* 2, a mensagem «OLA», seguida dum código TAB, e depois a outra mensagem «AMIGOS». Note-se que a informação é enviada ao *stream* exactamente do mesmo modo como seria enviada ao *écran*. A instrução INPUT é um pouco mais complicada porque, além de pedir informação ao *stream* 0, ainda envia a mensagem «Escreva o seu nome». De facto, cada número de *stream* está associado com dois *streams* — um de entrada e outro de saída. A informação escrita no *stream*, seja por um PRINT ou por um INPUT, é enviada ao *stream* de saída, e a informação lida do *stream* obtém-se do *stream* de entrada.

Na prática, é possível utilizar instruções PRINT ou INPUT que referem mais do que um *stream*. Por exemplo

```
PRINT # 5; «Ola»; # 6;«amigos»
```

enviará a mensagem «Ola» para o *stream* 5 e «amigos» irá para o *stream* 6. Por outras palavras, pode incluir-se um «especificador de *stream*» #s, numa lista de saída ou entrada, sempre que seja necessário mudar de *streams*. No entanto, é melhor não trocar de

streams a meio de uma instrução, a não ser que haja razões especiais para isso. Os programas que utilizam vários *streams* numa só instrução de I/O são muito difíceis de entender, verificar e alterar.

CANAIS – OPEN E CLOSE

É bastante fácil compreender o conceito de *stream* de informação, mas como se associarão os números de *streams* com os dispositivos de I/O? Antes de qualquer informação poder ser enviada ou recebida de um *stream* este deve, obrigatoriamente, ser aberto, com OPEN. Abrir um *stream* tem dois propósitos: associar um número de *stream* com um dado dispositivo de I/O, e assinalar qual dispositivo de I/O irá ser utilizado. Além de assinalar o dispositivo a usar, a abertura de um *stream* envolve, muitas vezes, a inicialização desse dispositivo para o pôr num estado em que possa ser utilizado. No entanto, esta inicialização depende muito do próprio dispositivo. Para abrir, com OPEN, um *stream,* o BASIC *ZX* tem o comando

OPEN #s,c

onde «s» é o número do *stream* a abrir e «c» é uma variável de texto especificando o canal com o qual está a ser associado. Depois deste comando, o destino de toda e qualquer informação enviada ao *stream* «s» será o canal «c», que também será a origem de qualquer informação lida do *stream*. Antes de exemplificar o uso do comando OPEN, é necessário saber quais são os canais do *Spectrum*.

Um Spectrum sem *microdrives* (isto é, não expandido) reconhece somente três canais diferentes:

K — o canal do teclado
S — o canal do *écran* e
P — o canal da impressora

Assim

OPEN #5,«K»

abre o *stream* 5 e associa-o com o teclado. Depois desse comando, a entrada seguinte

INPUT #5

será equivalente à entrada normal de dados pelo teclado, com um comando INPUT. No entanto, o comando

PRINT #5;«OLA»

envia agora a informação para a parte de saída do *stream* 5, que está associada com a área de imagem do teclado, ao fundo do *écran*. Assim, a mensagem «OLA» será impressa na zona inferior do *écran,* reservada normalmente a mensagens de INPUT. Aquela mensagem não pode ser observada, porque a zona inferior do *écran* é limpa quando se interrompe a execução de um programa ou quando se atinge uma instrução INPUT. Para ver o efeito obtido pelo envio de informação à «zona de INPUT» do *écran,* ensaie-se:

```
10 OPEN #5,"K"
20 PRINT #5;RND
30 GO TO 20
```

Ver-se-ão números aleatórios impressos a partir do fundo do *écran* e desenvolvendo-se para cima. O programa terminará com uma mensagem de erro «OUT OF SCREEN» porque a zona de INPUT do *écran* não faz o *scroll* da mesma maneira que a zona habitual de impressão.

Embora, em princípio, todos os *streams* tenham uma zona de entrada e outra de saída, na prática somente o canal do teclado aceita entradas e saídas. Os outros dois — *écran* e impressora — são unicamente canais de saída, e resultará num erro J qualquer tentativa para ler informação de qualquer deles. É de notar que esta restrição é inteiramente uma característica do equipamento ao qual estiver associado o *stream*. Pode-se associar mais do que um *stream* a um dado canal, mas querendo mudar esse canal (com o qual está associado um *stream*) deve, obrigatoriamente, acabar-se com essa associação fechando-o, com CLOSE. O comando do BASIC *ZX*

CLOSE #s

terminará com qualquer associação existente entre o *stream* «s» e um canal. Neste sentido, o comando CLOSE é o oposto do comando OPEN. O encerramento de um *stream* pode também utilizar-se para informar a componente física de um canal que o *stream* já não necessita dele e que deve ser executada qualquer operação de «limpeza» para ficar pronto a ser utilizado por qualquer outro canal.

É importante notar que, enquanto um canal pode ser utilizado por vários *streams*, um *stream* só se associa com um único canal. Por exemplo, a impressora *ZX* pode estar associada aos canais 4 e 6; por isso

PRINT #4; «lista de variáveis»

e

PRINT #6; «lista de variáveis»

enviariam, ambas, informação à impressora, mas é impossível associar, por exemplo, o *stream* 7 tanto com a impressora como com o *écran*.

A UTILIZAÇÃO DE «STREAMS» – INDEPENDÊNCIA DO DISPOSITIVO

Até agora, a única vantagem da utilização de *streams* foi a possibilidade de enviar informação para a zona inferior do *écran*. Existe, no entanto, uma outra razão, e importante, para que o programa de BASIC *ZX*, num *Spectrum* não expandido, utilize *streams*. A noção de «independência do dispositivo» é habitualmente dada em cursos complementares de ciências de computação, mas é uma noção simples e muito útil. A independência dos dispositivos refere-se simplesmente à possibilidade de escrever um programa sem preocupar-se donde vem a informação necessária nem para onde ela vai depois de tratada. Por exemplo, pode escrever-se um programa que liste informação financeira sem se preocupar se a saída será no *écran* ou na impressora. O periférico onde sairá a informação será seleccionado posteriormente, pelo utilizador do programa. Se se utilizar PRINT e LPRINT para enviar informação para o *écran* ou para a impressora, respectivamente, não será fácil escrever programas independentes dos periféricos, mas utilizando *streams* e canais, sim, é bastante fácil!

Consideremos o problema de escrever um programa para imprimir uma lista de números aleatórios, ou no *écran* ou na impressora, à escolha. Com PRINT e LPRINT, o programa seria mais ou menos assim:

```
  10 INPUT "P-Impressora S-Ecran
";A$
  20 IF A$(1)="P" THEN LPRINT RN
D
  30 IF A$(1)="S" THEN PRINT RND
  40 GO TO 20
```

Utilizando *Streams* e canais, o programa seria:

```
  10 INPUT "P-Impressora S-Ecran
";A$
  20 OPEN #5,A$(1)
  30 PRINT #5;RND
  40 GO TO 30
```

O *stream* 5 associa-se ou à impressora ou ao *écran*, porque o especificador do canal pode ser uma variável de texto. Outra forma de conseguir o mesmo resultado seria abrir, com OPEN, dois *streams* diferentes, um para a impressora e outro para o *écran*. Como o especificador do *stream* pode ser uma expressão numérica que seleccione o *stream* a utilizar, o programa seria:

```
  10 INPUT "P-Impressora S-Ecran
";A$
  20 OPEN #5,"P"
  30 OPEN #6,"S"
  40 IF A$(1)="P" THEN LET S=5
  50 IF A$(1)="S" THEN LET S=6
  60 PRINT #S;RND
  70 GO TO 60
```

Embora este exemplo seja demasiado curto, reconhece-se certamente que, num programa extenso, é uma vantagem usar *streams* para agrupar todas as instruções semelhantes PRINT e INPUT. Fazendo-o, para mudar de periférico de saída basta mudar o comando OPEN apropriado ou o número do *stream* utilizado. Agregando *microdrives* ao sistema, não se pode evitar a utilização de *streams* e, por isso, faz sentido tirar deles, desde já, o benefício máximo.

OS «STREAMS» POR DEFEITO

O *Spectrum*, no procedimento de inicialização, abre automaticamente os quatro canais de 0 a 3 e associa-se aos seguintes canais:

stream	canal
0	K
1	K
2	S
3	P

Por isso, mesmo sem um comando OPEN, a instrução

PRINT #2; «OLA»

imprime a mensagem do *écran*. O *Spectrum* utiliza estes *streams* para dirigir os dados do programa para o periférico correcto. Por exemplo, um comando LPRINT envia informação ao canal 3. Estas atribuições de *streams* a canais podem alterar-se com comandos OPEN, mas os *streams* em si próprios não podem ser fechados. Tentando fechar, com CLOSE, um dos *stream* por defeito, este será reaberto e associado ao canal inicialmente dado pela tabela referida acima.

OUTROS COMANDOS DE «STREAMS»

Os outros dois únicos comandos de *streams* que podem utilizar-se com um *Spectrum* não expandido são LIST e INKEY$. O formato completo de LIST é

LIST #s,n

onde «s» é o número do *stream* para onde será listado o programa e «n» é o número de linha pelo qual começará a listagem. Por exemplo

LIST #1

listará um programa na zona inferior do *écran* reservada habitualmente para o INPUT, mas

LIST #3

é equivalente a LLIST.

O outro comando referido, INKEY$, pode utilizar-se para devolver-se um único carácter *(byte)* de um *stream* qualquer que esteja associado com um periférico que suporte entrada de dados. A função

INKEY$ #s

devolverá um só carácter do *stream* «s». Se não estiver disponível nenhum carácter do periférico de entrada de dados, será devolvida a mensagem nula. O único problema desta forma generalizada de INKEY$ é que um *Spectrum* não expandido tem um único canal para entrada de dados — o teclado. No entanto, uma vez adicionadas *microdrives,* o número de canais para entrada de dados

aumenta, e também aumenta o número de comandos, orientados para *streams*, que são úteis.

CANAIS E «STREAMS» – FORMATOS DA MEMÓRIA

Embora o programador de BASIC *ZX* não precise saber como se implementam canais e *streams* para fazer uso deles, existem outras formas para o programador do código máquina utilizar esse conhecimento. Em particular, o canal é a forma ideal para alargar a escolha de dispositivos de I/O sem ter que escrever código para comandos especiais de I/O.

A informação que define cada canal está armazenada na área de informação dos canais que começa em CHANS e acaba em PROG-2 (onde CHANS e PROG são variáveis de sistema). Cada canal tem um «registo do canal» próprio que tem o formato seguinte

endereço	comprimento	
n	2 *bytes*	endereço da rotina de saída
n+2	2 *bytes*	endereço da rotina de entrada
n+4	1 *byte*	letra-código do canal

onde as rotinas de entrada e saída são sub-rotinas em código máquina. A rotina de saída deve, obrigatoriamente, aceitar os códigos dos caracteres do *Spectrum* que lhe sejam passados no registo A. A rotina de entrada deve devolver a informação em códigos de caracteres do *Spectrum* e assinalar que há informação disponível levantando a etiqueta *carry* (em inglês *flag carry,* registo binário que pode tomar 2 valores, 1 ou 0, usado para assinalar uma determinada condição; o microprocessador *Z80* possui vários *flags* ou etiquetas «como zero» e «*carry*», que dão informação acerca do resultado das operações aritméticas e lógicas). Se não está disponível informação nesse momento, haverá que assinalar isso baixando tanto a etiqueta zero como a antes citada. Se o canal não suportar entrada de dados ou não suportar saída de dados, o endereço de rotina que executa a operação ilegal deve ser posto numa rotina de manuseamento de erros. A forma normalizada de tratar erros em BASIC *ZX* é através de uma chamada *Restart* ao endereço 8. O que, em linguagem assembly do *Z80,* resulta no seguinte:

```
ERROR   RST 0008
        DEFB coderro
```

onde «coderro» é o código numérico do erro que informará o utilizador. Os registos dos três canais de um *Spectrum* não expandido e de um outro canal ainda não descrito são os seguintes:

endereço	registo do canal do teclado
CHANS	endereço da rotina que imprime na zona inferior do *écran*
+2	endereço da rotina de entrada de dados pelo teclado
+4	«K», identificador do canal K
	registo do canal do *écran*
+5	endereço da rotina que imprime no *écran* (zona superior)
+7	endereço da rotina de erros
+9	«S» identificador do canal S
	registo do canal do *buffer* de edição
+10	endereço da rotina de entrada de dados no *buffer*
+12	endereço da rotina de erros
+14	«R» identificador do canal R
	registo do canal da impressora *ZX*
+15	endereço da rotina da impressora *ZX*
+17	endereço da rotina de erros
+19	«P» identificador do canal P

É de notar que todos os registos de canal descritos estão no formato atrás indicado, e que o único canal que suporta tanto entrada de dados como saída de dados é o canal K. O novo canal R é utilizado internamente pelo *Spectrum* para enviar informação ao *buffer* de edição. O comando OPEN não nos permite a associação de um *stream* com o canal R e, por isso, a sua utilidade é limitada.

Nota importante: quando se utilizam *microdrives* e a *Interface 1*, o formado de um registo de canal é diferente. Para tirar proveito da matéria tratada criando os nossos próprios registos de canais, e querendo que os nossos programas funcionem tanto no *Spectrum* expandido como no não expandido, deveremos consultar o capítulo 10.

A informação relativa à associação de *stream* com canais é armazenada na área das variáveis de sistema, a área com 38 *bytes* começando em STRMS (endereço 23568). A tabela de *streams*, e cada par de *bytes* desta tabela, guarda um número «x» que representa o endereço onde começa um registo de canal. Em vez de simplesmente guardar o endereço do registo de canal, «x» é a «distância» a que o registo do canal está do início da área contendo informação sobre canais:

endereço do começo do registo do canal = endereço da área de canais +x−1

Assim, cada elemento da tabela de *streams* é igual a um mais o número de posições de memória que vão desde o início da área com informação sobre canais até ao registo do canal. Como há um máximo de 16 *streams*, pensar-se-ia ser suficiente um máximo de 32 *bytes* (isto é, um endereço de canal por cada *stream*) para armazenar todas as associações entre canais e *streams*. Na realidade, utilizam-se os seis *bytes* adicionais para armazenar informação (relativa a canais) para três *streams* internos correspondentes aos números 255, 254 e 253. Estes três *streams* internos são associados automaticamente com os canais R, S e K, respectivamente, e, como só existem números de *streams* entre 0 e 15, não são acessíveis partindo do BASIC *ZX*. No entanto, a sua presença deve ter-se em conta ao tentar encontrar o endereço do registo de canais correspondente a qualquer dos *streams* de 0 a 15. Os primeiros três valores da tabela de *streams* dizem respeito aos *streams* internos de 253 a 255; o quarto dá o endereço do registo de canais a usar com o *stream* 0, e assim sucessivamente. Isto significa que o começo da tabela de *streams*, no que diz respeito a *streams* externos, é

STRMS+6 ou 23574

e o endereço do início do registo do canal associado com o *stream* s (s varia de 0 a 15) está armazenado em duas posições de memória começando em:

23574+s*2

Quando se abre, com OPEN, um *stream* a um canal particular, o comando OPEN armazena a diferença entre o início do registo desse canal e a área de canais propriamente dita, mais um, na posição correcta da tabela de *streams*. Quando se encerra um *stream*, com CLOSE, o elemento respectivo na tabela de *streams* é posto a zero. Deste modo, um elemento que seja zero na tabela de *streams* servirá para detectar uma tentativa de utilização de um *stream* que ainda não tenha sido aberto. São abertos automaticamente sete *streams:* os três *streams* internos e os *streams* de 0 a 3, como referimos anteriormente.

Este sistema de registos de canais e a tabela de *streams* foi generalizado, para ser utilizado pelas *microdrives*, mas as suas características essenciais mantêm-se. Descreve-se cada canal pelo seu registo e os *streams* associam-se com os registos através da tabela de *streams*. Antes de considerar a utilização de I/O com *streams* e canais, vale a pena mencionar a outra única variável de sistema que tem ligações a I/O através de canais — CURCHL. De cada vez que se utiliza um comando de I/O rellativo a *streams*, o número do *stream* serve para procurar, na tabela de *streams*, o endereço do registo de canais. Uma vez encontrado, este endereço é guardado na variável de sistema CURCHL para conduzir toda a informação gerada pelo comando de I/O ao canal correcto. Assim, depois de um comando tal como PRINT #s, CURCHL contém o endereço do início do registo de canais associado ao *stream* «s».

CRIEMOS OS NOSSOS PRÓPRIOS CANAIS

Quem tem um dispositivo especial de I/O ligado ao *Spectrum* ou planeia construir um novo dispositivo, já defrontou o problema de enviar e receber informação desse dispositivo. Sem dúvida que é, habitualmente, mais fácil construir uma *interface* para o ligar ao *Spectrum* do que escrever uma *interface*, em programação BASIC *ZX*. Utilizando a matéria que demos acerca de I/O relativo a *streams*, é comparativamente fácil estabelecer uma ligação entre dispositivos de I/O especiais e o BASIC *ZX*, de forma a serem considerados como parte integrante do equipamento *(hardware)* do *Spectrum*.

A maneira habitual de fornecer programação que manuseie dispositivos I/O especiais é escrever sub-rotinas BASIC que utilizem IN e OUT. Estas enviam e recebem dados directamente dos portos para I/O atribuídos ao dispositivo. Por exemplo, se se atribuir o porto 31 ao registo que controla a frequência de um gerador de som

OUT 31,f

enviará os dados (no intervalo de 0 a 255) ao emissor de som, estabelecendo assim a sua frequência. No entanto, se o dispositivo for «orientado para caracteres» — se recebe e envia dados na forma de caracteres — IN e OUT não são comandos adequados. Por exemplo, tanto uma impressora paralela como um *modem* (palavra que é uma condensação de *modulator-demodulator*, dispositivo destinado a enviar e/ou receber informação digital através de redes telefónicas; possibilita a comunicação entre computadores a longa distância) são dispositivos de manuseio de caracteres, e a melhor forma de obter resultados é utilizar as instruções PRINT e INPUT. Mesmo que se escrevessem rotinas adequadas utilizando OUT e IN para enviar e receber dados numéricos e texto, é difícil ver como se usariam para listar, com LIST, programas para os novos dispositivos. Torna-se portanto claro que deve programar-se *a interface* de dispositivos orientados para caracteres, utilizando *streams* e canais.

Há duas maneiras de adicionar um dispositivo orientado para caracteres ao sistema de *streams* e canais do BASIC *ZX:* ou mudando os endereços armazenados num registo de canais, existente, ou criando um registo de canais completamente novo.

O primeiro método obriga a introduzir, com POKE, novos endereços num registo de canais já existente que apontem às rotinas em código máquina feitas por nós e posicionadas em qualquer lugar da memória.

Por exemplo, supondo que se quer ligar ao *Spectrum* uma impressora profissional em vez da impressora *ZX*, mudar o endereço armazenado nas duas primeiras posições do registo do canal da impressora *ZX* (para apontar a rotina de controlo da impressora, escrita pelo utilizador) fará com que os comandos LPRINT e LLIST, bem como quaisquer comandos de I/O referentes aos *streams* abertos para o canal P, enviem a informação à nova impressora. Escrever a nova rotina de controle da impressora é fácil, em princípio. Só se terá de aceitar códigos de caracteres ASCII no registo A, e utilizá-los para imprimir os correspondentes

caracteres ASCII na impressora. No entanto, o conjunto de caracteres do *Spectrum* inclui muitos caracteres que o conjunto ASCII normalizado não possui, e estes terão de ser detectados e interpretados correctamente pela rotina de controle. Por exemplo, todos os itens que controlam a posição e os atributos numa instrução PRINT serão enviadas à rotina-guia sob a forma de códigos de controle, de acordo com a lista do Apêndice A do manual do *Spectrum*. Por exemplo, LPRINT TAB(10) envia os códigos ASCII 23, 10 e 0 à rotina-guia da impressora. O código 23 é o código de controle utilizado pelo *Spectrum* para TAB, e os dois códigos seguintes são o *byte* menos significativo e o *byte* mais significativo do parâmetro a utilizar na função TAB. Os códigos enviados às rotinas-guias de saída de dados são estudados, com mais pormenor, no próximo capítulo. É importante observar que toda a saída de dados do *Spectrum* é convertida numa sequência de caracteres e códigos ASCII antes de ser impressa no *écran*. Isto torna possível que a rotina-guia da impressora responda a todos os itens que controlam a posição e os atributos, ou os ignore, conforme seja necessário. Por exemplo, se a nova impressora for a cores, poderá mudar de cor de impressão em resposta a

```
LPRINT INK3;«Ola»
```

que enviará os códigos ASCII 16 (para INK) seguido do código 03 (para a cor 03) à rotina de controle da impressora.

O programa seguinte, exemplo deste método de estabelecer uma *interface* com um novo dispositivo de I/O, altera o endereço de saída armazenado no canal P de forma a apontar a uma rotina em código máquina guardada na área do *buffer* da impressora. Note-se que isto só funciona porque a impressora *ZX* não está a ser utilizada. A nova rotina de saída, em código máquina, não faz nada realmente útil com a informação recebida; somente a envia ao porto 254, de I/O, que controla o altifalante e a cor da moldura. Isto assegura, pelo menos, que os seus efeitos se podem ver e ouvir! Em linguagem assembly do *Z80*, esta rotina de controle é simples:

endereço	assembler	código	observações
23296	outdrv LD BC, 254	01,256,00	carrega o reg BC com 254
23299	OUT (C), A	237,121	envia o registo A ao porto 254
23301	RET	201	regressa ao interpretador de BASIC *ZX*

O seguinte programa em BASIC *ZX* utiliza-a:

```
 10 DATA 01,254,0,237,121,201
 20 FOR A=23296 TO 23301
 30 READ D
 40 POKE A,D
 50 NEXT A


100 GO SUB 1000
110 FOR I=0 TO 7
120 LPRINT I;
130 NEXT I
140 GO TO 110


1000 LET C=PEEK 23631+256*PEEK 2
3632
1010 LET C=C+15
1020 POKE C,23296-INT (23296/256
)*256
1030 POKE C+1,INT (23296/256)
1040 RETURN
```

A rotina de saída, em código máquina, está guardada na instrução DATA da linha 10, e é introduzida na memória pelas linhas 20 a 50 (23296 é o início do *buffer* da impressora *ZX*). A sub-rotina 1000 altera o endereço que se encontra no registo de canais, definindo-o como o canal P. A linha 1000 associa a C o endereço do início da área dos canais e, em seguida, a linha 1010 calcula o início do registo de canais relativo ao canal P. As linhas 1020 e 1030 introduzem, com POKE, o endereço da nova rotina de saída no primeiro par de posições do registo de canais. Se se introduzir e executar este programa ver-se-á que a moldura do *écran* ficará intermitente e mudará inesperada e aloucadamente. Se interrompermos a execução do programa, obteremos provas adicionais de que a nova rotina de saída está a enviar informação ao porto de

controle da moldura, listando (com LLIST) o programa para ela. (Nota: desligue-se a impressora *ZX* antes de executar este programa.)

A alteração dos endereços guardados em registos de canais existentes é um método fácil de agregar novos dispositivos, mas tem a desvantagem de retirar um dos dispositivos I/O existentes. Na prática é impossível alterar o canal K (o registo de canal do teclado) porque os seus endereços de I/O são armazenados novamente, de cada vez que se executa uma instrução INPUT. Por isso, só podem modificar-se permanentemente os registos dos canais S e P, e, como o canal S está sempre a ser utilizado, o único que se pode alterar é P. E quando quisermos utilizar um outro dispositivo de I/O ao mesmo tempo que a impressora *ZX?*

Para agregar um número qualquer de dispositivos de I/O, é necessário adicionar novos registos de canais. Querendo fazê-lo de uma maneira completamente geral, deveremos ter em conta o modo como a *microdrive* modifica o sistema de operação *streams*/canais. Estudaremos este assunto no capítulo 12. Parece muito fácil adicionar um novo registo de canais, mas existem alguns pequenos pormenores a considerar. Em primeiro lugar, é possível criar um registo de canais em qualquer parte da memória, e não somente na área de informação dos canais, mas se o registo se armazena acima da área de trabalho do INPUT (começando em WORKSP) a variável do sistema CURCHL (canal actual) será alterada, porque se adiciona memória à área de trabalho durante um comando INPUT. Isto significa que se perde a localização actual do registo de canais, e o *Spectrum* ficará descontrolado. No entanto, se o registo de canais for guardado abaixo da área de trabalho de INPUT, tudo funcionará correctamente. Na demonstração que se terá mais adiante, usa-se o *buffer* da impressora *ZX* para armazenar tanto o novo registo de canais como as novas rotinas de I/O. Numa aplicação real, o registo de canais seria adicionado à área de informação; os pormenores da sua realização também se dão no capítulo 12. Uma segunda dificuldade é que os comandos OPEN e CLOSE somente funcionarão com os registos de canais de K, S e P. Isto significa que, para além de fornecer novas rotinas de I/O e um novo registo de canais, devemos ainda fornecer uma sub-rotina para abrir o canal a qualquer *stream* e, se necessária, uma sub-rotina para o encerrar. Tomando em consideração tudo isto, apresentam-se as seguintes rotinas, em assembly do *Z80*, de I/O e do registo de canais:

endereço		assembler	código	observações
23296	chanrec	DEFB 0	5	*byte* menos significativo do endereço de saída.
23297		DEFB 91	91	*byte* mais significativo do endereço de saída.
23298		DEFB 11	11	*byte* menos significativo do endereço de entrada.
23299		DEFB 91	91	*byte* mais significativo do endereço de entrada.
23300		DEFB «E»	69	identificador do canal.
23301	outdrv	LDBC,254	01,254,00	carrega reg BC com 254
23304		OUT(C),A	237,121	envia conteúdo de A a 254
23306		RET	201	regressa ao código da ROM
23307	indrv	RST 8	207	*restart* de erro
23308		DEFB	18	código de erro por dispositivo ilegal.

Os primeiros cinco *bytes* formam o novo registo de canais. Em 23301 começa a rotina de saída que envia simplesmente o código contido no registo A para o porto de saída 254, que é o porto do altifalante e da cor da moldura. Em 23307 começa a rotina de entrada que informa simplesmente que há um erro, indicando que não é permitida entrada de dados para este canal. Obviamente que, numa aplicação prática, qualquer das rotinas poderia ser muito mais complicada. O seguinte programa BASIC utiliza este código máquina:

```
  10 DATA 0,91,11,91,69,1,254,0,
237,121,201,207,18
  20 FOR A=23296 TO 23308
  30 READ D
  40 POKE A,D
  50 NEXT A

 100 LET S=5: GO SUB 1000
 110 PRINT #5;RND;
 120 GO TO 110
```

```
1000 LET A=23574+2+5
1010 LET C=PEEK 23631+256*PEEK 2
3632
1020 LET R=23296-C+1
1030 POKE A,R-INT (R/256)*256
1040 POKE A+1,INT (R/256)
1050 RETURN
```

Nas linhas 10 e 50 carrega-se o novo registo de canais e rotinas de I/O no *buffer* da impressora *ZX*. A sub-rotina 1000 abre o *stream* «S» ao novo canal. Noutras palavras, equivale a pôr OPEN *S,«E». A linha 1000 calcula o endereço correcto do *stream* «S», na tabela de *streams*. As linhas 1010 e 1020 calculam os deslocamentos dos novos registos de canais em relação ao início da área de informação dos canais (mais um), e as linhas 1030 e 1040 armazenam-no na tabela de *streams*. A linha 100 utiliza a sub-rotina 1000 para abrir o *stream* 5 ao dispositivo E, e as linhas 110 e 120 dão uma demonstração do programa, enviando os códigos correspondentes a números aleatórios ao porto que controla o som e a moldura. Pode completar-se a demonstração interrompendo o programa e introduzindo pelo teclado o comando LIST #5. Isto produz um relâmpago de cor e som, indicando que o programa foi listado na porta 254! Se alterarmos a linha 110 para

```
110 INPUT #5;1
```

obteremos a mensagem de erro correspondente, indicando que este canal não serve para entrada de dados.

Não é difícil adicionar novos registos de canais ao BASIC *ZX*, a não ser por se ter de programar rotinas de controle de I/O mais completas e especializadas. É de notar, contudo, que, se estiverem ligadas *microdrives*, o programa acima referido não funcionará — mas a necessária alteração é fácil e já a descrevemos no capítulo 10.

Antes de terminar este capítulo, e depois da descrição dos problemas relacionados com a programação de uma rotina de controle de saída, vale a pena mencionar os restantes requisitos necessários à programação de uma rotina de controle de entrada. Se um canal de entrada vai fornecer um único código de carácter, o melhor comando BASIC a utilizar é INKEY$#, que devolverá um único carácter. Seria esse o caso se utilizássemos um conversor A/D de 8 *bits* (é um conversor analógico digital que converte valores de intensidade ou tensão de uma corrente em valores digitais manipuláveis pelo computador; a corrente variável pode representar uma variação de qualquer grandeza física, como intensidade de som, pressão sanguínea, intensidade luminosa, etc.). No entanto, se tivermos em mente utilizar INPUT# para ler todo um conjunto de códigos de caracteres, deveremos precaver-nos contra duas situações: em primeiro lugar, as instruções INPUT também fazem saída de informação quando imprimem mensagens de aviso, etc.; não basta fazer com que a rotina de saída devolva um erro: devemos tratar a informação, seja qual for a que enviemos à instrução INPUT, mesmo que seja para ignorá-la!; em segundo lugar, uma instrução INPUT# aceita a informação como se tivesse sido digitada, o que significa que, se utilizamos INPUT#s;i para ler o valor de uma variável i, a rotina de controle do dispositivo tem de enviar um conjunto de códigos ASCII correspondentes a dígitos e terminando com um código ENTER, como se fosse introduzido um número pelo teclado. Finalmente, deve notar-se que mesmo quando se lê informação de um dispositivo especial, o comando INPUT interpretará, correctamente, códigos de edição, eliminação, etc.! A melhor forma de sintetizar tudo isto é observar que um INPUT funciona sempre como se recebesse uma sequência de códigos de caracteres correspondentes às teclas que se primem no teclado.

CONCLUSÃO

O sistema de *streams* e canais do *Spectrum* é como um prémio surpresa para o programador de BASIC *ZX*. A sua utilização em programas tem a vantagem de os tornar independentes dos vários dispositivos e de aumentar a sua flexibilidade geral, sem desvantagens de nenhuma espécie. Para o programador de linguagem assembly do *Z80*, esses *streams* e canais são a forma ideal de programar *interfaces* com qualquer equipamento novo.

CAPÍTULO 6

O «écran» de vídeo

As componentes físicas que contituem o circuito de vídeo, no caso do *Spectrum*, já foram descritas no capítulo 2, tendo-se aclarado os seus princípios gerais e como se relacionam com o resto da máquina. Os métodos utilizados no *Spectrum* para gerar a imagem serão descritos com pormenor neste capítulo, acentuando-se a forma como interactua a parte de programação *(software)* com o equipamento propriamente dito *(hardware)*.

O *écran* de vídeo, no *Spectrum*, merece uma atenção especialmente cuidada porque nele se combinam várias características interessantes, obtendo-se um sistema flexível com requisitos de memória razoáveis. Esta flexibilidade resulta da utilização de um único modo de alta resolução para mostrar tanto o texto como gráficos. Isto, pelo menos em teoria, permite a união de texto e gráficos em qualquer zona do *écran*, mas na prática a programação própria do *Spectrum* restringe o posicionamento dos caracteres a um número predeterminado de posições. A poupança em memória consegue-se pela utilização de «atributos paralelos» para controlar as cores, o que, sem dúvida, poupa bastante memória e permite utilizar oito cores. A contrapartida desta economia é a restrição do número de cores que se podem usar em cada posição de um carácter. Surpreendentemente, os atributos paralelos funcionam bastante bem e encaixam-se naturalmente no modo como muitos programas de gráficos organizam as cores.

Embora a imagem do *Spectrum* mereça louvores, há ainda campo para melhoramentos. Afortunadamente, a maioria dos problemas encontra-se ao nível da programação, podendo portanto modificar-se para incluir quaisquer outras necessidades. No entanto, para que isto seja possível é preciso conhecer a fundo o seu funcionamento.

DO PRETO E BRANCO ÀS CORES

O tipo de *écran* mais simples é o preto e branco ou expositor de duas cores, porque basta um único algarismo binário (isto é, 0 ou 1) para indicar um dos dois possíveis estados em que está um elemento

qualquer. A estratégia gráfica mais simples associa uma cor com cada estado, por exemplo preto com 0 e branco com 1. Desta forma pode usar-se um modelo binário para representar as cores de um conjunto de pontos do *écran*. É de notar que cada algarismo binário do modelo controla a cor de um único ponto do *écran*. Esta correspondência entre algarismos binários e pontos do *écran* dá a este método de geração de gráficos o seu nome: "gráficos binários".

O *Spectrum* utiliza um método de gráficos binários de forma que cada um dos pontos que constituem o *écran* de 192 por 256 pontos é controlado por um único algarismo binário armazenado em alguma parte da memória. Note-se que, como cada posição de memória pode guardar oito algarismos binários, controla a cor de oito pontos do *écran*. A correspondência exacta entre as posições de memória e os grupos de oito pontos do *écran* será descrita ulteriormente.

Este método que utiliza um só algarismo binário para controlar a cor (preto ou branco) de um só ponto do *écran* deve ser modificado para incluir a utilização de mais do que duas cores. Isto é mais difícil do que parece. O método mais claro para associar mais do que um algarismo binário como cada ponto do *écran* passa rapidamente a utilizar grande quantidade de memória. Por exemplo, são necessários dois algarismos binários para escolher quatro cores, o que faz duplicar a quantidade de memória necessária. São necessários três algarismos binários para escolher oito cores, dezasseis cores necessitam de quatro algarismos binários e assim sucessivamente. Para produzir a imagem do *Spectrum* com oito cores necessitar-se-ia de 18 k de memória, tornando impossível o uso do *Spectrum* de 16 k. Além de utilizar grande quantidade de memória, este método generalizado de gráficos binários cria outros problemas: é muito difícil retirar informação da memória com suficiente rapidez para obter três algarismos binários por cada ponto do *écran*.

A solução adoptada no *Spectrum* baseia-se na observação de que a maioria dos *écrans* a cores só utiliza duas delas, em qualquer zona do *écran*. Por exemplo, um céu azul com nuvens brancas e um sol amarelo tem três cores, mas perto da nuvem só há azul e branco e perto do sol só existe azul e amarelo. No *Spectrum* os pontos do *écran* estão agrupados em quadrados de oito por oito, correspondentes às já familiares posições de 24 linhas com 32 caracteres. Dentro de cada posição de um carácter, cada ponto só pode ter uma

de duas cores possíveis — conhecidas, na terminologia do BASIC *ZX*, como a cor da tinta *(ink)* e fundo *(paper)*. E como no exemplo simples com duas cores, a escolha da cor da tinta e do fundo de um ponto é controlada por um só algarismo binário numa posição de memória. A flexibilidade adicional deste novo esquema resulta do facto de as cores da tinta e do fundo, dentro de cada posição de um carácter, serem controladas por uma única posição de memória, ou *byte* de atributos. O expositor a cores do *Spectrum* é um meio caminho entre uma imagem simples a duas cores e um verdadeiro expositor com várias cores. Cada ponto do *écran* corresponde a um único algarismo binário da memória que determina se é um ponto de tinta ou de fundo. A cor atribuída, num dado momento, a pontos de tinta e de fundo, dentro de uma dada posição de carácter, é determinada pelos valores armazenados no correspondente *byte* de atributos. São fáceis de apreciar as vantagens deste método de atributos paralelos para produzir uma imagem a cores. Poupa-se uma grande quantidade de memória usando um único *byte* de atributos para controlar as cores da tinta e do fundo dos 64 pontos de uma posição de carácter. No entanto, também é óbvia uma limitação — só podem estar presentes duas cores em cada posição de um carácter.

A MEMÓRIA DO «ÉCRAN»

Há duas áreas da RAM relacionadas com o *écran* de vídeo no *Spectrum* — o ficheiro de imagem, ou ficheiro *display*, entre 16384 e 22527, e o ficheiro de atributos, entre 22528 e 23295. Como é de esperar, o ficheiro do *écran* representa a região da memória onde se armazenam os algarismos binários que determinam se um ponto é da cor da tinta ou do fundo. Da mesma forma, o ficheiro de atributos é a área da memória onde estão os *bytes* de atributos. Este conhecimento é uma ajuda, mas para controlar directamente o *écran* temos de saber encontrar, exactamente, o algarismo binário que controla um ponto particular, ou o *byte* que controla uma posição de carácter particular. Torna-se indispensável uma fórmula que converta coordenadas do *écran* no endereço da posição de memória interessada. E, obviamente, haverá duas fórmulas, uma para o ficheiro do expositor e outra para o ficheiro de atributos.

O MAPA DO FICHEIRO DE IMAGEM

A combinação mais óbvia do ficheiro de imagem é a que utiliza a primeira posição de memória, 16384, para armazenar os primeiros oito pontos da fila superior, a segunda posição de memória para armazenar os oito pontos seguintes dessa fila e assim por diante. De facto assim é e em geral cada fila de 256 pontos é armazenada em 32 posições de memória consecutivas. No entanto, surgem complicações. As filas não se guardam na sequência mais evidente, isto é, primeiro a primeira fila, segundo a segunda fila, até à fila do fundo. Em vez desta ordem, as filas armazenam-se numa sequência que reflecte as 24 linhas de posições de caracteres. Depois da fila superior de pontos vem a fila superior da segunda linha de posições de caracteres, seguida da fila superior da terceira linha de posições de caracteres, e assim sucessivamente até à fila superior da oitava linha de posições de caracteres. Por outras palavras, armazenam-se primeiro as filas superiores das primeiras oito linhas de posição de caracteres. Segue-se armazenamento da segunda fila de pontos de cada uma das oito primeiras linhas de posições de caracteres, e depois a terceira fila, e assim por diante. Depois repete-se este modelo de armazenamento para as oito linhas seguintes de posições de caracteres, e finalmente para as últimas oito linhas de posições de caracteres. É de notar que esta distribuição divide o *écran* em três zonas de oito linhas cada uma. Portanto armazena-se cada zona na mesma sequência, em que as filas de pontos formam as linhas de posições de caracteres, ou seja, primeiro todas as primeiras filas,

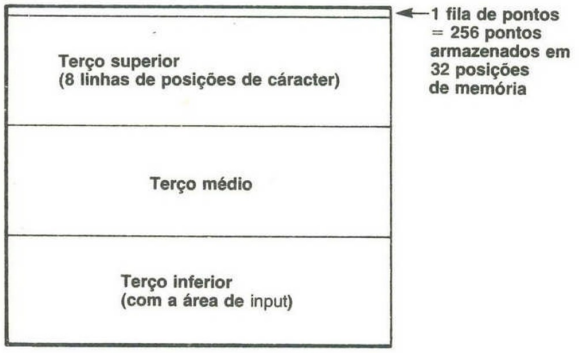


Fig. 6.1. *Organização da memória de* écran.

seguidas de todas as segundas filas e assim sucessivamente (ver figura 6.1). Este esquema de armazenamento compreende-se facilmente uma vez compreendida a sequência básica. Talvez a forma mais fácil de compreendê-lo seja observar o seguinte programa em acção:

```
10 FOR I=16384 TO 22527
20 POKE I,255
30 NEXT I
```

Este programa armazena 255 em cada uma das posições de memória do ficheiro do *écran*, uma de cada vez. Como 255 é 11111111, o programa mostra os pontos de tinta associados com os oito pontos controlados pela posição de memória. Desta forma, a posição desses pontos pode ser observada na sequência em que os pontos são mudados para a cor da tinta, sequência essa acima descrita.

Conhece-se agora a correspondência entre posições de memória e pontos, mas para que esta informação tenha alguma utilidade deve ser apresentada como uma fórmula que converta coordenadas do *écran* no endereço da posição da memória que as controla. Há duas maneiras de especificar a localização de um ponto no *écran*: pela posição de um carácter e por coordenadas gráficas. Por exemplo, querendo calcular o endereço da posição de memória que controla uma fila dada de oito pontos de uma dada posição de um carácter, se a posição deste está na linha L e coluna C, a posição de memória que controla a fila R é dada por:

$$16384+2048*INT(L/8)+32*(L-8*INT(L/8))+256*R+C$$

Para verificar se a fórmula está correcta, introduzamos o programa seguinte:

```
  10 DEF FN m(L,C,R)=16384+2048*
INT (L/8)+32*(L-8*INT (L/8))+256
*R+C
  20 CLS
  30 FOR N=0 TO 7
  40 FOR I=0 TO 31
  50 FOR J=0 TO 23
  60 POKE FN M(J,I,N),255
  70 NEXT J
  80 NEXT I
  90 NEXT N
```

Este programa enche o *écran* de cima para baixo e da esquerda para a direita.

Outra alternativa para especificar um ponto único: definir a posição pelas suas coordenadas x e y, o que resulta numa fórmula ainda mais complicada para calcular o endereço da posição de memória que controle esse ponto:

$$16384+32*(INT((175-Y)/8)-INT((175-Y)/64)*8 +8*(175-Y-INT((175-Y)/8)*8)+64*INT((175-Y)/64)) +INT(X/8)$$

O algarismo binário da posição de memória é dado por:

$$8-X+INT(X/8)*8$$

Esta fórmula pode parecer muito trabalhosa, e assim é quando escrita em BASIC. No entanto, as operações de divisão e exponenciação quadrática são fáceis de implementar em assembler do *Z80*. É mais fácil compreender a fórmula se for escrita usando as operações normalizadas

x DIV y significando INT(x/y)

e

x MOD y significando o resto da divisão de x por y, isto é, x−INT(x/y)*y

Utilizando estas operações e Z=175−Y, a equação aparece sob a seguinte forma

$$16384+32*((Z \text{ DIV } 8) \text{ MOD } 8+8*Z \text{ MOD } 8+64*Z \text{ DIV } 64) +X \text{ DIV } 8$$

Mesmo depois de todo este trabalho há que admitir que, além da sua utilização em assembler do *Z80*, estas fórmulas têm pouquíssimo valor simplesmente por demasiado complexas. No entanto, o conhecimento da estrutura geral do armazenamento do *écran* pode ser, sem dúvida, muito útil, como se ilustrará nos programas do capítulo seguinte.

O MAPA DO FICHEIRO DE ATRIBUTOS

A fórmula que dá a localização do *byte* de atributos (que controla

qualquer posição dada de um carácter) é muito simples. Um descanso depois de toda a complexidade do mapa do ficheiro de *écran*. Os *bytes* de atributos estão armazenados, a partir de 22528, na sequência natural de impressão das posições de carácter que controlam. Dito de outro modo, o primeiro *byte* de atributos controla a posição do carácter no canto superior esquerdo, o segundo controla a posição imediatamente à direita na mesma linha, e assim por diante até ao fim dessa linha. Este modelo é então repetido para cada linha até ao fim do *écran*. Para observar este esquema de armazenamento, ensaie-se o seguinte

```
10 FOR I=22527 TO 23295
20 POKE I,0
30 NEXT I
```

que armazena o código de atributo para fundo negro em cada *byte* de atributos, na sequência acima referida. Verificar-se-á que, diversamente do que sucede nos programas já apresentados e que manipulavam o ficheiro de *écran*, cada posição de memória introduzida, com POKE, altera uma posição completa de um carácter. A fórmula que dá o endereço do *byte* de atributos que controla a posição do carácter na linha L, coluna C, é:

22528+32*L+C

Esta fórmula é muito mais útil do que qualquer das obtidas no ficheiro de *écran*. Em particular, pode ser utilizada para modificar os atributos que controlam uma posição de carácter sem alterar o modelo dos pontos de tinta e de fundo no *écran*. Tente-se, por exemplo, o seguinte:

```
10 DEF FN a(C,L)=22528+32*L+C
20 PRINT AT 10,5;"isto e uma m
ensagem"
30 LET L=10
40 LET C=INT (RND*32)
50 LET A=INT (RND*256)
60 POKE FN a(C,L),A
70 GO TO 40
```

Este programa imprime primeiro uma mensagem no *écran*, e depois utiliza a função FNa para introduzir, com POKE na linha 60, códigos de atributos, aleatórios, nos *bytes* de atributos que controlam a linha onde se imprime a mensagem.

PESQUISA, COM PEEK, DO FICHEIRO DE IMAGEM – POINT E SCREEN$

Qualquer das duas fórmulas dadas antes serve para examinar o estado momentâneo de um algarismo binário contido no ficheiro de imagem *(display)*. Para isso basta pesquisar, com PEEK, a posição de memória correcta. No entanto, o cálculo de endereço é tão complexo que é sempre mais rápido utilizar a função POINT do BASIC *ZX*. Para implementar POINT (x,y), o BASIC *ZX* calcula o endereço da posição de memória que controla o ponto de coordenadas x,y, e depois devolve o valor do algarismo binário, contido nessa posição, controlando o ponto. Assim POINT (x,y) devolve 0 se o ponto tem a cor do fundo, ou 1 se tem a cor da tinta. É invulgar um programa cujo comportamento dependa do estado de um único ponto do *écran*, o que limita a utilidade da função POINT. Se precisarmos de verificar o estado de certo número de pontos, o uso repetido da função POINT tende a desacelerar a execução desse programa.

É habitualmente mais importante descobrir qual o carácter armazenado numa posição dada. Afortunadamente o BASIC *ZX* tem uma função para resolver este problema. A função SCREEN$ (linha, coluna) devolve o carácter que está no cruzamento da linha, coluna do *écran*. Consegue-se isto examinando cada um dos 64 pontos que constituem a posição do carácter em questão e comparando-os às definições de forma armazenadas na tabela de caracteres do *Spectrum*. Esta tabela descreve-se numa secção posterior, mas essencialmente é uma área contida na ROM do BASIC *ZX* que conserva o modelo dos pontos de tinta e fundo que constituem a forma de cada carácter. A função SCREEN$ é fácil de utilizar, mas importa estar atento a uma ou duas peculiaridades. Por exemplo, essa função, como funciona comparando modelos de pontos, contidos numa posição de carácter, com definições de caracteres, trata apenas a forma por que os pontos se agrupam, e não o modo porque se conseguiu essa forma. Assim, por exemplo, se se imprime a letra A, ou com PRINT «A» ou utilizando o comando PLOT para marcar pontos gráficos com a forma da letra A, a função SCREEN$ devolverá «A», em ambos os casos. Outra característica de SCREEN$ é que tanto verifica um carácter como o seu inverso. Isto significa que a «A» tanto será devolvida se a sua forma for feita por pontos de tinta, como se for feita de pontos de fundo. Assim, um bloco totalmente preenchido com pontos de tinta fará com que

SCREEN$ devolva um espaço em branco! Como última observação, SCREEN$ não reconhece caracteres definidos pelo utilizador que estejam impressos no *écran*.

CÓDIGOS DE ATRIBUTOS E ATTR

Num passo anterior, os códigos de atributos foram introduzidos, com POKE, no ficheiro de atributos, mas, a não ser que se saiba exactamente como é que o valor guardado no *byte* de atributos afecta a posição do carácter a que ele se refere, isto terá muito pouca utilização prática. Os oito algarismos binários que constituem um código de atributos tem a seguinte utilização:

b7	b6	b5 b4 b3	b2 b1 b0
f	b	fundo	tinta

onde f é a intermitência, b é o brilho e «fundo» e «tinta» são os códigos habituais das cores de 0 a 7. Por exemplo, se f for igual a um, a posição do carácter controlada pelo código de atributos tornar-se-á intermitente. Sabendo isto, e tendo em conta os pesos associados com cada algarismo um número binário (ver capítulo 1), resultará

128*f+64*b+8* fundo+tinta

cujo valor é o código de atributos que dá as cores desejadas à tinta e ao fundo e um carácter intermitente ou brilhante. Deste modo, se se quiser um carácter não intermitente (f=0), brilhante (b=1), escrito a tinta negra (tinta=0) em fundo branco (fundo=7) deverá introduzir-se o número 64+7*8, com POKE, no *byte* de atributos que controla a posição do carácter.

Assim como se pode modificar um código de atributos, alterando, com POKE, o conteúdo do ficheiro de atributos, também é possível pesquisar este ficheiro, com PEEK, para saber o valor, nesse momento, do código de atributos. E até é desnecessário calcular o endereço desse código, porque o BASIC *ZX* tem a função ATTR (linha, coluna) que devolve o valor armazenado na posição de memória que controla a posição do carácter localizado em linha, coluna. Não é difícil decompor o código de atributos nos seus elementos, por exemplo, qual a cor do papel nesse instante. Em seguida apresentam-se as funções que fazem essa decomposição:

```
DEF FNf(L.C)=INT(ATTR(L,C)/128)
DEF FNb(L,C)=INT(ATTR(L,C)-(FNf(L,C)*128)/64)
DEF FNp(L,C)=INT((ATTR(L,C)-INT(ATTR(L,C)/64)*64)/8)
DEF FNi(L,C)=ATTR(L,C)-INT(ATTR(L,C)/8)*8
```

onde FNf devolve o valor de f, FNb o de b, FNp devolve o valor do código do fundo e FNi devolve o da tinta.

A ROTINA DE CONTROLE DO VÍDEO

Descrevemos antes o funcionamento da imagem de vídeo do *Spectrum* relativamente à utilização e organização dos ficheiros de imagem e de atributos. O comando PRINT dá a possibilidade ao utilizador de não se preocupar com estas considerações. Na ROM, que contém o BASIC *ZX*, há rotinas em código máquina que examinam os elementos da informação dada numa instrução PRINT e que armazenam modelos de algarismos binários na RAM, reservada ao *écran*, para formar os caracteres que representam a informação. Por exemplo, PRINT «A» faz com que as rotinas em código máquina armazenem o modelo de pontos, referente à letra A, na posição onde o cursor de impressão estiver posicionado nesse instante. O processo de representação de uma variável numérica é um pouco mais complexo. Por exemplo, PRINT A faz com que as rotinas em código máquina convertam o número, armazenado em A, numa sequência de algarismos decimais, que se mostrarão, então, no *écran*. Recorde-se que o número armazenado em A, ou outra variável numérica qualquer, está armazenado em binário, tendo de ser convertido numa variável de texto contendo algarismos decimais antes de poder ser impresso. Neste sentido, PRINT A desencadeia as mesmas acções que PRINT STR$(A) (a função STR$ converte um valor numérico numa variável de texto contendo os seus algarismos).

A forma mais fácil de implementar as rotinas em código máquina que executam a instrução PRINT seria simplesmente escrever o que fosse necessário para imprimir no *écran* cada tipo diferente de dados. Afortunadamente, os autores da ROM do BASIC *ZX* pensaram antes de começar a programar! Como resultado, os programas que implementam a instrução PRINT estão divididos em duas partes, as «rotinas PRINT» e a rotina de controle do vídeo.

As rotinas PRINT são responsáveis pela conversão de cada elemento da informação contida numa instrução PRINT numa

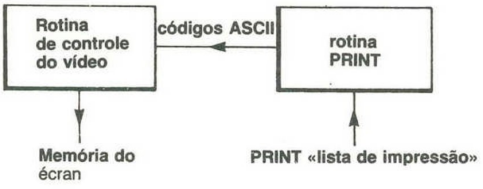


Fig. 6.2. *Duas rotinas distintas — a rotina PRINT e a rotina de controle do vídeo — transferem informação da RAM para a memória do* écran.

sequência de códigos ASCII dos caracteres. A rotina de controle do vídeo aceita estes códigos ASCII e é responsável pela produção de modelos de pontos no *écran* que representem cada um dos caracteres imprimíveis (ver figura 6.2). É esta separação em duas partes que torna o sistema de I/O do *Spectrum* tão flexível. Com a utilização de *streams* e canais (ver capítulo 5) é possível associar uma rotina de controle de cada dispositivo diferente com as rotinas PRINT e saber que essa rotina tem a manusear apenas uma sequência de códigos ASCII. Da mesma forma, é possível enviar, de outra origem, *streams* de códigos ASCII à rotina de controle do vídeo. Sem a referida separação seria extremamente difícil redireccionar a entrada/saída (I/O).

Por outro lado, a utilização de códigos ASCII como meio de comunicação entre a rotina PRINT e a rotina de controle do vídeo abre o caminho a novas técnicas de programação. Não somente aparecem caracteres imprimíveis, como letras e algarismos, numa instrução PRINT; também há os itens de controle, como TAB, AT, INK e PAPER. Mesmo estes itens não imprimíveis são convertidos em códigos ASCII pela rotina PRINT antes de serem transmitidos à rotina de controle do vídeo. Estes códigos ASCII são conhecidos como «códigos de controle» porque, embora nada imprimam no *écran*, controlam ou afectam o expositor. O código, ou códigos, seguintes a um de controle podem ser interpretados como «parâmetros» que governam o resultado produzido. Por exemplo, o item INK 7, não imprimível, é convertido pela rotina PRINT para o código 16 (associado a INK), seguido pelo código ASCII 07, representativo do código da cor. É de notar que o código da cor é

enviado à rotina de controle do vídeo como o código ASCII 07, e não o algarismo 7, ou seja, é enviado como CHR$(07) e não CHR$(55). Apresenta-se a seguir uma tabela de códigos de controle e respectivos parâmetros.

Item não imprimível e seu efeito	Código	Parâmetros
, (mover para a zona de impressão seguinte)	6	nenhum
cursor à esquerda	8	nenhum
cursor à direita	9	nenhum
ENTER	13	nenhum
INK c	16	c
PAPER c	17	c
FLASH f	18	f
BRIGHT b	19	b
INVERSE i	20	i
OVER o	21	o
AT y,x	22	y x
TAB x	23	x 0

É de notar que tanto AT como TAB são seguidos por dois parâmetros, embora TAB só utilize o primeiro. Há outros códigos de controle que não se apresentam nesta tabela e que são utilizados pelo editor do programa.

Como a rotina de controle do vídeo não recebe mais do que uma sequência de códigos ASCII da rotina PRINT, não importa como sejam gerados. Por exemplo

```
PRINT INK 6
```

e

```
PRINT CHR$(16);CHR$(6);
```

produzem ambas a mesma sequência de códigos ASCII, por isso têm o mesmo efeito! É possível, incluindo sequências de códigos de controle, fazer mensagens que se posicionem a si próprias, ou que especifiquem automaticamente as suas próprias cores, etc. Por exemplo, no programa seguinte

```
  10 LET M$=CHR$ (22)+CHR$ (10)+
CHR$ (9)+"Esta mensagem e sempre
 impressa no mesmo sitio"
  20 PRINT M$
  30 GO TO 20
```

a variável de texto M$ inclui os códigos de controle correspondentes a AT 10,5, por isso será sempre impressa na mesma posição, independentemente de onde o utilizador a quiser imprimir!

A rotina de controle do vídeo pode ser utilizada directamente pelo programador de linguagem assembly do *Z80* em todas as operações de impressão, para as quais o programador em BASIC *ZX* utiliza PRINT. Para ter acesso à rotina de controle do vídeo, tudo o que é necessário é um comando RST 16 depois de carregar o registo A com o código ASCII do carácter a ser impresso (ou com o código da operação a executar). Por exemplo, para imprimir a letra A poderia utilizar-se

```
LDA#65
RST 16
```

que carrega, em primeiro lugar, o registo A com o código ASCII correspondente a A, e depois repete a chamada (RST) à rotina de controle do vídeo. (Note-se que o código máquina para RST 16 é 215, decimal.)

Finalmente, e antes de considerarmos outras características da imagem do *Spectrum*, vale a pena realçar que a rotina de controle do vídeo não só modifica o ficheiro de imagem quando é impresso um carácter mas também armazena o código de atributos actual no correspondente *byte* de atributos. Comandos como INK e PAPER que aparecem fora de uma instrução PRINT usam a rotina de controle do vídeo. Os únicos comandos de imagem que não utilizam a rotina de controle do vídeo são o comando CLS e os comandos para gráficos de alta resolução PLOT, DRAW e CIRCLE.

AS TABELAS DE CARACTERES

Uma parte importante dos programas do *Spectrum* que geram texto é formada pelas duas tabelas de caracteres. A maior parte das definições de caracteres do *Spectrum* é mantida na tabela principal de caracteres, na ROM. Como a tabela está situada na

ROM, não é possível alterar qualquer das definições. No entanto, como a variável de sistema CHARS armazena o endereço do início desta tabela, é possível mover a tabela completa para RAM e, assim, dar ao *Spectrum* um conjunto de caracteres totalmente definíveis pelo utilizador! (ver capítulo 7). A segunda tabela de caracteres utiliza-se para armazenar as definições dos caracteres definíveis pelo utilizador, e como é de imaginar, está habitualmente armazenada na RAM. Contudo, o endereço inicial desta tabela também se guarda numa variável de sistema, UDG, e portanto esta tabela também pode ser movida para qualquer endereço desejado. O formato da informação utilizada para definir as formas de todos os caracteres do *Spectrum* é idêntica à utilizada no caso dos caracteres definíveis pelo utilizador. Isto é, os 64 pontos que constituem um carácter são armazenados em oito posições de memória. Cada posição de memória guarda oito algarismos binários que representam o estado dos oito pontos que formam uma fila do carácter. Tendo isto presente, não é difícil verificar que se o início da tabela principal de caracteres é START, as oito posições de memória contendo a definição de CHR$(I) são dadas por:

START+8*(I−32)

(O primeiro carácter imprimível é CHR$(32).) Esta fórmula pode utilizar-se para imprimir o modelo de pontos duma letra qualquer:

```
  10 DEF FN t()=256+PEEK (23606)
+256*PEEK (23607)
  20 INPUT C$
  30 LET I=CODE (C$(1))
  40 LET A=FN t()+8*(I-32)
  50 FOR K=A TO A+7
  60 LET D=PEEK (K)
  70 GO SUB 1000
  80 IF LEN (B$)<8 THEN LET B$="
0"+B$: GO TO 80
  90 PRINT B$
 100 NEXT K
 110 GO TO 20
1000 LET B$=""
1010 LET B=D-INT (D/2)*2
1020 IF B=0 THEN LET B$="0"+B$
1030 IF B=1 THEN LET B$="1"+B$
1040 LET D=INT (D/2)
1050 IF D=0 THEN RETURN
1060 GO TO 1010
```

A linha 10 dá a função FNt( ), que devolve o endereço do início da tabela principal de caracteres. (O endereço armazenado na variável de sistema CHARS é, realmente, 256 menos o endereço do início da tabela.) O resto do programa utiliza simplesmente PEEK para pesquisar as oito posições de memória que armazenam o modelo de pontos do carácter em C$(1), e imprime o modelo em binário.

O programa apresentado pode ser utilizado para descobrir a forma de qualquer dos caracteres definíveis pelo utilizador, se a linha 10 for modificada para a seguinte:

```
10 DEF FN t()=PEEK (23675)+256
*PEEK (23676)
```

que devolve o início da tabela de caracteres definíveis pelo utilizador, pesquisando, com PEEK, a variável do sistema UDG. Também a linha 40 deve ser modificada para:

```
40 LET A=FN t()+8*(I-144)
```

Há muitas utilizações directas das tabelas de definição dos caracteres, e algumas delas serão exploradas no capítulo seguinte. Os pontos principais a recordar são que se pode alterar qualquer dos endereços do início das duas tabelas e, se as tabelas forem armazenadas em RAM, podem modificar-se as definições dos caracteres utilizando POKE.

AS VARIÁVEIS DE SISTEMA DE IMAGEM

As variáveis de sistema de imagem tomam parte numa série grande de tarefas associadas com o *écran* de vídeo do *Spectrum*. As variáveis CHARS (23606) e UDG (23675), descritas antes, guardam, respectivamente, o início da tabela principal de caracteres e o início da tabela dos caracteres definíveis pelo utilizador.

Outras variáveis com interesse são:

CO ORDS (23677 e 23678)

Esta variável dá a coordenada x (em 23677) e a coordenada y (em 23678) do último ponto marcado num gráfico por um comando de alta resolução. Pesquisando, com PEEK, estas duas posições, pode

saber-se a posição que o cursor gráfico ocupa num dado instante. Por exemplo, se se desejar desenhar um segmento de recta desde a posição em que se encontra o cursor gráfico até à posição absoluta X, Y, utilizaremos

```
DRAW X-PEEK(23677),Y-PEEK(23678)
```

que funciona pela pesquisa, com PEEK, da posição actual do cursor de gráficos e calculando a diferença entre ele e a posição desejada.

S POSN (23688 e 23689)

Estas duas posições utilizam-se para armazenar a posição em que se encontre o cursor do texto, num dado momento. Para ser mais preciso, a posição do cursor do texto, num dado momento, é:

```
número de coluna=33-PEEK(23688)
número de linha=24-PEEK(23689)
```

DF CC (23684 e 23685)

DF CC guarda a mesma informação que S POSN — a posição actual do cursor do texto — mas, em vez do seu número de linha e coluna, DF CC guarda o endereço correspondente no ficheiro de imagem.

SCR CT (23692)

Esta variável de sistema guarda uma «contagem regressiva» da ocorrência seguinte da pergunta «scroll?». O seu valor é sempre maior em uma unidade do que o número de linhas a imprimir no *écran* antes que apareça a próxima pergunta «scroll?». Introduzindo repetidamente, e com POKE, o número 255 nesta variável de sistema, enquanto se executa o programa, assegurar-se-á que essa mensagem nunca apareça.

ATTR P e ATTR T (23693 e 23695)

Estas duas variáveis de sistema guardam, respectivamente, os códigos de atributos permanente e temporário. Dito de outro modo, se não houver comandos de atribuição numa instrução PRINT, o valor armazenado em ATTR P serve para atribuir os *bytes* de atributos correspondentes a cada uma das posições de carácter utilizadas. Contudo, se há um qualquer dos comandos de atribuição numa instrução PRINT, o valor de ATTR T é atribuído de forma

apropriada, e utilizado em lugar de ATTR P enquanto durar essa instrução PRINT.

MASK P e MASK T (23694 e 23696)
Estas duas variáveis de sistema guardam, respectivamente, os atributos transparentes permanente e temporário. Normalmente, sempre que se imprime um carácter, o código de atributos em ATTR P ou em ATTR T é armazenado no correspondente *byte* de atributos. Contudo, se for utilizado o atributo 8 em qualquer dos comandos de atribuição (por exemplo, INK 8), não é modificada aquela parte do *byte* de atributos.

MASK P e MASK T são utilizados para registar quais os atributos que estão, permanente ou temporariamente, postos em transparência. A codificação é tal que, sendo 1 qualquer algarismo binário em MASK P ou MASK T, isto indica que o algarismo binário correspondente será tomado do código de atributos existente e não de ATTR P ou ATTR T.

BORDCR (23624)
Esta variável de sistema guarda o código de atributos utilizado na metade inferior do *écran*. A parte relativa à tinta do código de atributos é também utilizada para definir a cor da moldura.

VÍDEO CRIATIVO

Embora neste capítulo se tenha explicado grande parte do funcionamento do *écran* de vídeo, um bom número das suas características não foi abordada explicitamente. A maior parte será evidente uma vez que se tenha compreendido o funcionamento global da imagem e no próximo capítulo apresentaremos alguns exemplos que utilizam o modo como funciona o *écran* de vídeo. Tenhamos em atenção que a melhor forma de compreender o *écran* de vídeo do *Spectrum* é seguir adiante e usá-lo com criatividade.

CAPÍTULO 7

Aplicações do vídeo

Este capítulo apresenta um certo número de exemplos das aplicações menos evidentes do *écran* de vídeo do *Spectrum*. Alguns desses exemplos podem formar a base de rotinas úteis em programas de aplicação. No entanto, o seu valor principal é superior ao que se pode conseguir com o *écran* do *Spectrum* sem modificar um único circuito interno!

CARACTERES FUNCIONAIS

Embora seja evidente que a tabela de caracteres definíveis pelo utilizador não é mais do que uma sequência de posições de memória como outra qualquer, há uma tendência para pensar que a sua alteração somente se pode fazer com a habitual instrução

POKE USR «carácter»+n,BIN modelo binário

que é tão conhecida que vale a pena examinar os seus elementos mais minuciosamente. O parâmetro da função USR é, normalmente, o endereço de um programa em código máquina (ao qual USR transfere o controle). No entanto, quando o parâmetro de USR é uma expressão literal, como USR «carácter», a função calcula o endereço da primeira posição de memória, na tabela de caracteres definíveis pelo utilizador, correspondente a «carácter». Reconheçamos que isto significa que USR «carácter»+n calcula o endereço da posição de memória que guarda o modelo binário da fila n do carácter defível pelo utilizador «carácter». Por exemplo, ensaiando

PRINT USR «A»

o *Spectrum* imprimirá o endereço da primeira posição de memória da tabela de caracteres definíveis pelo utilizador. O resto da instrução para definição do modelo de pontos é, agora, evidente. O POKE tem como resultado o armazenamento do modelo binário, que é o parâmetro de BIN, na posição de memória definindo a fila n do carácter definível pelo utilizador, «carácter».

Sem dúvida que, uma vez obtido o endereço inicial das oito posições de memória que controlam o modelo de pontos de um

carácter definível pelo utilizador, poderemos modificá-los como entendermos. Por exemplo, o seguinte programa

```
10 LET A=USR "b"
20 FOR I=0 TO 7
30 POKE A+I,INT (RND*256)
40 NEXT I
50 PRINT AT 10,10;CHR$ 145;
60 GO TO 20
```

produz um carácter definido aleatoriamente e em constante mutação. A linha 10 calcula o endereço inicial da posição de memória do carácter definível pelo utilizador, «b», ou CHR$ 145. As linhas de 20 a 40 introduzem, com POKE, valores aleatórios definindo cada uma das filas do carácter, e a linha 50 imprime, repetidamente, a nova forma aleatória. (Utiliza-se CHR$ 145 para evitar quaisquer ambiguidades no programa.) Esta definição funcional de um carácter aleatório pode generalizar-se e incluir caracteres novos que sejam funções simples de caracteres existentes. Por exemplo,

```
10 LET A=USR "a"
20 LET B=USR "b"
30 FOR I=0 TO 7
40 LET D=PEEK (A+I)
50 POKE B+7-I,D
60 NEXT I
70 PRINT CHR$ 144,CHR$ 145
```

define CHR$ 145 como o CHR$ 144 de pernas para o ar! A essência deste programa encontra-se nas linhas 40 e 50. A linha 40 pesquisa, com PEEK, a definição da fila I do CHR$ 144 e depois a linha 50 introduz essa definição na fila 7-I de CHR$ 145. Com médotos semelhantes definiremos caracteres que sejam imagens reflectidas e rotações de outros caracteres.

A ALTERAÇÃO DO CONJUNTO DE CARACTERES

Como a variável do sistema CHARS guarda o endereço do início (menos 256) da tabela principal de caracteres, é bastante fácil deslocar toda a tabela de caracteres para uma zona da RAM e então modificar uma ou todas as definições. Por exemplo, o programa

```
10 CLEAR 32768-1024
20 LET A=256+PEEK 23606+256*PE
EK 23607
30 LET B=32768-1024+1
40 FOR I=0 TO 95
50 FOR J=0 TO 7
60 LET D=PEEK (A+I*8+J)
70 POKE B+I*8+7-J,D
80 NEXT J
90 NEXT I
100 POKE 23607,INT ((B-256)/256
)
110 POKE 23606,(B-256)-INT ((B-
256)/256)*256
```

transfere toda a tabela principal de caracteres para a RAM mudando, ao mesmo tempo, a ordem de todas as filas de pontos, o que equivale a inverter todos os caracteres. Devemos reconhecer as componentes deste programa. A linha 10 reserva 1 k de memória, mais que suficiente para o conjunto de caracteres. Quem possuir um *Spectrum* de 48 k deverá alterar 32768 para 2*32768. A linha 20 calcula a posição da tabela de caracteres na ROM, e a linha 30 armazena essa nova posição em B. As linhas 40 a 90 deslocam a tabela de caracteres e o leitor reconhecerá que as linhas 60 e 70 são muito parecidas com as utilizadas na inversão de caracteres definíveis pelo utilizador, nas explicações anteriores. Finalmente, as linhas 100 e 110 introduzem, com POKE, o novo valor do endereço inicial da tabela de caracteres na variável do sistema CHARS.

É boa ideia gravar, com SAVE, este programa antes de executá-lo: será talvez um pouco difícil fazer-lhe quaisquer alterações estando todos os caracteres invertidos. E ao executar o programa segunda vez não se pense que o conjunto de caracteres volta à sua forma original. Não, em vez disso, ainda os complica mais!

ANIMAÇÃO INTERNA

Embora cada uma das tabelas de caracteres esteja organizada em grupos de oito posições de memória que correspondem à forma de um único carácter, há casos em que vale a pena considerar a tabela como um todo.

Por exemplo, se a tabela de caracteres definíveis pelo utilizador for preparada de tal forma que cada carácter seja uma letra de uma mensagem, esta mensagem poderá ser impressa num desenrolar suave imprimindo repetidamente o primeiro carácter definível pelo

utilizador e movendo o endereço inicial da própria tabela. Por exemplo, se a variável de sistema UDG contiver o habitual endereço da primeira posição de memória em que se encontra a tabela de caracteres definíveis pelo utilizador, PRINT CHR$ 144 imprimirá o primeiro carácter definido pelo utilizador. No entanto, se o valor de UDG é incrementado em uma unidade, PRINT CHR$ 144 imprimirá as últimas sete filas do primeiro carácter definido pelo utilizador e a primeira fila do segundo. Incrementando constantemente o valor de UDG, pode fazer-se mover as oito posições de memória, que definem a forma do primeiro carácter definido pelo utilizador, através da tabela original de um modo semelhante a um visor:

```
  10 GO SUB 1000
  20 FOR I=0 TO 20*8
  30 PRINT AT 10,10;CHR$ 144;
  40 POKE UDG,D+I-INT ((D+I)/256
)*256
  50 POKE UDG+1,INT ((D+I)/256)
  60 NEXT I
  70 GO TO 20

1000 LET UDG=23675
1010 LET D=PEEK UDG+256*PEEK (UD
G+1)
1020 RETURN
```

Este programa produz uma mensagem que se desenrola suavemente e que consiste nos caracteres, por defeito, definíveis pelo utilizador, ou seja, os caracteres de A a U. A sub-rotina 1000 armazena o início da tabela de caracteres definíveis pelo utilizador em D. O ciclo FOR, de 20 até 60, imprime o primeiro carácter definido pelo utilizador e depois desloca o início da tabela para a posição de memória seguinte.

Esta técnica de deslocar o início de uma tabela de caracteres serve para produzir animação interna notavelmente suave, utilizando somente o BASIC *ZX*.

CARACTERES LIVRES

Neste título, a palavra «livres» refere-se à posição dos caracteres e não à descoberta de quaisquer novos caracteres! Sabemos agora como é que o ficheiro de imagem controla o modelo de pontos no *écran*, por isso reconheceremos que a restrição de caracteres às suas posições é mais uma conveniência do que uma necessidade. Na realidade, a única razão para não permitir a colocação de caracteres em qualquer ponto especificado em coordenadas de alta resolução é que os *bytes* de atributos controlam toda uma posição de caracteres. Se não nos preocuparmos com a falta de equivalência entre a área ocupada pelo carácter e a área controlada por um *byte* de atributos, será fácil imprimir caracteres em qualquer posição do *écran*. Por exemplo, o programa seguinte imprime um «x» da maneira habitual e depois traça o número 2 como um expoente, produzindo a conhecida notação de x ao quadrado:

```
  10 PRINT AT 10,10;"X"
  20 LET X=95
  30 LET Y=99
  40 LET C$="2"
  50 GO SUB 5000
  60 STOP

5000 LET A=256*PEEK 23606+256*PE
EK 23607
5010 LET A=A+8*(CODE C$-32)
5020 FOR I=0 TO 7
5030 LET D=PEEK (A+I)
5040 FOR J=0 TO 7
5050 LET B=D-INT (D/2)*2
5060 LET D=INT (D/2)
5070 IF B=1 THEN PLOT X,Y
5080 LET X=X-1
5090 NEXT J
5100 LET Y=Y-1
5110 LET X=X+8
5120 NEXT I
5130 RETURN
```

O verdadeiro trabalho é levado a cabo pela sub-rotina 5000, que traçará o carácter guardado em C$, na posição de coordenadas X e Y. (X e Y são as coordenadas do canto superior direito do quadrado de 8×8 de pontos que formam o carácter.) O princípio em que isto se baseia é simples. Obtêm-se os *bytes* que definem cada fila do carácter em questão, um após o outro. Cada *byte* é decomposto nos seus modelos binários elementares de zeros e uns pelo ciclo FOR, mais interior, nas linhas de 5040 a 5090, e marca-se um ponto de tinta, com PLOT, se o algarismo binário armazenado em B for 1. O resto da sub-rotina trata de deslocar o valor guardado em X e Y na forma do quadrado de 8×8 pontos.

Pode obter-se um outro exemplo de aplicação desta sub-rotina mudando o programa principal para o seguinte:

```
10 LET A$="UMA MENSAGEM"
20 LET X=10
30 LET Y=170
40 FOR K=1 TO LEN (A$)
50 LET C$=A$(K)
60 GO SUB 5000
70 LET X=X+4
80 NEXT K
90 STOP
```

Aqui, utiliza-se a sub-rotina 5000 para imprimir a mensagem, armazenada em A$, diagonalmente no *écran*. A única informação adicional necessária para compreender o funcionamento deste programa é que, depois de ser executada a sub-rotina 5000, as variáveis X e Y contêm as coordenadas do canto inferior direito do carácter que se acaba de desenhar.

CARACTERES DE TAMANHO VARIÁVEL

Basta uma pequena alteração ao método utilizado para desenhar caracteres num ponto qualquer para se conseguir traçar caracteres de qualquer tamanho e em qualquer posição! O princípio básico é marcar, com PLOT, mais do que um ponto por cada algarismo binário pertencente à definição do carácter. Fazendo as alterações seguintes ao último programa apresentado, obter-se-á uma série de caracteres de diferente tamanho:

```
10 LET A$="ABCDE"
20 LET X=30
35 LET SX=5: LET SY=5
65 LET X=INT (X+SX/2)
66 LET SX=SX-1
67 LET SY=SY-1


5070 IF B=1 THEN GO SUB 6000
5080 LET X=X-SX
5100 LET Y=Y-SY
5110 LET X=X+SX+8
```



```
6000 FOR M=1 TO SY
6010 FOR N=1 TO SX
6020 PLOT X+N,Y+M
6030 NEXT N
6040 NEXT M
6050 RETURN
```

A sub-rotina 6000 desenha, com pontos, um quadrado ou um rectângulo de lados SX e SY, sendo portanto SX e SY as escalas de X e Y, respectivamente. As modificações introduzidas no programa principal acarretam que a chamada à sub-rotina 5000 seja feita com um par decrescente de escalas para desenhar uma diagonal constituída por caracteres, cada um menor que o anterior.

A ANIMAÇÃO NO «ÉCRAN»

Uma das tarefas realizadas pela programação interna do vídeo, no *Spectrum*, é o desenrolar *(scroll)* vertical do *écran*. Esta operação, simples na aparência, é na realidade muito mais complexa do que se possa imaginar. Basicamente, tudo o que essa programação realiza é mover cada grupo de oito filas de pontos, que formam uma linha de caracteres, para outra fila imediatamente acima daquela. Além de mover os pontos no ficheiro do *écran*, a programação responsável por este desenrolar *(scroll)*, deve também deslocar os *bytes* de atributos pelo equivalente a uma linha de texto no ficheiro de atributos. Relembrando o complexo esquema do ficheiro de *écran*, reconheceremos as dificuldades postas ao deslocar a informação para conseguir o mencionado desenrolamento. Como o ficheiro do *écran* está armazenado em três secções diferentes, cada uma constituída por oito linhas de caracteres, a verdadeira dificuldade resulta da obrigação de deslocar a linha superior de cada secção para a linha inferior da secção de armazenamento seguinte. (Ver o capítulo 6, para relembrar o esquema do ficheiro do *écran*.) Resumindo, o desenrolar vertical é suficientemente difícil para ser deixado à programação interna do *Spectrum*. No entanto, o desenrolar horizontal é muito mais fácil.

Há muitos programas de aplicação, especialmente jogos, que envolvem o movimento de gráficos ou texto para a esquerda ou para a direita. Por exemplo, um programa típico de jogos é capaz de produzir o efeito de uma nave de ataque voando sobre uma paisagem qualquer, mantendo a posição da nave fixa e «desenrolando» a paisagem horizontalmente. Não é difícil fazer o desenro-

lar horizontal de um ponto (movendo o *écran,* ou uma área dele, para a esquerda ou para a direita e por um ponto), mas é necessário utilizar linguagem assembly do *Z80.*

Para desenrolar o *écran* para a direita e por um ponto só é preciso começar no lado esquerdo de cada fila de pontos e deslocá-los um ponto para a direita. O ponto «final» da fila perde-se, e desloca-se um ponto de fundo para a primeira posição da fila. Desde que se comece pelo princípio, o ficheiro de *écran* está organizado de forma a que cada grupo de 32 *bytes* guarde o modelo de pontos de uma fila completa. Isto significa que o deslocamento de uma fila pode conseguir-se deslocando o conteúdo dos 32 *bytes* «para cima» e por um algarismo binário. Isto é, o conteúdo da primeira posição de memória do ficheiro de *écran* é deslocado um algarismo binário para a direita, de forma a que b1 se converte em b0, b2 em b1 e assim sucessivamente. Tem que ser dado o valor zero ao novo valor de b7, e o valor de b0 tem que ser guardado de modo a poder ser deslocado para b7, da posição de memória seguinte. Repete-se esta operação na segunda posição de memória, e assim por diante até à última posição, armazenando a fila de pontos. Em cada operação, todos os algarismos binários da posição de memória são deslocados um lugar à direita e b0, da posição de memória anterior, é movido para dentro de b7. Esta operação, numa única posição de memória, e em linguagem assembly do *Z80* chama-se «rotação à direita». Por isso o deslocamento do *écran* um algarismo binário à esquerda é equivalente a aplicar a operação de rotação à direita, repetidamente, a cada uma das 32 posições de memória que armazenam o modelo de uma fila de pontos.

A seguir apresenta-se uma rotina, em linguagem assembly, que desloca o terço superior do *écran* (linhas de texto de 0 a 7) um ponto para a direita:

endereço	assembler	código	observações
23296	LD HL, 16384	33,0,64	carregar registo HL
23299	LDA 63	62,63	carregar A com 63
23301	LOOP1 LDB 32	6,32	carregar B com 32
23303	ANDA	167	limpar a flag C
23304	LOOP2 RR (HL)	203,30	rotação à direita de A, de 1 algarismo binário
23306	INC HL	35	adicionar 1 a HL
23307	DJNZ LOOP2	16,251	B=B−1 e reciclar se B<>0
23309	DECA	61	A=A−1
23310	JR NZ, LOOP1	32,245	reciclar se A<>0
23312	RET	201	regressar ao BASIC

Pode carregar-se esta rotina para o *buffer* da impressora e chamá-la utilizando USR 23296, cada vez que se deseje deslocar o *écran* de um algarismo binário. Na segunda linha, LDA63 inicializa o número de filas de pontos a deslocar, neste caso 63 mais um, ou seja, 64. Embora a rotina tenha sido introduzida à mão como se fosse para ser executada em 23296, ela é, de facto, independente dessa posição e pode ser carregada em qualquer posição da memória. Demonstra-se a utilização da rotina no seguinte programa em BASIC:

```
 10 DATA 33,0,64,62,63,6,32,167
,203,30,35,16,251,61,32,245,201
 20 FOR I=23296 TO 23312
 30 READ D
 40 POKE I,D
 50 NEXT I

 60 PRINT AT 7,0;"ABCDE"
 70 PRINT AT 8,0;"ABCDE"
 80 LET A=USR 23296
 90 GO TO 80
```

As linhas de 10 a 50 carregam o código máquina no *buffer* da impressora. As linhas de 60 a 90 imprimem algo no *écran,* utilizando, em seguida, a rotina para deslocar a mensagem da linha 7 suavemente, até sair do *écran.*

Para exemplificar a sua utilização num jogo, ensaiem-se as seguintes substituições no programa anterior.

```
 60 LET Y=120
 70 LET S=1
 80 IF RND<.2 THEN LET S=-1*S
 90 IF Y=115 THEN LET S=1
100 IF Y=174 THEN LET S=-1
110 LET Y=Y+S
120 PLOT 0,Y
```

```
130 PRINT AT 2,10;"*";
140 LET A=USR 23296
150 GO TO 80
```

Este novo exemplo imprime um asterisco numa posição fixa e desenha uma «paisagem» que se desloca ao longo do *écran*. Este deslocamento cria a ilusão de um asterisco «a voar» sobre a paisagem, de uma forma impossível de conseguir utilizando somente o BASIC *ZX*.

CONCLUSÃO

Os exemplos apresentados neste capítulo foram todos relativamente curtos, o que facilitou a sua introdução e ensaio. No entanto, também foram suficientemente elaborados para ilustrar algumas ideias e torná-los merecedores de serem incluídos nos nossos programas. Por exemplo, o programa de deslocamento horizontal converte-se facilmente num jogo de acção de boa qualidade, sem razão para utilizar mais linguagem assembly do que a da rotina USR apresentada. Por outro lado, se não estivermos interessados em jogos, aquela rotina servir-nos-á para traçar gráficos móveis, a simular um osciloscópio. A computação, apesar de tudo que ouvimos dizer, é um assunto experimental, e as experiências perdem muito do seu valor se nos limitarmos a aceitar o que já sabemos que vai acontecer! Por isso é importante incorporar estes exemplos nos nossos programas e fazer experiências com eles.

CAPÍTULO 8

Gravação, som e a impressora

A *interface* de gravação, no *Spectrum*, e o seu limitado emissor de som utilizam os mesmos componentes físicos, dentro do ULA. No entanto, o factor essencial que une os três assuntos deste capítulo — a *interface* de gravação, o emissor de som e a impressora *ZX* — é a presença de programação comum, contida na ROM do BASIC *ZX*, que os controla a todos. Diz-se, sem grande precisão, que os três aparecem sob o título de dispositivos de I/O *standard*. À parte estas ligeiras relações não existe muito mais que os relacione, e por isso este capítulo está dividido em três secções principais: o sistema de gravação e leitura, o sistema de som e a impressora *ZX*.

O SISTEMA DE GRAVAÇÃO E LEITURA

Uma das melhores características do *Spectrum* é o seu sistema de gravação e leitura, notavelmente seguro. E não é, de nenhum modo, complexo: parece que a raiz dessa segurança é a atenção dispensada aos pormenores. Não se podem fazer grandes alterações à forma como funciona o sistema ou adicionar outras opções sem ter que escrever programas excessivamente longos, em linguagem assembly. Com isto não se quer dizer que não haja pontos a alterar, ou mesmo a melhorar, mas, sim, que nada justifica esse esforço. Quem estiver a planear qualquer modificação do sistema de gravação e leitura, o que necessita de saber depende muito do que tiver em vista. Por exemplo, estando-se interessado em escrever nova programação referente ao sistema de gravação do *Spectrum*, ou desejando ler *cassettes* produzidas por outros computadores, necessitar-se-á de conhecer a forma por que estão organizadas as componentes físicas. Se o projecto se referir à leitura de *cassettes* produzidas pelo *Spectrum* utilizando outras máquinas, é importante conhecer o formato utilizado ao escrever a informação. Por outro lado, quem estiver a escrever programas de aplicação em linguagem assembly do *Z80* é primordial que conheça as rotinas de código máquina que executam a leitura e gravação de ficheiros de fita magnética. Para abranger tudo isto, dividiu-se a descrição do

sistema de gravação em três pontos: componentes físicas, formato da informação e pormenores da programação própria do sistema.

COMPONENTES FÍSICOS DO SISTEMA DE GRAVAÇÃO

As características principais dos componentes físicos relacionados com a gravação já foram descritas no capítulo 2, mas sem explicar como se utilizam para armazenar informação na *cassette*. Tanto a linha que envia informação à *cassete* (MIC), como a que recebe informação da *cassete* (EAR) estão ligadas ao mesmo terminal (28) do ULA. Este terminal do ULA responde ao porto de I/O cujo endereço é 254, como já referimos no capítulo 2. O envio de informação para o porto 254, de I/O, põe a linha MIC, da *cassette*, em dois estados dependentes da condição em que estiver b3. Se b3 é igual a 0, a voltagem de saída é 0,75 volts. Se b3 é igual a 1, a voltagem de saída é 1,3 volts. Escrevendo, alternadamente, um 0 e um 1 no algarismo binário b3 da porta 254 possibilita-se o envio de uma onda quadrada e gravada como um tom de altura fixa de frequência dependente do tempo em que b3 permanece constante.

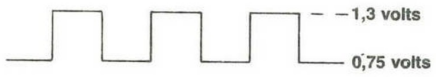


Fig. 8.1. *A onda quadrada, enviada para a linha MIC do gravador de* cassettes *através do porto 254, é obtida pondo b3 a 0 e a 1, alternadamente.*

Quanto mais longo for o intervalo de tempo entre mudanças de estado *(flips)* de b3, mais baixo é o tom do som. Por exemplo, o seguinte programa em BASIC

```
10 OUT 254,0
20 OUT 254,8
30 GO TO 10
```

altera o algarismo binário em b3 do porto 254 de 0 para 1, e a onda quadrada resultante pode ser gravada carregando nos botões *play* e *record* do modo habitual. Note-se ainda que a cor da moldura muda para negra enquanto se executa este programa, porque b0, b1 e b2 do porto de saída 254 controlam esta cor, e ambas as instruções

OUT põem esses algarismos binários a 0. A frequência deste tom é muito baixa, porque o BASIC *ZX* não é suficientemente rápido para mudar a condição do *bit* as vezes necessárias para produzir uma frequência alta. No entanto, utilizando assembler do *Z80* pode-se consegui-lo de modo a alterar a condição de b3 tantas vezes e em tão pouco tempo, que produz frequências acima do intervalo de audição humana! As rotinas de gravação e leitura da ROM do *Spectrum* utilizam este método simples para produzir um certo número de tons que são enviados, como informação, ao gravador de *cassettes*.

Se se ler uma *cassette* gravada com tons audíveis e desde que o gravador esteja ligado ao *Spectrum*, o nível da entrada determina a condição de b6 do porto de entrada 254. Na realidade, o nível da entrada determina a condição de b6 de qualquer porto de entrada correspondente a um endereço em que b0 esteja posto a 0 (ver capítulo 2). Se a voltagem da linha de entrada (EAR) no *Spectrum* for baixa, b6 é 0; se a voltagem for alta, b6 é 1. (Na prática, antes de ler a linha de entrada, o *Spectrum* eleva a voltagem da linha de saída controlada por b3, e por isso o sinal do gravador baixa a de entrada que, doutro modo, se encontra usualmente alta.) Para observar a forma em que é afectado b6, da porta de entrada, pelo sinal do gravador, experimente o leitor:

```
10 OUT 254,8
20 PRINT IN 254
30 POKE 23692,255
40 GO TO 20
```

A linha 10 eleva a saída para MIC antes de a linha 20 ler e imprimir o estado da porta 254. A linha 30 faz simplesmente com que a mensagem «scroll?» não apareça periodicamente e páre a execução do programa.

Se se fizer ler uma *cassette* gravada durante a execução deste programa, ver-se-ão os números 255, correspondente a b6 alto, e 191, correspondente a b6 baixo, impressos no *écran*. É de notar que a pressão em qualquer tecla altera o valor devolvido por IN 254. O sinal resultante da leitura de uma *cassette* gravada com uma onda quadrada é muitas vezes bastante diferente da onda original (ver figura 8.2). No entanto, o tempo entre pontos extremos deve ser bastante próximo do original. Por outras palavras, o tom da onda quadrada deve ser o mesmo, a não ser que a velocidade do gravador

Fig. 8.2. *Sinal característico obtido pela reprodução, no gravador de* cassettes, *da onda quadrada apresentada na figura 8.1*

tenha mudado (quando, por exemplo, tem as baterias descarregadas). Deste modo, é o tempo entre mudanças do nível do sinal que o *Spectrum* utiliza para ler a informação gravada em fita magnética.

O FORMATO DA INFORMAÇÃO

Todos os ficheiros, no *Spectrum,* são gravados em dois blocos de informação, o «cabeçalho» *(header)* e o «bloco de dados». O cabeçalho é um som audível de curta duração, emitido quando se armazena a informação relativa aos dados guardados no bloco que lhe segue. Por exemplo, o cabeçalho utiliza-se para armazenar o nome daquele ficheiro e o número de *bytes* existentes no bloco de dados. O formato exacto dos dados armazenados no cabeçalho serão descritos mais adiante.

Cada um destes tipos de bloco começa com um som de curta duração: aproximadamente 5 segundos para o cabeçalho e cerca de 2 segundos para o bloco de dados. Este som inicial é uma onda quadrada de 619,4 uS (1 uS= um microssegundo ou um milionésimo de segundo) entre cada mudança de estado (ver figura 8.3); corresponde a uma frequência aproximada de 807 Hz. O fim deste som inicial é marcado pela ocorrência de um impulso de muito curta duração — o impulso de sincronização. O impulso de sincronização é baixo durante 190,6 uS e alto durante 210 uS. A seguir ao impulso de sincronização, e sem qualquer interrupção, aparece o primeiro impulso de dados. O comprimento de um impulso de dados depende de representar um 0 ou um 1. Se representar um zero é baixo durante 244,3 uS e alto para esse mesmo tempo. Se representar um 1 o impulso dura exactamente o dobro do tempo. Toda esta informação sobre tempos é resumida na figura 8.3, onde também se apresenta o número de estados T (impulsos de relógio dos *Z80*) durante os quais dura o impulso.

Utilizando esta informação, será possível escrever um programa de linguagem assembly, praticamente em qualquer máquina, que

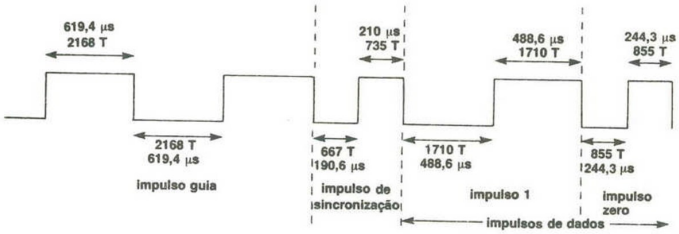


Fig. 8.3. *Os sinais utilizados pelo* Spectrum *no registo em* cassette, *mostrando os tempos em microssegundos e o número de estados T do* Z80 *por impulso.*

permita ler ou gravar *cassetes* de e para o *Spectrum*. No *Spectrum*, a utilização de uma gama de intervalo de tempos aceitáveis para cada tipo de impulso torna muito fiável o processo de reler os dados. Os impulsos de dados, que se seguem ao impulso de sincronização, formam grupos de oito algarismos binários que correspondem aos *bytes* que estão a ser gravados ou lidos. Por outras palavras, os primeiros oito impulsos que seguem o de sincronização formam o primeiro *byte* de dados, os oito seguintes o segundo *byte*, e assim sucessivamente até ao fim do bloco. Além desta, a única informação necessária é o formato dos *bytes* que constituem o cabeçalho e como se relaciona com o bloco de dados.

Um cabeçalho é formado por 19 *bytes* mas só a informação guardada em 17 deles é fornecida pelo utilizador. O primeiro *byte* do cabeçalho ou do bloco de dados é um *byte «flag»*, gerado pela rotina SAVE e que marca a diferença entre um cabeçalho e um bloco de dados. Se o *byte «flag»* for 0 é porque o bloco seguinte é um cabeçalho e, se for 255, o bloco contém dados. O *byte* final do cabeçalho (ou dum bloco de dados) é um *byte* de «paridade», utilizado na detecção de qualquer erro de leitura. Estes dois *bytes*, o *byte «flag»* no começo do bloco e o *byte* de paridade no fim do bloco, são agregados pela rotina SAVE quer ao cabeçalho quer ao bloco de dados, tornando-os assim dois *bytes* mais compridos do que seria de esperar. Observa-se na figura 8.4 a utilização dos 17 *bytes* que formam o cabeçalho. O primeiro *byte* regista o tipo de bloco de dados, de acordo com a tabela seguinte:

TIPO tipo do bloco de dados

0 programa BASIC
1 vector numérico
2 vector de texto
3 código máquina ou cópia do *écran*

1	10	2	2	2
TIPO	NOME DO FICHEIRO	COMPRIMENTO	N.º DA LINHA OU ENDEREÇO DE INÍCIO	COMPRIMENTO DO PROGRAMA

Fig. 8.4. *Formato de um bloco de cabeçalho.*

Os 10 *bytes* seguintes guardam o nome do ficheiro. Depois vêm dois *bytes* que registam o comprimento do bloco de dados seguinte. Os quatro restantes *bytes* utilizam-se conforme o tipo de bloco de dados que o cabeçalho descreve. Se o tipo é 0, os *bytes* 14 e 15 guardam o número de linha para a execução automática de um programa BASIC acabado de carregar, e os *bytes* 16 e 17 guardam o número de *bytes* existentes na parte do ficheiro com o programa. (Recordemos que a gravação, com SAVE, de um programa BASIC grava tanto a área do programa como a área dos dados.) Se o tipo é 1 ou 2, utiliza-se só o *byte* 15, que guardará o nome do vector. Se tipo é 3, só se utilizam os *bytes* 14 e 15 para guardar o endereço a partir do qual se deverão carregar os *bytes* de dados.

Ao cabeçalho segue-se o bloco de dados que ele descreve. Como já se referiu, este tem mais dois *bytes* do que os registados no cabeçalho: o primeiro *byte* chamado *byte* «tipo» e o último, *byte* de paridade. A seguir ao *byte* tipo, pode associar-se cada um dos *bytes* do bloco de dados a uma «imagem» da zona de memória que foi gravada.

Um último pormenor a descrever: a forma exacta como se utiliza o *byte* de paridade para detectar erros de leitura. Ao gravar tanto um cabeçalho como um bloco de dados, faz-se uma operação OR exclusiva entre o *byte* de paridade e cada um dos *bytes* que vai sendo gravado. O valor inicial do *byte* de paridade é dado pelo *byte* «*flag*». Se, ao reler os dados, se for constituindo um *byte* de paridade da mesma maneira, isto é, formando um OR exclusivo

entre o *byte* actual de paridade e cada *byte* lido, e se não houver erros de leitura, o valor final do *byte* de paridade será 0. É de notar que isto pressupõe que o valor inicial de *byte* de paridade era 0, e que foi feita a operação OR exclusiva entre todos os *bytes* lidos, incluindo o *byte* «*flag*» e o *byte* final de paridade, e o referido *byte*.

AS ROTINAS SAVE E LOAD

Existem duas rotinas fundamentais em código máquina, contidas na ROM do BASIC *ZX*, que servem para gravar (SAVE) e carregar (LOAD) uma área da memória. Como sucede com quaisquer das rotinas contidas na ROM do BASIC *ZX*, a posição dessas rotinas fundamentais poderá vir a ser alterada em futuras versões do *Spectrum*, mas não parece que assim seja para estas duas importantes rotinas.

A rotina de gravação começa no endereço 1218 (ou 04C2 em hexadecimal). A sua acção real depende de certo número de parâmetros que lhe são passados pelos registos seguintes:

Registo	acção
DE	número de *bytes* de dados a gravar
IX	endereço do primeiro *byte* de dados
A	0 para o cabeçalho 255 para o bloco de dados

É importante notar que esta rotina grava uma área de memória, tal como ela está, sem mais requisitos ou alterações, à parte da agregação do *byte* «tipo», no início, e do *byte* de paridade, no fim. Em particular, esta rotina não emite quaisquer mensagens avisando para carregar nos botões *play* e *record* do gravador, e não grava automaticamente qualquer cabeçalho se for utilizada para gravar um bloco de dados. Na realidade, se se desejar gravar um cabeçalho, deve-se criar uma área de memória com 17 *bytes* que contenha os dados correctos de acordo com o formato já descrito, por exemplo, o nome do ficheiro, seu comprimento, etc. A não ser que se tenha em mente uma aplicação muito especial, a rotina de gravação é utilizada, geralmente, duas vezes: uma para gravar o cabeçalho e outra para gravar o respectivo bloco de dados.

A rotina de carregamento começa em 1366 (ou 0556 em hexadecimal). Uma vez mais, a sua acção depende de certo número de parâmetros:

Registo	acção
DE	número de *bytes* de dados a carregar
IX	endereço onde deve ser armazenado o primeiro *byte*
A	0 significa carregar o cabeçalho 255 significa carregar o bloco de dados

Se a *flag carry* estiver baixa, a informação não será carregada da memória, mas, sim, comparada com o que já lá existe, isto é, será executada uma verificação. Assim, para a informação ser realmente carregada, a *flag carry* deve estar levantada. Se o tipo do ficheiro não estiver correcto, a rotina regressa com as *flags* zero e *carry* baixas. Se foi detectado um erro ao carregar, levantam-se as duas *flags* (zero e *carry*). É de notar que, para utilizar a rotina de carregamento, devemos saber quantos *bytes* se irão tentar carregar. No caso de carregamento de um cabeçalho é fácil, porque todos têm 17 *bytes* de comprimento. Mas, se carregarmos um bloco de dados, só saberemos o número de *bytes* a carregar lendo o cabeçalho que precedia aquele bloco de dados.

Utilizando estas duas rotinas de gravação e leitura, é possível escrever e ler ficheiros não normalizados em fita. Por exemplo, escrevemos um ficheiro composto de um certo número de blocos de dados, com comprimento fixo, que podem ser lidos à medida que vão sendo necessários. No entanto, o problema principal na utilização do sistema de *cassettes* do *Spectrum* de uma maneira não normalizada é não haver controle do motor do gravador. Se dispusermos de um ficheiro constituído por um conjunto de blocos, teremos de fazer parar e pôr a funcionar o gravador à medida que o *Spectrum* necessite dele!

Para exemplificar a utilização das rotinas de gravação e leitura, apresenta-se um programa que imprime uma lista de tipos de ficheiros e respectivos nomes. Neste sentido constitui um comando, limitado, de catalogação. A primeira parte do programa é uma rotina em linguagem assembly que lê cabeçalhos da fita e os armazena no *buffer* da impressora.

linguagem	assembler	código	observações
23296 LOOP	LD DE, 17	17,17,0	comprimento do cabeçalho em DE
23299	XOR A	175	pôr A a zero
23300	SCF	55	levantar a *flag carry*
23301	LD IX,23311	221,33,15,91	início da área de dados em IX
23305	CALL 1366	205,86,5	chamar a rotina de carregamento
23308	JR NC,LOOP	48,242	saltar para trás, se não é cabeçalho
23310	RET	201	regressar ao BASIC

O código desta rotina é carregado no *buffer* da impressora e, por sua vez, o código armazena os *bytes* do cabeçalho também no *buffer* da impressora com início em 23311. O seguinte programa BASIC utiliza esta rotina para ler cabeçalhos e imprimir o tipo e nome do ficheiro:

```
  10 DATA 17,17,0,175,55,221,33,
15,91,205,86,5,48,242,201
  20 FOR A=23296 TO 23310
  30 READ D
  40 POKE A,D
  50 NEXT A

  60 LET A=USR 23296
  70 PRINT "TYPE=";PEEK 23311
  80 PRINT "NAME=";
  90 FOR I=1 TO 10
 100 PRINT CHR$ (PEEK (23311+I))
;
 110 NEXT I
 120 PRINT
 130 GO TO 60
```

As linhas 10 a 50 carregam o código máquina no *buffer* da impressora. A linha 60 utiliza a rotina para carregar os *bytes* do cabeçalho e as linhas 70 a 120 imprimem o tipo e o nome. Seria bastante fácil generalizar a última parte do programa para pesquisar, com PEEK, dados adicionais do cabeçalho e imprimir o comprimento do ficheiro e o ponto onde se carrega.

SOM

O emissor de som do *Spectrum* tem relações muito íntimas com o sistema da *cassete*. O pequeno altifalante que produz o som está ligado ao mesmo terminal do ULA que as linhas MIC e EAR. A única diferença é que a saída é controlada por b4 da porta 254, de I/O. Se b4 é 0, a voltagem de saída é 0,75 volts, e se b4 é 1, a voltagem de saída é 3,3 volts. É de notar que o intervalo das voltagens obtidas pela alteração de b4 é maior do que o do sistema de gravação. A voltagem mais baixa utilizada pelo sistema de gravação é insuficiente para controlar o altifalante, e por isso não se ouvem os sinais da *cassette*, mas a voltagem utilizada para controlar o altifalante, que é mais alta, aparece realmente tanto em EAR como em MIC.

O método básico para produzir som é idêntico ao utilizado para gerar tons no sistema da *cassette*. Pode conseguir-se uma onda quadrada mudando, repetidamente, o conteúdo de b4 de 0 a 1. O tom do som produzido pela onda quadrada está relacionado com a velocidade conseguida na mudança, repetida, de alto para baixo. À parte do tom da nota, não há nada mais que possa ser alterado. A altura é fixada pelo intervalo de voltagens correspondentes aos dois estados da onda quadrada pela forma da onda. Eis um programa para controlar o altifalante directamente:

```
10 OUT 254,16
20 OUT 254,0
30 GO TO 10
```

Este programa muda b4 de 1 a 0 cada vez que se executa um ciclo. O som resultante é bastante áspero e grave, devido à falta de velocidade do BASIC. É também de notar que a cor da moldura muda para negro porque, de b0 a b2 da porta 254, controla-se a cor da moldura.

BEEP, o comando de som do *Spectrum*, é notável, pois produz uma escala musical precisa. É mais um dos tantos exemplos da forma como a excelente programação de sistema do *Spectrum* consegue tirar o máximo partido de uma característica limitada dos seus componentes físicos.

É muito difícil melhorar o som do *Spectrum* sem utilizar componentes adicionais. No entanto, a seguinte rotina em linguagem assembly controlará a porta de I/O directamente, utilizando uma tabela de valores como dados.

endereço		assembler	código	observação
23296		LD B,count	06,0	número de *bytes* na tabela
23298		LD HL,(table)	237,107,23,91	início da tabela
23302	LOOP	LD A,(HL)	126	carregar dados em A
23303		OR 8	246,8	preparar *bit* MIC
23305		OUT (254),A	211,254	enviar dados à porta
23307		LD C, time	14,0	carregar tempo de atraso
23309	DEL	DEC C	13	ciclo de atraso
23310		JP NZ, DEL	194,13,91	saltar atrás se C<>0
23313		INC HL	35	*byte* de dados seguinte
23314		DEC B	5	fim da tabela?
23315		JP NZ,LOOP	194,6,91	voltar atrás para o resto da tabela
23318		RET	201	regressar ao BASIC
23319		DEFW table		endereço da tabela

A melhor maneira de explicar como se utiliza esta rotina é fazer um exemplo em BASIC. Ruído de fundo é a espécie de «sshh» que se ouve num rádio sintonizado entre estações. Na realidade é uma mistura aproximadamente igual de uma banda muito larga de frequências. Pode produzir-se, no *Spectrum*, uma aproximação a ruído de fundo mudando ao acaso, o valor de b4 da porta de saída 254. O único problema é onde encontrar uma tabela aleatória de algarismos binários. Surpreendentemente, a fonte mais próxima de algarismos binários muito aproximadamente aleatórios é a própria ROM com o BASIC *ZX*. O programa que a seguir apresentamos utiliza a rotina acima referida para enviar 256 *bytes* da ROM à porta de saída.

```
  10 DATA 6,0,237,107,23,91,126,
246,8,211,254,14,0,13,194,13,91,
23,6,194,6,91,201
  20 FOR A=23296 TO 23318
  30 READ D
  40 POKE A,D
  50 NEXT A
```

```
 60 POKE 23319,0
 70 POKE 23320,20
 80 POKE 23297,255
 90 POKE 23308,128
100 LET A=USR 23295
110 GO TO 100
```

A primeira parte do programa carrega o código máquina no *buffer* da impressora. As linhas 60 a 90 inicializam os parâmetros utilizados no controle da rotina. Antes de usá-la devem carregar-se as posições de memória 23319 e 23320 com o endereço do início da tabela de dados. A posição de memória 23297 deve inicializar-se com o número de *bytes* existentes na tabela, e a posição de memória 23308 deve armazenar um valor que produza o tom global desejado. As linhas 100 e 110 chamam repetidamente a rotina, dando como resultado um som sibilante. Como não se limitaram de qualquer modo os dados enviados a b4 do porto de I/O, a cor da moldura também muda aleatoriamente.

Utilizando esta rotina juntamente com tabelas de dados formadas por padrões regulares, é possível produzir um intervalo limitado de efeitos sonoros. Por exemplo, façam-se as alterações seguintes, a partir da linha 60 do último progama:

```
 60 GO SUB 1000
 70 POKE 23319,25
 80 POKE 23320,91
 90 POKE 23307,N
100 POKE 23308,250
110 LET A=USR 23296
120 PAUSE 1
130 GO TO 110

1000 LET N=220
1010 LET S=1
1020 LET D=0
1030 LET C=1
1040 FOR I=0 TO N-1
1050 IF I=S THEN GO SUB 2000
1060 PRINT D
1070 POKE I+23321,D*16
1080 NEXT I
1090 RETURN

2000 LET S=INT (S+C)
2010 LET C=C+.25
2020 IF D=0 THEN LET D=1: RETURN
2030 LET D=0
2040 RETURN
```

As sub-rotinas 1000 e 2000 preparam uma tabela de valores no *buffer* da impressora tal que a frequência da onda decresce com o tempo. O som resultante é uma espécie de «zas». Cada qual experimente criar os seus próprios efeitos sonoros, mudando o padrão de 0 e 1 produzido pela sub-rotina 1000.

A IMPRESSORA «ZX»

A impressora *ZX* apresenta uma maneira, notavelmente barata, de imprimir em papel listagens e gráficos. Talvez os seus únicos pontos fracos sejam a pouca precisão do posicionamento do ponteado que forma os caracteres e a qualidade do papel de alumínio que utiliza, pontos que, aliás, são resolvidos na última impressora da Sinclair, a *Timex 2040*. Vale a pena mencionar, entretanto, que o papel de alumínio se fotocopia muito bem!

A impressora *ZX* funciona pela evaporação da camada superficial de alumínio sobre um rolo de papel negro. Nos pontos onde o alumínio se evaporou, ressalta o negro do papel de fundo, não sendo portanto nenhuma espécie de tinta que imprima os caracteres. A impressora *ZX* utiliza a faísca produzida por duas pontas metálicas movediças para evaporar o alumínio, e se se puser a impressora a funcionar no escuro, ver-se-ão claramente faíscas eléctricas azuladas mesmo por baixo da barra de cortar o papel. As pontas metálicas estão montadas em lados opostos de uma banda móvel, de forma a que uma delas está sempre posicionada sobre o papel. O papel move-se graças a um motor eléctrico, à medida que passa pelas pontas que o percorrem, de modo que a cada deslocamento da ponta corresponde uma nova linha ponteada no papel.

Os programas *(software)* do *Spectrum* que controlam a impressora *ZX* são bastante completos, e pouco se pode fazer para melhorá-los. No entanto, o modo como a impressora *ZX* é controlada constitui um exemplo interessante da forma como os computadores controlam os equipamentos. E conhecer o funcionamento da impressora talvez sugira novas aplicações.

A impressora está ligada ao porto 251, de I/O. A leitura do porto dá informação acerca do estado da impressora nesse momento. Se

a impressora não está ligada, então b6 será 1 (e, inversamente, b6 será 0 se a impressora estiver ligada). O estado de b7 reflecte a posição das pontas metálicas ou agulhas. Se uma delas estiver em posição sobre o papel, b7 é 0. Assim, b7 servirá para conhecer o momento em que uma das agulhas está sobre o papel, pronta a imprimir uma linha ponteada. A velocidade com que as agulhas percorrem o papel varia, dependendo da carga no motor. Para ultrapassar esta dificuldade, foi adicionado ao motor um disco de codificação, daí resultando que b0 pulsa aproximadamente 256 vezes à medida que uma agulha percorre uma linha. Assim, estando a produção de pontos relacionada com a pulsação de b0, o ponteado ficara espaçado igualmente, mesmo que o motor varie de velocidade.

Na saída para a porta 251, o motor é controlado por b1 e b2. Se b2 é 0, o motor da impressora arranca. Se b1 é 0, o motor trabalha rapidamente; se não, vai mais lentamente. Esta velocidade mais baixa é utilizada para imprimir as duas últimas linhas, de forma que as agulhas possam parar fora do papel, prontas para imprimir a primeira linha, na próxima vez que a impressora seja utilizada. Para finalizar, b7 controla a voltagem nas agulhas. Se b7 é 1, então as agulhas estarão sob alta tensão e a faísca resultante queimará o papel, fazendo-lhe uma pequena marca negra.

Para além da forma como os algarismos binários assinalam o estado da impressora e controlam a sua operação, há mais um ou dois pormenores importantes. Em primeiro lugar, a voltagem da agulha deve ser zero para se detectar quando atingem o lado do papel. Isto justifica-se para a voltagem da agulha pôr b7, da entrada, automaticamente, em estado alto. Em segundo lugar, b0 e b7 estão ambos *latched* (alerta) até que seja escrita informação para a porta de I/O. («Latched» é terminologia inglesa de electrónica para «manter-se constante até novas instruções»). Por isso, mesmo que desejássemos ter novas informações dos *bits* b0 e b7, teríamos primeiro de escrever algo na porta referida.

Como todas as operações da impressora são muito rápidas, não há esperança de as controlar utilizando BASIC *ZX*. A rotina seguinte, escrita numa mistura entre linguagem assembly e português, exemplifica o método fundamental utilizado para escrever uma fila única de 256 pontos.


```
          LDA 0
          OUT 251,A          arrancar com o motor á máxima
                             velocidade
fundo     IN A,251           obter condições da impressora
          RL A               rodar um bit à esquerda para
          JP M, não imprimir testar a impressora
          JP NC, fundo       testar se agulha está sobre papel
código    IN A,251           agora ler bit do disco de
                             codificação
          RR A
          JP NC, código      e esperar até que seja 1

e então,  ou:
          LDA 0
          OUT 251,A          imprimir um ponto da zona do
                             fundo

ou
          LDA 128
          OUT 251,A          imprimir um ponto de «tinta»
```

Para imprimir 256 pontos numa linha, repete-se este processo, regressando à posição «código». Tome-se ainda em consideração que o motor deve ser desacelerado ao imprimir as duas últimas linhas.

Como já se referiu, a descrição que demos tem pouca aplicação prática para o programador do *Spectrum,* porque a programação própria *(software)* do *Spectrum* fornece todas as oportunidades necessárias para a utilização da impressora *ZX*. Um projecto possível é a utilização da impressora *ZX* com outros computadores — mas isso está, evidentemente, para além da finalidade deste livro!

CAPÍTULO 9

«Interface 1» e as «microdrives»

A adição duma *Interface 1* e várias *microdrives* ao *Spectrum* converte-o num sistema de computação muito poderoso e versátil. A *Interface 1* agrega os componentes físicos e programação necessários a uma *interface* série *RS232*, normalizada, e ainda agrega uma rede local. Estas duas características tornam a *interface 1* importante por si mesma, e a rede local é tão interessante que mereceria um livro só para ela!

As *microdrives* conferem ao *Spectrum* uma nova dimensão no tratamento de dados. Baseadas num anel fechado de fita, as *microdrives* não são tão rápidas como *diskettes* nem podem (presentemente) armazenar tanta informação. Para aplicações simples — por exemplo, gravar e carregar programas — as *microdrives* são um sistema do tipo *cassette*, mas mais rápido. Não será esta diferença de velocidades justificativa para preferir *microdrives:* o seu tempo de resposta está longe de ser instantâneo, e ainda teremos de esperar alguns segundos enquanto carregamos um programa. A razão principal para a utilização de *microdrives* é que elas abrem novas possibilidades em aplicações que eram difíceis, se não impossíveis, de realizar no *Spectrum*. Por exemplo, é muito difícil imaginar como é que se processariam mesmo pequenas quantidades de dados se os resultados também tivessem de ser armazenados em fita. O *Spectrum* não expandido está principalmente limitado pelo facto de ter que processar quantidades de dados suficientemente pequenas para caberem na RAM disponível. Mesmo só com uma *microdrive* é possível ler de um ficheiro enquanto se escreve noutro. Além disso, a extensão ao BASIC *ZX*, dada na *Interface 1*, permite a criação de ficheiros «reais», e não só o armazenamento de vectores e quadros *(arrays)*. Dito de outro modo, as *microdrives* podem não ser tão rápidas como as *diskettes* mas têm a possibilidade de realizar o mesmo tipo de operações. E basta que se possa armazenar um certo número de programas numa única *cartridge* para justificar a expansão de qualquer *Spectrum*.

Este capítulo estuda as extensões ao BASIC *ZX* que acompanham a *Interface 1*. A parte final do capítulo dá alguns exemplos simples da utilização das novas características para criar e processar ficheiros de dados. O capítulo seguinte examina algo do funcionamento interno da *Interface 1* e das *microdrives*. Este assunto é tão vasto que só há possibilidade de oferecer princípios gerais e pormenores importantes. No entanto nessa altura já cada qual deverá ser capaz de criar os seus próprios programas.

O BASIC «ZX» DA «MICRODRIVE» – ESPECIFICADORES DE FICHEIRO

A *Interface 1* contém uma ROM adicional, de 8 k, que amplifica os 16 k da ROM onde, no *Spectrum*, se guarda o BASIC *ZX*. Descrevemos mais adiante a forma como se conseguiu esta amplificação. Neste momento, tem maior interesse descrever a forma que tomam estas adições e como utilizá-las.

Os comandos adicinais do BASIC alargado, que será referido como «BASIC *ZX* da *microdrive*», ou BASIC *ZXM*, dividem-se em quatro categorias:

1) comandos alargados de gravação e leitura, como LOAD*, SAVE*, etc.
2) comandos novos, referentes só à *microdrive*, como CAT, etc.
3) comandos alargados de canal
4) comandos *ad-hoc* CLEAR# e CLS#

A forma destes comandos é muito mais fácil de compreender e relembrar, uma vez que se saiba que a informação é armazenada numa *microdrive* na forma de um «ficheiro» com nome, como se faz na fita. A diferença principal é que, para identificar um ficheiro na fita, basta dar o «nome de ficheiro», enquanto na *microdrive* é necessário dar um «especificador de ficheiro» completo. O formato de um especificador de ficheiro é:

dispositivo; número do dispositivo; nome do ficheiro

onde «dispositivo» é uma variável de texto que identifica o tipo de dispositivo onde está armazenado o ficheiro, «número do dispositivo» é um número que identifica exactamente qual é esse dispositivo e, finalmente, «nome do ficheiro» é uma variável de texto com o nome do ficheiro, que segue a regra habitual de um máximo de 10 letras. Qualquer dos parâmetros pode ser substituído por variáveis

de tipo correcto. Por exemplo, as *microdrives* são especificadas por um código de dispositivo «M» ou «m» e por isso

«m»;2;«ficheiro»

especifica um ficheiro chamado «ficheiro», armazenado na segunda *microdrive* do sistema. Embora os especificadores que usam constantes sejam, de longe, os mais comuns, vale a pena recordar que

D$;D;F$

é perfeitamente legal desde que D$ contenha um tipo de dispositivo, D o seu número e F$ um nome de um ficheiro com o máximo de 10 letras. Por agora, o único tipo de dispositivo a utilizar é «m» para as *microdrives*, mas, no capítulo 11, daremos outros códigos de dispositivo para referir os outros dispositivos controlados pela *interface 1*.

EXTENSÕES AOS COMANDOS DA «CASSETTE»

Uma vez conhecido o formato de um especificador de ficheiro, os novos comandos BASIC são muito fáceis de relembrar. As extensões aos comandos que, previamente, controlavam a operação da *cassette* são:

LOAD* especificador de ficheiro
MERGE* especificador de ficheiro
SAVE* especificador de ficheiro
VERIFY* especificador de ficheiro

Estes comandos executam as acções familiares à operação da *cassette*, mas utilizando os dispositivos controlados pela *interface 1*. Por exemplo

SAVE* «m»;1;«meuprog»

lê-lo-á da *microdrive* para a memória central. Tanto VERIFY* como MERGE* funcionam nas *microdrives* como os seus semelhantes no sistema de *cassette*. Os comandos seguintes são todos válidos nas *microdrives* e idênticos, na sua operação, aos comandos equivalentes da *cassette:*

SAVE* especificador de ficheiro LINE número
SAVE* especificador de ficheiro DATA nome de vector ()
SAVE* especificador de ficheiro CODE início, comprimento
SAVE* especificador de ficheiro SCREEN$

LOAD* especificador de ficheiro DATA nome de vector ()
LOAD* especificador de ficheiro CODE início, comprimento
LOAD* especificador de ficheiro SCREEN$

OS NOVOS COMANDOS DA «MICRODRIVE»

Há quatro comandos da *microdrive* completamente novos.

CAT número da unidade

Este comando produzirá uma lista dos ficheiros existentes na *microdrive* indicada em «número da unidade», que pode ser uma variável. Por exemplo, CAT 1 dá uma relação da *microdrive 1*, e CAT d dará a relação da unidade indicada pelo valor armazenado em d.

ERASE especificador de ficheiro

Este comando eliminará, isto é, apagará, o ficheiro indicado no especificador. O espaço ocupado pelo ficheiro fica então disponível. Por exemplo, ERASE«m»;1;«meuprog» removerá o ficheiro «meuprog» da *cartridge* que estiver na *microdrive 1*.

FORMAT especificador de ficheiro

Este comando elimina todos os ficheiros na *cartridge* e prepara-a para nova utilização. Uma *cartridge* completamente nova tem que formatar-se antes de poder utilizar-se. O «especificador de ficheiro» usado com este comando selecciona o dispositivo a ser formatado e o «nome do ficheiro», contido no especificador, é o nome dado à *cartridge*. Por exemplo, FORMAT«m»;1;«dados» formatará a *cartridge* da *microdrive 1* e dar-lhe-á o nome de «dados».

MOVE especificador 1 (de ficheiro) TO especificador 2 (de ficheiro)

Este comando copiará o ficheiro indicado no «especificador 1» para o dispositivo, e com o nome, indicados no «especificador 2». Por exemplo, MOVE«m»;1;«dados» TO«m»;2;«dados 2» criará uma cópia do ficheiro «dados 1» na *microdrive 2* e chamá-lo-á «dados 2». Com o comando MOVE podem fazer-se duas cópias do mesmo ficheiro (com nomes diferentes) na mesma unidade ou duas cópias do mesmo ficheiro (talvez até com o mesmo nome) em unidades

diferentes. É importante notar que MOVE somente funciona com ficheiros de dados, isto é, com ficheiros que não tenham sido criados com SAVE. Para copiar ficheiros de programas basta utilizar LOAD* e SAVE*. Há outras maneiras mais complexas de utilizar MOVE e que explicaremos melhor mais adiante.

OS COMANDOS DE CANAIS E «STREAMS»

Os comandos relativos a canais e *streams* das *microdrives* não são difíceis de compreender ou relembrar. Enquadram-se na filosofia geral de canais e *streams* descrita no capítulo 5. E não há dúvida de que somente com a *Interface 1* ligada ao *Spectrum* é que o sistema de canais e *streams* de I/O se torna realmente útil.

O comando OPEN # serve ainda para associar canais a *streams*, mas agora aumenta o número de especificadores de canal para poder incluir os especificadores de ficheiro. Isto é

OPEN #s, especificador de ficheiro

associa o *stream* «s» com o canal dado pelo «especificador de ficheiro». Por exemplo

OPEN #5, «m»;1;«dados»

abre o *stream* 5 ao canal formado pelo ficheiro «dados» na *microdrive 1*. É de notar que esta definição generaliza o conceito de canal como dispositivo de I/O de modo a incluir qualquer origem ou destino de informação, separada e identificável. Neste sentido, embora a *microdrive* seja um único dispositivo de I/O, o facto de poder guardar ficheiros diferentes, cada um ainda com sua origem ou destino de informação, torna-a de mais fácil compreensão se for visualizada como um certo número de canais diferentes. Uma vez aberto, com OPEN, um *stream* a um ficheiro, os comandos habituais INPUT #, INKEY # e PRINT # podem utilizar-se para ler e escrever dados, e o comando CLOSE # para terminar aquela associação. E com LIST # até se envia a listagem de um programa para um ficheiro da *microdrive*. Um ponto importante a ter em conta no funcionamento de um canal é que o comando PRINT lhe envia o mesmo *stream* de códigos ASCII, não importando a que se refira, e a instrução INPUT interpreta os códigos ASCII que receba, exactamente da mesma maneira, não importando a que se refira o canal. Este princípio é evidente quando os dispositivos associados a canais são os habituais teclado, *écran* e a impressora, mas já não

será assim quando o canal é um ficheiro numa *microdrive*. Por exemplo, se A=1234, então PRINT #5;A enviará cinco códigos ASCII (49, 50, 51, 52 e 13, correspondentes aos dígitos 1, 2, 3, 4 e ENTER) ao caudal 5, seja qual for o dispositivo associado com esse caudal. Apesar de ser enviada às *microdrives* uma sequência de códigos igual à enviada a outro dispositivo qualquer, há alguns casos nos quais o canal de um ficheiro se comporta de modo diferente. Para ilustrar estas diferenças não há melhor do que um exemplo.

LEITURA E ESCRITA NUM FICHEIRO – «BUFFERING»

Consideremos o problema de escrever 20 números aleatórios num ficheiro seguido pela sua leitura. Uma das muitas soluções possíveis é:

```
 10 OPEN #5,"m";1;"dados"
 20 FOR i=1 TO 20
 30 LET X=RND
 40 PRINT X
 50 PRINT #5;X
 60 NEXT I
 70 CLOSE #5
 80 OPEN #5,"m";1;"dados"
 90 FOR I=1 TO 20
100 INPUT #5;R
110 PRINT R
120 NEXT I
130 CLOSE #5
```

Ao executar este programa e observar ou escutar a *microdrive*, descobriremos que a fita corre durante algum tempo antes de os números aleatórios começarem a ser impressos, na linha 40. Seguidamente, e depois de os 20 números aparecerem no *écran*, o motor arranca, as cores da moldura acendem e apagam e, depois de nova espera, os números são impressos, novamente, no *écran*. A razão desta sequência de operações é que a informação que vai, ou que vem, das *microdrives* é guardada temporariamente num *buffer*. Em vez de ser enviado à *microdrive* assim que se executa o PRINT, cada carácter é recolhido numa área de memória, conhecida como *buffer*, até que haja caracteres suficientes para pôr a *microdrive* a

funcionar. Isto significa que os dados só serão enviados para a *microdrive* quando o *buffer* ficar completo. Como o *buffer* pode guardar 512 caracteres, o programa dado termina antes de enchê-lo e, neste caso, a instrução CLOSE na linha 70 executa um trabalho adicional: além de assinalar ao *Spectrum* que deve terminar com a associação entre *stream* e canal, deve ainda informá-lo que há necessidade de enviar um *buffer* incompleto para as *microdrives*.

Sem esta instrução CLOSE, os dados continuariam no *buffer* e jamais seriam enviados. A *microdrive* arranca pela primeira vez porque o comando OPEN está à procura de um ficheiro, existente ou não, chamado «dados». Ao fim de percorrer, sem sucesso, toda a fita, a *microdrive* é desligada, e o ciclo FOR imprime os dados no *écran*, e também a envia para o *stream* 5, onde é recolhido num *buffer*. A *microdrive* arranca pela segunda vez por causa da instrução CLOSE, e o conteúdo do *buffer* incompleto é escrito no ficheiro «dados». A instrução seguinte, OPEN, faz com que a fita seja percorrida à procura de um ficheiro chamado «dados» e, como o consegue encontrar desta vez, lê a informação necessária para encher o *buffer*. Quando se inicia o segundo ciclo FOR para introduzir, com INPUT #, dados, a *microdrive* é desligada porque esses dados vêm do *buffer*. Se houver mais do que 512 caracteres de dados para ler do ficheiro, a *microdrive* arrancará novamente para ler outro *buffer* de dados. Resumindo:

1) O comando OPEN procura, na fita, o ficheiro especificado. Se o encontra, lê um *buffer* de dados, que fica à disposição do primeiro INPUT # feito para aquele *stream*.

2) Dados enviados para o ficheiro por um comando PRINT # são recolhidos num *buffer* de 512 caracteres, antes de serem escritos nesse ficheiro.

3) A instrução INPUT # toma os dados do *buffer*, a não ser que esteja vazio. Se estiver vazio será lido outro *buffer* de dados.

4) O comando CLOSE envia automaticamente os dados de um *buffer* incompleto para a *microdrive*.

Não há uma verdadeira necessidade de compreender a operação exacta do sistema de *buffer* utilizado pelo *Spectrum*, mas ajudará a

explicar a razão pela qual a *microdrive* arranca em ocasiões inesperadas.

A UTILIZAÇÃO DE PRINT #, INPUT # E INKEY

Os canais associados com *streams* de ficheiros têm uma ou duas características especiais que vale a pena descrever. Em primeiro lugar, somente se pode enviar dados para um ficheiro que não exista antes da instrução OPEN e, o que é mais evidente, só se pode ler de um ficheiro que já exista antes da instrução OPEN. Um ficheiro só pode ser aberto com OPEN, para ler ou escrever, e não para ambas as operações ao mesmo tempo. Se quisermos ter a certeza de que um ficheiro não existe antes de tentar escrever nele, poderemos experimentar apagá-lo primeiro, com ERASE. Por exemplo, agregue-se

```
5 ERASE "m";1;"dados"
```

ao programa da última secção.

É um pouco mais difícil ter a certeza que não se vai escrever num ficheiro que esteja a ser lido. Uma intrução INPUT tal como

```
INPUT # 5,A
```

enviará o código de controle relativo a «mover para a zona de impressão seguinte» para o caudal 5, por causa da vírgula antes do A. Para evitar o envio de informação para ficheiros de leitura, as instruções INPUT só devem utilizar pontos e vírgulas como separadores. Semelhantemente, o único separador a ser utilizado numa instrução PRINT que envie informação para um ficheiro é a apóstrofe. A justificação deste facto já foi dada anteriormente (ver capítulo 6): a sequência de caracteres enviada para um ficheiro com uma instrução PRINT é, exactamente, a mesma que a enviada para a rotina de controle do vídeo. No entanto, ao ler outra vez o ficheiro, a instrução INPUT aceita caracteres do ficheiro e trata-os como se tivessem sido digitados no teclado.

Como o leitor verificará muito rapidamente, a única forma válida para terminar a introdução de um dado numa instrução INPUT é carregar em ENTER. Por exemplo, a única maneira correcta de introduzir dados, em resposta a

```
10 INPUT A;B;C$
```

é digitar um número válido e carregar em ENTER, seguido de outro

número válido e ENTER, e finalmente uma sequência válida de caracteres seguida de ENTER. Esta regra (terminar cada dado com ENTER) também se aplica no INPUT de ficheiros nas *microdrives*, mas, utilizando PRINT, é possível criar ficheiros que contêm sequências de caracteres que não podem ser lidas! Por exemplo, experimente-se

```
10 OPEN #5,"m";1;"ilegivel"
20 LET A=RND: LET B=RND
30 PRINT A,B
40 PRINT #5;A,B
50 CLOSE #5
60 OPEN #5,"n";1;"ilegivel"
70>INPUT #5,A;B
80 PRINT A,B
```

e execute-se. O resultado será uma quebra na linha 70, porque a linha 40 escreve dois números separados pelo código de controle «,» (código 6 em ASCII) e não há forma em que esta sequência de caracteres possa ser lida por uma instrução INPUT que use variáveis numéricas. No entanto, pode ser lida usando uma variável de texto, por exemplo,

```
70>INPUT #5;A$
80 PRINT A$
```

que lerá a sequência completa de códigos ASCII enviados ao ficheiro pelo comando PRINT, e os armazenará na variável de texto A$. O leitor também poderá ler o ficheiro, carácter a carácter, utilizando INKEY$:

```
70>LET A$=INKEY$ #5
80 PRINT A
90 GO TO 70
```

Em geral, INKEY$# devolve o carácter que vier a seguir naquele ficheiro, sem importar qual seja — carácter que poderá ser impresso ou não (tal como um código de controle).

Isto significa que se desejarmos escrever um item de informação num ficheiro e voltar a lê-lo como item separado, o item deverá, obrigatoriamente, ser seguido por um código ENTER. Este código ENTER pode ser gerado automaticamente no fim de uma instrução PRINT #, ou incluindo apóstrofes entre itens. Por exemplo

PRINT #s;A
PRINT #s;B

e

PRINT #s;A'B

são idênticos: escrevem dois itens numéricos no ficheiro associado com o *stream* s.

Um exemplo final, e bastante surpreendente, de um INPUT de um ficheiro ser tratado exactamente como se fosse um INPUT do teclado:

```
10 OPEN #5,"m";1;"questoes"
20>PRINT #5;"2*2"
30 CLOSE #5
40 OPEN #5,"m";1;"questoes"
50>INPUT #5;A
60 PRINT A
70 CLOSE #5
```

Admitir-se-ia que como a linha 20 escreve no ficheiro uma mensagem não numérica (2*2), a instrução INPUT na linha 50 nunca poderia ser executada. Mas é: sucede que a expressão é calculada e a resposta, 4, é armazenada em A. Isto resulta de que uma expressão numérica digitada em resposta a um INPUT será primeiramente calculada e só depois se armazenará o seu resultado!

Resumindo, as regras a recordar são:

1) Cada item de informação escrito num ficheiro, e que seja necessário ler outra vez como um item isolado, deve ser seguido por um código ENTER.

2) Um item numérico deve ser uma expressão numérica válida.

3) Um item de texto pode incluir qualquer tipo de carácter e o seu carácter final é um código ENTER.

CAT AVANÇADO

O formato completo do comando CAT é:

CAT #s, número da unidade

que enviará a saída do catálogo ao *stream* s. Por isso

CAT #3,1

catalogará a unidade 1 na impressora, o canal aberto, por defeito, ao *stream* 3. Uma das principais utilidades desta forma do comando CAT é preparar um ficheiro na *microdrive* que contenha toda a informação sobre os ficheiros de uma *cartridge*. Por exemplo:

```
10 ERASE "m";1;"ccat"
20 OPEN #4,"m";1;"ccat"
30 CAT #4,1
40>CLOSE #4
```

criará um ficheiro de dados contendo o catálogo actual da unidade 1. Este ficheiro pode ser lido mais adiante, no programa, para saber se já existia um dado ficheiro, ou simplesmente para saber que espaço ainda resta na *cartridge*.

MOVE AVANÇADO – «RENAME» E «APPEND»

O comando MOVE tem uma forma mais geral em que qualquer dos especificadores pode ser substituído por números de *streams*. Por exemplo, o comando

MOVE«m»; 1; «ccat»TO # 2

moverá a informação do ficheiro «ccat» para o *écran*, o canal aberto, por defeito, ao *stream* 2. Desta forma, pode utilizar-se MOVE para listar ficheiros de dados no *écran* ou na impressora. É possível mover, com MOVE, informação do teclado para um ficheiro de dados, mas terminar a transferência de dados é muito complexo, e é melhor não utilizar MOVE para ligar *streams* uns aos outros de formas não ortodoxas. Por exemplo

MOVE #1 TO #3

moverá informação do teclado para a impressora *ZX*, mas a única maneira de acabar esta ligação é desligar a máquina!

Não existe nenhum comando que permita explicitamente dar outro nome a um ficheiro já existente, mas o comando MOVE serve para o mesmo efeito. O comando

MOVE especfich 1 TO especfich 2: ERASE especfich 1

primeiro fará uma cópia de «especfich 1» com um novo nome, «especfich 2» e em seguida apagará (ERASE) a cópia antiga, dando efectivamente outro nome ao ficheiro.

Se MOVE for utilizado com especificadores de ficheiro, fechará o ficheiro no fim da operação mas, usando-se números de *streams*, o *stream* ficará aberto até ser explicitamente fechado (CLOSE). Isto possibilita o uso do comando MOVE para acrescentar um ficheiro de dados a outro. Por exemplo

```
10 OPEN #4,"m";1;"longo"
20 MOVE "m";1;"primeiro" TO #4
30 MOVE "m";1;"segundo" TO #4
40>CLOSE #4
```

acrescentará o ficheiro «segundo» ao ficheiro «primeiro», sendo o resultado guardado num ficheiro chamado «comprido». Como o comando MOVE não fecha *streams* ao terminar a sua operação, pode transferir um ficheiro completo para qualquer posição dentro do outro. Poderíamos adicionar dados a um ficheiro movendo-o, com MOVE, para um novo ficheiro, imprimindo nele (PRINT) os novos dados, e depois, usando MOVE e ERASE, poderíamos dar ao novo ficheiro o mesmo nome que o original.

CLEAR # e CLS

Os comandos CLEAR # e CLS # parecem ter sido agregados ao BASIC *ZX* mais para melhorá-lo do que por serem realmente necessários. CLEAR # repõe os *streams* e canais nos seus estados iniciais, depois de ligar o sistema. Com efeito este comando fecha (CLOSE) todos os *streams* e repõe os *streams* 0 a 3 nos seus canais, por defeito. No entanto, é importante observar que CLEAR # não é um substituto para o fecho (CLOSE) de quaisquer ficheiros que estiverem abertos. A diferença é que, depois de um CLEAR #, toda a informação armazenada em *buffers* incompletos é desprezada sem ser escrita na *microdrive!* (Recordemos que um CLOSE fará com que seja escrita toda a informação contida em *buffers* nos ficheiros apropriados antes de terminar a associação estabelecida *stream*/canal.)

O comando CLS #, além de limpar o *écran* como o faria CLS, também torna a inicializar todos os atributos do *écran* aos seus valores logo após ligar o sistema, isto é, INK será negra, PAPER

será branco e assim por diante. É uma boa ideia começar todos os programas, a serem utilizados com a *interface* 1, da seguinte forma:

```
10 CLS #: CLEAR #
```

Asseguraremos assim que todos os atributos terão os seus valores iniciais e que todos os *streams* (à parte, por defeito, os 0 a 3) serão fechados.

O PROBLEMA DO FIM DE FICHEIRO

Até agora não foi considerado o problema de saber quando é que um dado programa atingiu o último elemento, ao ler um ficheiro. Isto é importante porque o *Spectrum* dará uma mensagem de erro, seguida de uma quebra, se tentar ler qualquer elemento para além do fim do ficheiro. Diversamente de outras versões do BASIC, não existe qualquer função no BASIC *ZX* para detectar o fim de ficheiro; por isso devemos utilizar uma rotina em código máquina especialmente escrita para isso ou colocar uma marca especial no fim de cada ficheiro. A utilização da marca especial é bastante fácil em BASIC *ZX*, porque há alguns códigos de caracteres que, habitualmente, nunca aparecem. Por exemplo, aos códigos de CHR$(0) a CHR$(5) não é atribuído significado em BASIC *ZX*, podendo assim ser utilizados para referenciar certos pontos de um ficheiro. Esta utilização de marcas ou *flags* é fácil se a informação for armazenada sob a forma de texto, mas não é, evidentemente, possível a inclusão de semelhantes caracteres com dados numéricos.

A solução deste problema é sempre ler dados numéricos para uma variável de texto, e depois verificar se nesse texto existe a marca de fim de ficheiro. Se a variável de texto não é a *flag* de fim de ficheiro, é presumivelmente um item de dados numéricos válido e pode ser convertida à forma numérica utilizando VAL. Por exemplo, se CHR$(0) for utilizado para marca de fim de ficheiro, o programa seguinte escreverá um número aleatório de itens de dados num ficheiro, e depois lê-los-á sem causar qualquer erro de fim de ficheiro:

```
10 OPEN #4,"m";1;"aleatorio"
20 LET L=INT (RND*50)+100
30 FOR I=1 TO L
40 PRINT #4;RND
50 NEXT I
60 PRINT #4;CHR$ 0
70 CLOSE #4
80 OPEN #4,"m";1;"aleatorio"
90 INPUT #4;a$
100 IF A$=CHR$ 0 THEN CLOSE #4:
STOP
110 LET A=VAL A$
120 PRINT A
130 GO TO 90
```

UM PROGRAMA RÁPIDO PARA APAGAR FICHEIROS

Uma das ocupações mais aborrecidas que se podem imaginar é tentar apagar os ficheiros não desejados de uma *cartridge*. O programa seguinte, que lê o catálogo e depois nos pergunta se desejamos ou não apagar cada ficheiro, evita digitar repetidas vezes ERASE, etc.:

```
10 CLEAR #: CLS #
20 INPUT "Drive no. ? ";D
30 PRINT AT 10,8;"Espere, por
favor"
40 ERASE "m";D;"ccat"
50 OPEN #4,"m";D;"ccat"
60 CAT #4,D
70 CLOSE #4
80 OPEN #4,"m";D;"ccat"
90 CLS
100 INPUT #4;C$
110 PRINT AT 0,10;C$
120 INPUT #4;F$
130 INPUT #4;F$
140 IF LEN F$=0 THEN GO TO 220
150 PRINT "Apagar ";F$;" s/n ?"
160 INPUT A$
170 IF A$(1)<>"N" AND A$(1)<>"Y
" THEN GO TO 150
180 PRINT A$
190 IF A$(1)="N" THEN GO TO 130
200 ERASE "m";D;F$"
210 GO TO 130
220 PRINT "Nao ha mais ficheiro
s"
```

A primeira parte do programa (linhas 10 a 80) cria um ficheiro «ccat» na unidade especificada, que contém o catálogo dessa mesma unidade. É de notar a maneira como se utiliza d para especificar o número da unidade. A segunda parte do programa (linhas 90 a 200) lê o ficheiro para obter o nome de cada um dos ficheiros presentes no catálogo, e pergunta, um por um, se deve ser

eliminado ou não. O INPUT duplo nas linhas 120 e 130 não é por engano! A primeira entrada no ficheiro «ccat» é o nome da *cartridge*, que é lido na linha 100. Em seguida vem a variável de texto nula, lida na linha 120, e só depois aparece o primeiro nome de um ficheiro propriamente dito, que é lido em 130.

Depois desta sequência, cada nova leitura do ficheiro dá um nome de ficheiro ou uma variável de texto nula que assinala o fim da lista de ficheiros. A variável de texto nula é detectada na linha 140 e utilizada para terminar o programa.

TRATAMENTO DE FICHEIROS DE DADOS – UM EXEMPLO

O exemplo anterior ilustrava um modo pelo qual se podem utilizar comandos comuns do BASIC *ZX* para executar operações úteis, na *microdrive*. O tratamento de ficheiros de dados parece ser uma aplicação, de tal maneira simples, dos comandos do BASIC que parecem ser desnecessários quaisquer exemplos. Na prática, contudo, esse tratamento de ficheiros prova muitas vezes estar cheio de subtis armadilhas. O curto exemplo seguinte envolve a criação e manutenção do tipo mais simples do ficheiro de dados — um ficheiro sequencial. A aplicação é, no fim de contas, uma lista telefónica pessoal, mas isso é irrelevante para o exemplo. Toda a informação a ser armazenada, agregada e depois examinada apresentaria o mesmo conjunto de problemas na sua programação.

O exemplo consiste em dois pequenos programas. O primeiro utiliza-se para agregar entradas novas à lista:

```
  10 CLEAR #: CLS #
  20 GO SUB 1000

  30 CLS
  40 PRINT "Introduza novos nome
s e numeros"
  50 PRINT "introduzir # para te
rminar"
  60 INPUT "Apelido";S$
  70 IF S$="#" THEN GO TO 180
  80 INPUT "Iniciais";I$
  90 INPUT "Numero de telefone";
T$
 100 PRINT AT 5,0;"Nova entrada-
";T$
 110 PRINT AT 10,0;I$;" ";S$
 120 PRINT " Tel ";T$
 130 INPUT "Esta correcto (y/n)?
";A$
 140 IF A$(1)<>"y" AND A$(1)<>"n
" THEN GO TO 130
 150 IF A$(1)="n" THEN GO TO 30
 160 PRINT #4;S$'I$'T$
 170 GO TO 30

 180 PRINT #4;CHR$ 0'CHR$ 0'CHR$
 0
 190 CLOSE #4
 200 ERASE "m";1;"numtel"
 210 MOVE "m";1;"temp$$$" TO "m"
;1;"numtel"
 220 ERASE "m";1;"temp$$$"
 230 STOP

1000 OPEN #5,"m";1;"numtel"
1010 OPEN #4,"m";1;"temp$$$"
1020 INPUT #5;S$'I$'T$
1030 IF S$=CHR$ 0 THEN RETURN
1040 PRINT #4;S$'I$'T$
1050 GO TO 1020
```

As linhas 10 e 20 são o início de tudo. A linha 20 chama a sub-rotina 1000, que lê o ficheiro, já existente, de números de telefone chamado «numtel» e cria um novo ficheiro chamado «temp $$$». O ficheiro «numtel» está organizado em grupos de três itens de texto. O primeiro item é o apelido, o segundo as iniciais e o terceiro o número de telefone. O fim do ficheiro é assinalado por um grupo de três itens, todos iguais a CHR$0. A sub-rotina 1000 copia o ficheiro «numtel» para «temp$$$» de forma aos novos itens serem agregados ao final. Pensar-se-ia que a maneira mais fácil e rápida de o conseguir seria utilizar MOVE, como se explicou anteriormente. Contudo, MOVE copiaria o ficheiro «numtel» completo, incluindo os três caracteres CHR$0 que assinalam o seu fim. É evidente que, para poder aumentar o ficheiro, se teria de não incluir os CHR$0 na cópia; é isto que a linha 1030 assegura.

Uma vez constituído o ficheiro «temp$$$», a parte principal do programa (linhas de 30 a 170) permite que se introduzam novos nomes e números de telefone nas três variáveis de texto S$ (apelido), I$ (iniciais) e T$ (telefone). Se a nova entrada está bem, é

escrita no ficheiro pela linha 160. Quando terminar a agregação, a linha 180 adiciona os três caracteres CHR$0 para assinalar o fim do novo ficheiro. Em seguida as linhas 200 a 230 mudam o nome de «temp$$$» para «numtel», restaurando assim a condição inicial da *cartridge*.

A primeira vez que este programa for executado para constituir uma lista telefónica, não funcionará, porque tenta ler dados de um ficheiro «numtel» que não existe. A solução é criá-lo, directamente, com

```
OPEN #4,"m";1;"numtel":
PRINT #4;CHR$ 0;CHR$ 0;CHR$ 0:
CLOSE #4
```

preparando, assim, a *cartridge* para o programa.

O segundo programa lê o ficheiro de nomes e números de telefone e procura por um apelido dado:

```
10 OPEN #4,"m";1;"numtel"
20 INPUT "Apelido";N$
30 INPUT #4;S$;I$;T$
40 IF S$=CHR$ 0 THEN CLOSE #4:
GO TO 10
50 IF S$<>N$ THEN GO TO 30
60 PRINT I$;" ";S$;" Tel ";T$
70 GO TO 30
```

Este programa surpreende pela sua simplicidade. A linha 10 abre o ficheiro e as linhas 30 a 70 lêem-no, à procura do apelido contido em N$. A linha 40 detecta o fim do ficheiro. Poderá admirar-nos o facto de CLOSE, na linha 40, estar seguido por um OPEN na linha 10. A razão é que, cada vez que se procura um nome, o ficheiro tem que ser lido, de novo, desde o início, o que é assegurado pelo comando OPEN.

Nada há a acrescentar a este exemplo a não ser apontar o facto de que o tempo para encontrar o número de telefone não depende muito do comprimento do ficheiro, porque, de cada vez que se procura um nome, não só se lê o ficheiro completo mas também toda a fita! O comando OPEN, que é essencial para que o ficheiro volte a ser lido, pesquisa a fita restante para regressar ao início do ficheiro. As consequências deste método de tornar a ler um ficheiro estudam-se mais cuidadosamente no próximo capítulo.

TRABALHO COM «MICRODRIVES»

Este capítulo descreveu o tipo de operações que as *microdrives* podem executar. Elas abrem todo um novo mundo de programação do *Spectrum*, e é importante não ignorar o desafio de produzir aplicações úteis e bem concebidas que aproveitem as suas características. A *microdrive* não se pode tomar como um dispositivo tradicional de armazenamento, porque, na realidade, não é mais do que uma unidade rápida de fita com um conjunto bem desenvolvido de extensões ao BASIC *ZX*. Com tal dispositivo, é decididamente possível conseguir tempos de resposta razoáveis e operação simplificada para o utilizador, embora seja necessário compreender profundamente tanto o funcionamento das *microdrives* como a aplicação a desenvolver.

CAPÍTULO 10

Princípios da «interface» 1 e das «microdrives»

No funcionamento da *interface* 1 e das *microdrives* há duas áreas principais com interesse. Em primeiro lugar, põe-se a questão de como pode a *interface* fornecer 8 k de rotinas ROM que agregam novos comandos ao BASIC *ZX* (e alargam alguns dos antigos), isto num *Spectrum* de 48 k com todos os endereços já ocupados! Em segundo lugar, há a considerar a forma como se alarga o sistema de canais e *streams* para acomodar as *microdrives*. Estes dois tópicos são estudados neste capítulo, e exploram-se algumas das muitas aplicações deles resultantes.

A PAGINAÇÃO DA ROM

É difícil alargar as rotinas em código máquina contidas na ROM de 16 k, onde está o BASIC *ZX*, porque a totalidade dos 64 k do *Spectrum* disponíveis para endereços já está reservada à memória RAM ou à ROM. Esta ocupação total dos endereços está a mostrar-se um problema corrente para os microcomputadores, conforme vão aumentando de complexidade. A solução habitual é utilizar a paginação! A paginação é uma técnica que permite usar o mesmo bloco de endereços para diferentes blocos de memória. É evidente que num instante dado, só se pode endereçar um único bloco da memória, o que implica que, para endereçar outros blocos de memória com os mesmos endereços, terá de haver uma maneira de chamar um novo bloco de memória depois de terminar o acesso a outro. Este procedimento é semelhante à leitura de um livro, na qual, com os mesmos olhos (bloco de endereços) se vai lendo (endereçando) páginas diferentes (blocos de memória). Daí o nome «paginação». Por exemplo, o micro da marca BBC usa a paginação para escolher uma de várias ROMs de 16 k, cada uma das quais pode conter uma aplicação ou linguagem diferente. A paginação também se utiliza no *Spectrum* para agregar a nova ROM de 8 k da *interface* 1. Num dado instante, ou está presente a ROM de 16 k habitual, contendo o BASIC *ZX*, ou se passa à nova ROM de 8 k.

Os circuitos electrónicos que possibilitam a paginação não são muito elaborados porque o *Spectrum* foi projectado com uma linha de desactivação da ROM, conduzida exteriormente, à conexão de expansão (ver capítulo 2). Mantendo uma voltagem de +5 volts nesta linha, obriga-se a ROM de 16 k a não responder a qualquer endereço, permitindo assim que outra ROM a substitua.

A forma como o *Spectrum* utiliza a paginação da ROM é muito diferente da maior parte dos outros microcomputadores, porque alarga os comandos do BASIC *ZX*, já existentes, utilizando as rotinas em código máquina na ROM de 8 k paginada. Isto implica que esta ROM paginada tenha que ser chamada e substituída por outra, automaticamente, enquanto está a ser executado um programa. Naturalmente, a pergunta que se põe é: como? Consideremos o que acontece quando o *Spectrum* encontra uma instrução que não faz parte do seu léxico habitual. Imediatamente, ele assinala um erro, fazendo um RST 8, para chamar a rotina de tratamento de erros na ROM do BASIC *ZX*. Se, ao ser detectado, este salto ao local 8 da memória for utilizado para ter acesso à nova página de ROM, as rotinas que ela contém podem verificar a forma do comando e ver se corresponde a alguma coisa que possa tratar. É isto o que realmente acontece. A *interface* 1 inspecciona continuamente o *bus* de endereços do *Spectrum*, verificando se ocorre o endereço 8. No caso afirmativo, imediatamente o interpreta como uma ordem para passar à página da nova ROM de 8 k. Assim, o comando

```
CAT 1
```

resultará num erro, fazendo um RST 8, se o *Spectrum* não estiver ligado à *interface* 1, mas se estiver ligado, o RST 8 fá-lo passar à página da nova ROM de 8 k, que executa a catalogação e, no fim, retorna ao BASIC *ZX* depois de limpar as *flags* que assinalam o erro. É evidente que se a instrução não for reconhecida pela nova ROM, esta devolve o erro à ROM do BASIC *ZX*, que então emite uma mensagem de erro.

A *interface* 1 também paginará a nova ROM se for usado o endereço 5896, porque este endereço pertence à rotina CLOSE da ROM onde está o BASIC *ZX*, e esta rotina deverá ser interceptada antes que o *Spectrum* tente fechar (com CLOSE) um canal da *microdrive*. A ROM de 16 k é repaginada pela nova ROM de 8 k,

utilizando o endereço 1792. Os métodos de paginação e a utilização da nova ROM de 8 k serão estudados mais adiante, neste capítulo.

O FORMATO DA INFORMAÇÃO NA «MICRODRIVE»

A *microdrive* é, essencialmente, uma unidade rápida de fita, em que a fita forma um anel sem princípio nem fim. Desta forma, qualquer ponto da fita pode ser lido ou gravado sem rebobinação. A informação é escrita e lida em duas pistas, para conseguir uma densidade suficientemente alta de armazenamento de dados. Os pormenores do formato físico exacto que se usa no armazenamento não são importantes, porque a *microdrive* é um dispositivo único na gama dos computadores Sinclair. No entanto, a organização da informação na fita é importante. Ao contrário do sistema normal de *cassettes*, a informação é armazenada na *microdrive* em blocos de dimensão fixa, conhecidos como «sectores».

O facto de chamar a um «sector» um «bloco de dados» é somente aflorar o problema. É melhor pensar num sector com uma área de fita onde se pode armazenar informação. Quando se dá um formato a uma *cartridge*, com FORMAT, escrevem-se na fita tantos sectores quantos sejam possíveis. No início todos estes sectores estão assinalados como livres para utilização; quando se escreve informação na fita, os sectores utilizados vão sendo marcados. Para ajudar a visualizar tudo isto, pensemos num sector livre como contendo dados arbitrários sem qualquer interesse, e num sector utilizado como aquele que contém informação necessária para nós. Na realidade, o comando FORMAT assinala alguns dos sectores que cria como já utilizados, porque estão numa parte da fita que não está boa — ou porque está perto da união das duas pontas da fita para formar o anel ou porque tem uma parte danificada por qualquer outra razão.

Como o sector é a unidade fundamental de armazenamento nas *microdrives*, vale a pena, evidentemente, examiná-lo mais minuciosamente.

O FORMATO DE UM SECTOR

Cada sector de fita é constituído por duas partes: um bloco de cabeçalho e um bloco de dados. A finalidade do cabeçalho é

identificar qual é o sector que passa, nesse momento, na cabeça de leitura. O formato do cabeçalho é o seguinte

12 *bytes* de sinal inicial
1 *byte* da *flag*
1 *byte* do número do sector
2 *bytes* não utilizados
10 *bytes* para o nome da *cartridge*
1 *byte* para a soma de verificação

É importante verificar que o fim único do cabeçalho é marcar a posição nesse instante na fita e, neste sentido, o seu componente mais importante é o número de sector, ocupando 1 *byte*. Quando o comando FORMAT cria os sectores, atribui a cada um um único número de sector, entre 0 e 255. No entanto, nem todos esses números de sector existem numa dada *cartridge*, quando a fita não é suficientemente comprida. Os blocos de cabeçalho são lidos tanto por operações de leitura como por operações de escrita, mas a única operação que os cria é um FORMAT. Os blocos de cabeçalho formam «marcos» inalteráveis para os blocos de dados que os seguem.

Visualiza-se melhor em duas partes o formato de um bloco de dados: um registo descritivo que armazena informação acerca dos dados seguintes e um registo que contém dados úteis. O formato em pormenor de um bloco de dados é o seguinte:

descritor do registo

12 *bytes* de sinal de início
1 *byte* de *flag*
1 *byte* para o número do registo
2 *bytes* para o comprimento do registo
10 *bytes* para o nome do ficheiro
1 *byte* para a soma de verificação

registo

512 *bytes* de dados
1 *byte* para a soma de verificação

Note-se que o descritor do registo tem o mesmo formato que um bloco de cabeçalho, e por isso é lido pelas mesmas rotinas. Contém certo número de elementos informativos essenciais à organização de sectores em «ficheiros com nome».

Um ficheiro com nome é uma colecção de sectores. O nome do ficheiro é armazenado em cada sector em dez *bytes*, reservados para isso no descritor de registo. Há um número de registo de 1 *byte* que dá a ordem pela qual devem ser tomados os sectores para constituírem um ficheiro. Por exemplo, tomemos um ficheiro de cinco sectores: o primeiro será o registo 0, seguido pelo registo 1 e assim por diante até ao registo 4. Note-se que o número de registo não tem nada a ver com o número do sector onde os dados estão armazenados. Por exemplo, o registo 0 estará armazenado no sector 57, o registo 1 no sector 66, etc. A complicação desta imagem simples é a possibilidade de uma área de dados dum sector não estar completamente utilizada. À medida que se cria um ficheiro, só se escreve um novo sector quando estiver cheio um *buffer*, e por isso só acontecerá escrever-se um *buffer* incompleto no fim dum ficheiro. Os dois *bytes* do comprimento do registo são utilizados para guardar o número de *bytes* da área de dados que realmente contém informação. Todos os registos de um ficheiro terão de comprimento 512 *bytes*, excepto o último, que, aliás, poderá ter também a mesma medida.

MAPAS DA «MICRODRIVE»

Há um problema fundamental com o formato do sector utilizado nas *microdrives*, que não é evidente para nós até que visualizemos como se escrevem os sectores durante a criação do ficheiro. Eis o problema: como é que se sabe se um sector, que está quase a passar por baixo da cabeça de leitura/gravação, está livre ou já utilizado? O cabeçalho nunca é reescrito, por isso não pode ser utilizado para guardar a mudança de estado de um bloco de dados que tenha passado a fazer parte de um ficheiro, ou sido libertado por uma operação ERASE. Pensar-se-ia que o próprio bloco seria o lugar óbvio para armazenar a informação acerca de quando está ou não libertado um bloco de dados. E é sem dúvida o único local onde se poderia guardar tal informação variável, mas a sua utilização acarretaria um novo problema. Devido a problemas de controle de tempos, a *microdrive* só pode tornar a escrever um bloco completo

de dados. Suponhamos que existe um *buffer* completo pronto para ser escrito num sector livre. A programação própria lê os descritores do registo à medida que passam debaixo da cabeça de leitura, de modo a descobrir se o próprio bloco de dados está livre. Quando encontra um livre, já é demasiado tarde para começar a escrever os dados: a primeira parte do bloco (o descritor do registo) já passou pela cabeça de leitura, e não há forma de escrever num fragmento de um bloco de dados. Uma solução seria esperar até que o cabeçalho do bloco livre, que se acabou de identificar, desse a volta outra vez! Isto significaria que cada operação de gravação envolveria, pelo menos, uma pesquisa por toda a fita, e globalmente a operação seria demasiado vagarosa. Na solução adoptada no *Spectrum*, constrói-se um mapa de *microdrive* que mostra que sectores de uma *cartridge* estão livres e quais estão já utilizados. O mapa de *microdrive* é constituído por um bloco de 32 *bytes*, onde cada um dos 256 sectores teoricamente possíveis é representado por um único algarismo binário ou *bit*. Se o *bit* representativo de um sector for igual a 1, isso significa que o sector já está utilizado ou que ele não existe nessa *cartridge*. Por outro lado, se o *bit* representativo de um sector for igual a 0, o sector está livre e pode ser utilizado para fazer parte de um ficheiro. Verificar-se-á que, dado o mapa de uma *microdrive*, o *Spectrum*, lendo simplesmente o número de sector no cabeçalho, detecta se um sector está livre, e neste caso fica pronto para voltar a escrever o bloco de dados completo. O mapa das *microdrives* é uma forma bastante perfeita de resolver o problema de saber se um sector está livre ou não. Tem que construir-se um novo mapa de cada vez que se abre um ficheiro, porque existe sempre a possibilidade de a *cartridge* ter sido mudada desde a última vez que se produziu o mapa. Durante operações com ficheiros, mantém-se em dia o mapa alterando o estado dos *bits* que representam quaisquer dos sectores utilizados. Assim, ao utilizar mapas de *microdrive* basta ter em consideração o tempo necessário para uma leitura completa da fita por cada comando OPEN e os 32 *bytes* de memória necessários ao armazenamento do próprio mapa.

O CANAL DA «MICRODRIVE»

A última componente que é necessário compreender é o canal da *microdrive*. Reportando-nos à descrição dos canais e *streams* do capítulo 5, verificaremos que para alargar esse sistema de forma a

incluir ficheiros na *microdrive* basta introduzir um novo tipo de registo de canais ou descritor de canais. Na realidade, é também necessário alargar a programação que trata dos *streams* e dos canais, mas isso consegue-se pela adição da nova ROM de 8 k.

Um descritor de canais, relativo a ficheiros nas *microdrives*, tem o formato seguinte:

byte	nome	conteúdo
0		0008 — endereço das rotinas de erro
2		0008 — endereço das rotinas de erro
4		identificador do canal «M»
5		endereço da rotina de saída
7		endereço da rotina de entrada
9		595 — comprimento do descritor de canais
11	CHBYTE	*byte* seguinte do registo
13	CHREC	número do registo actual
14	CHNAME	nome do ficheiro com 10 *bytes*
24	CHFLAG	*flag* b0=0 aberta para leitura
25	CHDRIV	número da unidade
26	CHMAP	endereço do mapa da *microdrive*
28		12 *bytes* de sinal de início
40	HDFLAG	*flag* do cabeçalho b0=1
41	HDNUM	número do sector
42		não é utilizado
44	HDNAME	nome da *cartridge*
54	HDCHK	soma de verificação do cabeçalho
55		12 *bytes* de sinal de início
67	RECFLG	*flag* do registo b0=0
68	RECNUM	número do registo
69	RECLEN	número de *bytes* no registo
71	RCNAM	nome do ficheiro 10 *bytes*
81	DESCHK	soma de verificação do descritor do registo
82	CHDATA	*buffer* de dados com 512 *bytes*
594	DCHK	soma de verificação dos dados

Há muitas características interessantes neste descritor de canal. Vamos considerá-lo dividido em três partes. Os *bytes* de 0 a 27 formam um conjunto de informação geral sobre o canal, os *bytes* de

28 a 54 formam a parte do cabeçalho de um sector e os *bytes* de 55 a 594 formam um bloco de dados. O facto de um descritor de canal conter informação formulada em termos do cabeçalho de um sector e de um bloco de dados não é acidental. Quando se está a gravar ou a ler um determinado sector, o cabeçalho é armazenado nos *bytes* de 28 a 54 e o bloco de dados nos *bytes* de 55 a 594. Neste sentido, a última parte do descritor do canal é uma imagem da memória, ou cópia do sector na fita.

A primeira parte do descritor do canal tem aproximadamente o mesmo formato que os registos de canais apresentados no capítulo 5. Na realidade, a programação relativa a canais e *streams* ainda trata as primeiras quatro posições de memória como os endereços das rotinas de entrada e saída a utilizar com o canal. Como estas posições armazenam agora o endereço 8, a rotina de erros, qualquer tentativa para utilizar as rotinas de entrada e saída, relativas ao canal, faz com que a nova ROM de 8 k seja repaginada. Quando isto acontece, as rotinas da ROM-sombra utilizam as quatro posições seguintes ao identificador do canal (*byte* 4) como o endereço das rotinas de saída e entrada contidas na nova ROM de 8 k. (Note-se que as primeiras quatro posições de memória podem ainda ser utilizadas para guardar os endereços das rotinas de saída e entrada da ROM de 16 k ou noutra parte qualquer da RAM. Explora-se esta ideia no capítulo final.)

O *byte* 9 guarda o comprimento do descritor completo do canal: no sistema alargado, os descritores de canal podem ter comprimentos diferentes, e será necessário a programação percorrer a área de memória relativa ao canal.

A localização do *buffer* descrita no capítulo anterior situa-se agora no fim do descritor de canal do respectivo ficheiro. Enche-se ou esvazia-se este *buffer* à medida que enviamos ou retiramos dados do ficheiro. Os *bytes* 11 e 12, CHBYTE, servem como um ponteiro indicador do *byte* seguinte, a agregar ou eliminar. Quando se procura agregar o 513.º *byte*, todo o bloco de dados é enviado para fora do *buffer*. Se se pede um 513.º *byte*, será lido o registo seguinte do ficheiro para o descritor do canal.

Vale ainda a pena descrever mais um *byte*, o 67, RECFLG. Este *byte* regista se o bloco que está a passar sob a cabeça é um bloco de dados (b0=0); mas guarda também mais uma ou duas informações. Se b1 é igual a 1, o registo que se acabou de ler é o último registo do ficheiro. Isto é, b1 é uma *flag* de fim de ficheiro. O *bit* 2 é igual a 1

se o ficheiro que se está a ler não é um ficheiro tipo PRINT, isto é, se foi criado por um comando SAVE*.

SUMÁRIO

Tendo sido descritas todas as características importantes da operação de uma *microdrive*, será talvez útil sumariá-las:

1) A informação é armazenada na fita com a forma de blocos de comprimento fixo, chamados "sectores".
2) Cada sector é composto de duas partes principais: o cabeçalho, que contém o número de sector e que não é alterado durante operações normais, e o bloco de dados, que contém o nome do ficheiro, número de registo e os 512 *bytes* de dados que cada sector pode armazenar.
3) O mapa de uma *microdrive* serve para descobrir se um sector está livre ou se já foi utilizado. O mapa é feito de cada vez que se abre (OPEN) um ficheiro, ao percorrer a fita inteira, e depois mantém-se em dia esse mapa à medida que se usam novos sectores.
4) O descritor de canal, relativo a ficheiros de uma *microdrive*, contém a mesma informação que um sector, mais uma certa informação relativa à constituição do ficheiro.

A melhor maneira de assegurar que este método de operação foi compreendido é apresentar exemplos, nas secções seguintes.

UM PROGRAMA PARA LISTAR REGISTOS/SECTORES

É bastante fácil saber quais os sectores que foram utilizados para armazenar os registos de um ficheiro. Basta ler todo o ficheiro e encontrar, com PEEK, o número do sector armazenado no descritor de canal, relativo ao ficheiro, todas as vezes que se lê um registo novo.

Antes de o executar, deve-se responder a duas questões:

1) Onde está armazenado o descritor de canal, referente a esse ficheiro?
2) Como se pode forçar a leitura de um novo registo?

O capítulo 5 dá a solução ao primeiro problema. O endereço de qualquer descritor de canal pode ser encontrado examinando o elemento correcto da tabela de *streams*. Se o descritor de canal foi aberto para o *stream* S, o endereço do início do descritor pode ser encontrado utilizando a sub-rotina seguinte:

```
1000 LET A=23574+2*S
1010 LET C=PEEK A+256*PEEK (A+1)
1020 LET D=PEEK 23631+256*PEEK 2
3632
1030 LET C=C+D-1
1040 RETURN
```

A linha 1000 calcula o endereço do elemento da tabela de *streams*; esse elemento dá o deslocamento, em relação à origem da área da memória reservada ao canal, do canal associado com o *stream* S. A linha 1010 armazena esse deslocamento em C, a linha 1020 guarda o endereço onde se inicia a área dos canais em D, e a linha 1030 utiliza toda esta informação para calcular o endereço do primeiro *byte* do descritor do canal e guarda-o em C.

O segundo problema resolve-se facilmente. Um INPUT # naquele *stream* fará com que seja lido um novo registo para o *buffer* depois de ter sido processada toda a informação nele contida. Os *bytes* 11 e 12 do *buffer* CHBYTE actuam como se fossem um ponteiro *(pointer)* indicando o *byte* seguinte a ser tirado do *buffer* e para satisfazer o INPUT #. Se for introduzido, com POKE, um valor suficientemente grande (>512) o programa de controle do *Spectrum* actuará como se o *buffer* já tivesse sido utilizado e lerá um novo registo. Assim:

```
POKE C+12,5
INPUT #S;A$
```

implicará sempre que seja lido um novo registo para o *buffer*, desde que C contenha o endereço do primeiro *byte* do descritor do canal. Tendo resolvido os dois problemas, o programa é fácil:

```
10 OPEN #4,"m";1;"grande"
20 LET S=4: GO SUB 1000
30 PRINT "registo ";PEEK (C+88
)
40 PRINT "sector ";PEEK (C+41)
50 POKE C+12,5
60 INPUT #4;A$
70 GO TO 30
```

A linha 20 utiliza a sub-rotina 1000 para armazenar, em C, o endereço do descritor do canal que está associado com o *stream* 4. As linhas 30 e 40 imprimem então o número do registo e do sector, pesquisando, com PEEK, os *bytes* apropriados do descritor do canal, e finalmente as linhas 50 e 60 obrigam à leitura de um novo registo.

Quem utilizar este programa numa *cartridge* que contenha só um ficheiro verificará que os registos não estão armazenados em sectores sequenciais. Por exemplo, o registo 0 pode estar armazenado no sector 20, o registo 1 no sector 22, o registo 2 no sector 24, e assim por diante. A explicação reside em que os sectores passam sob a cabeça da *microdrive* mais depressa do que podem ser recolhidos os dados no *buffer*: os sectores por cima dos quais se «passou» representam oportunidades perdidas!

CONSULTA DO MAPA

Os mapas das *microdrives* são armazenados dentro da região de memória que começa em 23792 (ver secção seguinte). O endereço exacto onde está armazenado o mapa pode ser calculado pelo exame dos *bytes* 26 e 27 (CHMAP) do descritor de canais; com esta informação é fácil imprimir o modelo binário de forma a ver as posições dos sectores livres e dos dados já utilizados. Veja-se o programa seguinte:

```
 10 OPEN #4,"m";1;"b"
 20 LET S=4: GO SUB 1000
 30 LET M=PEEK (C+26)+256*PEEK
(C+27)
 40 FOR I=0 TO 31
 50 LET B=PEEK (M+I)
 60 FOR J=1 TO 8
 70 PRINT B=2*INT (B/2);
 80 LET B=INT (B/2)
 90 NEXT J
100 NEXT I
110 STOP
```

A linha 30 armazena, em M, o endereço do mapa da *microdrive*. As linhas de 40 a 90 imprimem, então, os 32 *bytes* como se fossem um *stream* contínuo de algarismos binários de forma averiguar-se quais são os sectores utilizados ou não.

Executando-se este programa e se já existir o ficheiro «b», notar-se-á qualquer coisa de estranho no mapa, visto que, embora todo o comando OPEN construa um mapa da *microdrive*, este só se mantém depois de pesquisar até ao fim da fita e concluir que o ficheiro é um ficheiro de escrita.

CANAIS «AD HOC» E FICHEIROS NÃO PRINT

A maioria dos comandos do BASIC *ZX*, como MOVE, SAVE, etc., necessita de utilizar um descritor de canais. Estes descritores, criados pelos comandos e destruídos depois da operação, chamam-se «canais *ad hoc*». A única diferença entre eles e um descritor normal, como os que um comando OPEN cria, é que o identificador, M, do canal (no *byte* 4) é substituído por CHR$(205), ou seja, CHR$ (CODE («M»)+128).

Outra característica dos ficheiros das *microdrives* é a sua classificação em ficheiros criados por um PRINT ou, mais abreviadamente, ficheiros PRINT e ficheiros que «não» foram criados por um PRINT, ou ficheiros não PRINT. Os ficheiros PRINT são cirados pelos comandos OPEN, PRINT e CLOSE, e os ficheiros não PRINT são criados por SAVE. A única diferença entre estes dois tipos de ficheiro, PRINT e não PRINT, é que os ficheiros não PRINT armazenam certos itens de informação, acerca da sua natureza, no registo 0 do ficheiro. Para ser mais preciso:

registo 0 do ficheiro

byte 1	*flag* 0= BASIC 1/2= dados em vectores 3= código máquina
bytes 2 e 3	número de *bytes* de dados no ficheiro
bytes 4 e 5	endereço do início
bytes 6 e 7	comprimento da área do programa
bytes 8 e 9	número de linha de execução automática

Reconhece-se a semelhança entre este formato e o formato do cabeçalho da fita descrito no capítulo 8. Mesmo tomando em conta esta diferença, não há razão para que um ficheiro não PRINT não possa ser lido, utilizando INKEY$, como uma sequência de caracteres ASCII. Mas as rotinas de controle do *Spectrum* estabele-

cem uma distinção nítida entre ficheiros PRINT e não PRINT, e não permitem abrir, com OPEN, um ficheiro não PRINT. É pena, porque a possibilidade de ler e escrever em ficheiros constituídos por programas daria outra dimensão ao tratamento de dados pela *microdrive*. É possível enganar o sistema e fazê-lo criar ficheiros não PRINT, utilizando instruções PRINT pela introdução, com POKE, de 4 no *byte* 67 (RECFLG) do descritor de canal imediatamente depois de abrir (OPEN) o ficheiro. O valor de RECFLG é a única maneira que o sistema tem de reconhecer um ficheiro não PRINT. Fazendo-o teremos de ter a certeza de escrever a informação, acima descrita, no primeiro registo antes de escrevermos o programa ou outras informação.

AS NOVAS VARIÁVEIS DO SISTEMA

Quando se pagina a nova ROM de 8 k pela primeira vez, criam-se mais 58 variáveis de sistema que são necessárias à sua operação. Estas variáveis são acrescentadas no final da área habitual das variáveis de sistema e ocupam parte da área da memória reservada para os mapas da *microdrive* no *Spectrum* não expandido. Em vez de dar uma lista completa de todas as novas variáveis (que pode ser encontrada no apêndice 2 do manual da *interface* 1 e das *microdrives*) é melhor descrever somente as que poderão ter utilidade. Algumas das variáveis do sistema referem-se a outras características da *interface* 1 e serão explicadas nos capítulos seguintes. Algumas servem como áreas temporárias de trabalho para os comandos alargados e rotinas em código máquina da nova ROM de 8 k. Estas serão descritas à medida que vão sendo necessárias na secção sobre a utilização de código máquina. Depois de tomar tudo isto em consideração só restam duas variáveis novas com interesse e que nada têm a ver com as *microdrives*!

IOBORD (23750)

Define simplesmente a cor da moldura, intermitente, durante qualquer I/O controlado a partir da *interface* 1. Pode-se introduzir, com POKE, qualquer código de cor desejado nesta variável para alterar, ou até eliminar, a intermitência da moldura.

COPIES (23791)

O valor armazenado nesta variável de sistema define o número de

cópias de um ficheiro criado por um comando SAVE*. Se fizermos mais do que uma cópia do ficheiro, ocuparemos mais espaço, mas aumentaremos a velocidade do carregamento. Cada cópia feita terá de ser apagada (ERASE) separadamente, isto é, três cópias de um programa levarão três comandos ERASE antes de o programa se perder para sempre!

UTILIZAÇÃO DE LINGUAGEM ASSEMBLY

A utilização das rotinas da nova ROM de 8 k a partir de linguagem assembly pode ser muito difícil excepto por um mecanismo especial de chamada. A ROM de 8 k é paginada pela ROM do BASIC *ZX* quando uma instrução RST 8 chama a rotina de erros. O código de erro é guardado na posição de memória que segue à instrução RST 8 e esta é examinada pela ROM de 8 k para calcular o tipo de erro causador da sua paginação. No entanto, os códigos de erro só utilizam um intervalo limitado, e este facto tem sido utilizado para que programas em linguagem assembly chamem as rotinas da *microdrive* armazenadas na nova ROM de 8 k.

O programa seguinte

```
RST 8
DEF B código
```

chamará uma rotina particular da nova ROM de 8 k, relativa à *microdrive*, que dependerá do valor atribuído a «código», de acordo com a lista seguinte:

código	resultado
33	arrancar com o motor de uma *microdrive* (o registo A contém o número da unidade; se o registo A contiver zero, desligar-se-ão os motores de todas as unidades *microdrives*)
34	abrir (OPEN) um canal *ad hoc*
35	fechar (CLOSE) um ficheiro
36	apagar (ERASE) um ficheiro
37	ler registo seguinte de um ficheiro PRINT
39	ler um dado registo de um ficheiro PRINT (o registo é especificado no descritor de canal)
40	ler um sector de um ficheiro PRINT (o sector lido será o especificado em CHREC)

41 ler sector seguinte na fita
(o próximo sector que passe sob a cabeça de leitura será carregado para o descritor de canal)

42 escrever um sector
(o número do sector é armazenado em CHREC no descritor de canal)

As rotinas correspondentes aos códigos 34 e 36 — OPEN e ERASE — utilizam as novas variáveis do sistema D-STR1 (23766) e N-STR1 para guardar o número da unidade e o nome do ficheiro, respectivamente. As duas primeiras posições de N-STR1 (23770) guardam o comprimento do nome do ficheiro, e as duas últimas (23772) guardam o endereço da primeira letra do nome do ficheiro. No retorno da operação OPEN, o endereço do descritor de canal está no registo IX e o seu deslocamento (como armazenado na tabela de *streams*) está no registo DE. As rotinas restantes estão preparadas para que o endereço do descritor de canal esteja no registo IX.

Ao utilizar quaisquer das rotinas, é necessário cautela com o facto de nenhum dos registos estar guardado, e também que não se pode prever, em geral, o estado das interrupções desarmáveis. Antes de utilizar estas rotinas, é boa prática guardar o par de registos HL e desligar as interrupções. No regresso, haverá que restaurar o par de registos HL e ligar as interrupções.

UM COMANDO PARA REBOBINAR

Como exemplo de utilização das rotinas da nova ROM de 8 k considere-se o problema de conseguir uma operação de «rebobinagem». Neste contexto, rebobinar um ficheiro significa passar do registo que se está a consultar nesse momento para o registo 0. É possível, com a operação «ler registo» (código 39), ter acesso a qualquer registo de um ficheiro no *buffer* do descritor de canal. A rotina seguinte, em linguagem assembly, põe esta operação à nossa disposição na forma de uma função USR:

endereço	linguagem assembly	código	observações
23296	PUSH HL	229	guardar HL
23297	LD IX, canl	221,33,0,0	carregar IX com endereço do canal
23301	DI	243	desligar interrupções
23302	RST 8	207	ler registo
23303	39	39	código
23304	XOR A	175	limpar A
23305	RST 8	207	desligar motor
23306	33	33	código
23307	EI	251	ligar interrupções
23308	POP HL	225	restaurar valor em HI
23309	RET	201	regressar ao BASIC

Para utilizar esta rotina, haverá que introduzir, em 23299 e 23300, com POKE, o endereço do descritor de canal. O outro comentário a fazer é que o código 33 serve para parar o motor depois da leitura. Esta rotina, para ler um registo qualquer, pode utilizar-se vantajosamente no programa sobre números de telefone, apresentado no final do capítulo anterior. O método aí utilizado para voltar a ler o ficheiro era fechar, com CLOSE, o *stream* e reabrir, com OPEN, o canal. E abrir o canal significa que todos os sectores da fita têm que ser lidos para constituir o mapa duma *microdrive*. Ora bem, poupa-se tempo evitando o comando OPEN se for utilizada a rotina apresentada para passar da leitura dum registo qualquer à leitura do registo 0, ou seja, «rebobinar» o ficheiro. A utilização destes conceitos conduz às seguintes modificações do programa:

```
    5 GO SUB 1000

  40 IF S$=CHR$ 0 THEN GO SUB 20
00: GO TO 20

1000 DATA 229,221,33,0,0,243,207
,39,175,207,33,251,255,201
1010 FOR A=23296 TO 23309
1020 READ D
1030 POKE A,D
1040 NEXT A
1050 RETURN
```

```
2000 LET S=4
2010 LET A=23574+2*S
2020 LET C=PEEK A+256*PEEK (A+1)
2030 LET D=PEEK 23631+256*PEEK 2
3632
2040 LET C=C+D-1
2050 POKE 23300,INT (C/256)
2060 POKE 23299,C-INT (C/256)*25
6
2070 POKE C+13,0
2080 LET A=USR 23296
2090 POKE C+11,0
2100 RETURN
```

A sub-rotina 1000, em BASIC *ZX*, carrega a rotina em linguagem assembly que acabámos de descrever, isto é, a operação de rebobinagem. A sub-rotina 2000 prepara o acesso a essa rotina. As linhas de 2000 a 2040 põem em C o endereço do descritor do canal, utilizando métodos anteriormente tratados. As linhas 2050 e 2060 introduzem, com POKE, aquele endereço na rotina «ler qualquer registo». A linha 2070 põe o número de registo a zero de forma a que a linha 2080 carregue este sector no descritor. A linha 2090 reajusta CHBYTE de modo a que o primeiro *byte* do *buffer* seja devolvido como resposta ao comando INPUT seguinte.

Note-se que este programa tem um tempo de execução inferior, porque já não há necessidade de ler a fita completa para construir o mapa.

FICHEIROS DE ACESSO DIRECTO

A rotina «ler qualquer registo» serve para ler um registo do ficheiro, numa ordem qualquer. Basta isto para implementar ficheiros de acesso directo. No entanto, como já referimos, a *microdrive* é essencialmente um dispositivo sequencial. Se lemos o registo 3 e depois desejarmos ler o registo 2, a fita não voltará «atrás». Na realidade a fita será lida até ao fim e depois desde o princípio até o registo 2 tornará a passar na cabeça de leitura. O caso mais adverso será a necessidade de ler um ficheiro do fim para o princípio, porque para encontrar cada registo é preciso ler a fita completa! Como o tempo para ler toda a fita é relativamente curto (um ou dois segundos) não há tanta justificação para considerar o uso de técnicas de acesso directo nas *microdrives*. Os melhores tempos de processamento conseguem-se lendo os registos na sua sequência normal e processando o menor volume de dados possível. Por exemplo, uma

organização possível para a lista de números telefónicos é o acesso directo, com um registo do ficheiro associado a cada letra do alfabeto. Assim, encontrar-se-ia um número de telefone lendo o registo que guardasse todos os nomes que começassem pela mesma letra. Em média, e utilizando a rotina «ler qualquer registo», teria que ser lida metade da fita para encontrar o registo desejado. Se o ficheiro fosse lido sequencialmente, registo a registo, para localizar a posição desejada, a quantidade média de fita lida seria aproximadamente a mesma. No entanto, se cada item de dados de cada sector fosse processado da mesma forma, o tempo necessário para ler o ficheiro sequencialmente seria muito maior. A utilização de leitura sequencial em conjunção com o salto forçado dos sectores irrelevantes (ver o exemplo de listagem de sector/registo, dado anteriormente neste capítulo) é provavelmente tão rápido como qualquer método de acesso directo.

A HISTÓRIA DA «INTERFACE 1» E NÃO TERMINA AQUI

Neste capítulo descrevem-se os princípios da paginação da ROM e das *microdrives*, mas ainda há que considerar outros aspectos referentes à *interface 1*. Existem ainda outros dois tipos de canais, outras rotinas na ROM de 8 k e uma forma de ampliar o BASIC *ZX* que serão analisados nos dois últimos capítulos.

CAPÍTULO 11

«Interface» e comunicação

Este capítulo trata das outras duas características introduzidas pela *interface 1* — o porto *RS232* e a rede. Ambas são, na realidade, diferentes formas de comunicação em série. O porto *RS232* permite a ligação do *Spectrum* a uma impressora *standard* e também serve de comunicação com outros computadores. A rede é principalmente uma maneira para estabelecer comunicações entre vários *Spectrums*, mas é possível escrever programas que alarguem a rede de modo a incluir outros computadores. A primeira parte do capítulo trata do porto *RS232*, e de algumas das dificuldades ao utilizá-lo, e a segunda parte descreve a rede *Spectrum*. Muito do que se dirá acerca do seu funcionamento baseia-se em conceitos introduzidos no capítulo anterior.

«RS232» – QUASE UMA NORMA

Há um certo número de modalidades para passar informação entre computadores. Algumas até têm normas específicas, mas pouquíssimas são adoptadas na prática. É raro poder ligar dois computadores entre si, ou um computador e um periférico, por exemplo uma impressora, de forma a que funcionem de imediato. Normalmente basta um ou dois minutos para resolver a dificuldade; outras vezes leva-se mais tempo; e chega a ser necessário usar o ferro de soldar. No pior dos casos mostra-se impossível estabelecer a ligação, o que é, no entanto, raro.

RS232 é uma norma para *interfaces* em série com muitas especificações e grande pormenor. A principal das incompatibilidades entre diferentes ligações *RS232* assenta no facto de implementar diferentes partes da mesma norma. Por exemplo, a ligação *RS232* mais simples é formada por três cabos: um para o sinal vindo do computador, outro para o sinal enviado ao computador e o último para ligação à terra. O *Spectrum* utiliza mais dois cabos, um para assinalar que está pronto para receber dados e outro para assinalar que o dispositivo ao qual está ligado pode começar a receber dados. Estas duas ligações, muitas vezes incluídas na ligação *RS232*, chamam-se «linhas *handshake*». No entanto, incluem-se habitualmente outras ligações para assinalar, por exemplo, se o dispositivo na outra extremidade do cabo está ligado à corrente eléctrica. Muitas ligações *RS232* só têm uma das duas linhas *handshake* utilizadas pelo *Spectrum*. Todas estas variações têm como resultado criar dificuldades para saber o que se deverá ligar e a quê.

Claro que os conselhos sobre a ligação de um novo dispositivo ao *Spectrum* dependem tanto do que seja esse dispositivo como do próprio *Spectrum*. O que significa que a única maneira de abordar os problemas resultantes do uso da *RS232* é compreender o essencial do seu funcionamento.

A «RS232» DO «SPECTRUM»

Os nove terminais da ficha que transporta os sinais da *RS232*, localizada na parte de trás da *interface* 1, são:

número de terminal	utilização
1	desligado
2	TX — dados entrando no *Spectrum*
3	RX — dados saídos do *Spectrum*
4	DTR — sinal de «pronto» enviado ao *Spectrum*
5	CTS — sinal de que o *Spectrum* está «pronto»
6	desligado
7	terra
8	desligado
9	+9 volts

Examinando esta lista, verifica-se que TX (linha para entrada de dados) se associa a CTS (linha de saída «pronto», enviada ao *Spectrum*) e que RX (linha de saída) emparelha com DTR (a linha de «pronto» para o *Spectrum*). Quando o *Spectrum* recebe dados, a linha CTS indica que está pronto para receber informação. O dispositivo ao qual está ligado o *Spectrum* não deve enviar-lhe dados enquanto CTS está «baixa» (isto é, a 0 volts). A informação enviada com CTS «baixa» será ignorada ou lida incorrectamente. O *Spectrum* também não transmitirá dados enquanto DTR for mantida «baixa» pelo dispositivo que receber esses dados.

Resumindo

1) O *Spectrum* só receberá dados correctamente quando CTS (terminal 5) estiver «alto» e deve utilizar-se este sinal para activar o dispositivo que transmite os dados.
2) O *Spectrum* só transmitirá dados quando DTR (terminal 4) estiver «alto» e este sinal deve ser utilizado pelo dispositivo receptor para indicar que está pronto para receber dados.

«HANDSHAKING» OU NÃO?

Compreendem-se facilmente as condições para transmitir e receber dados descritas na última secção, mas levanta-se uma dificuldade mesmo quando se tenta ligar duas máquinas idênticas entre si. Descrevemo-lo melhor ao dizer que a saída de uma máquina é a entrada da outra. Por exemplo, se desejarmos ligar dois *Spectrums* entre si, ligaremos o terminal 3 do primeiro ao terminal 2 do segundo. Ou seja, o sinal de saída de dados de um tem de entrar no terminal de entrada de dados do outro. Isto, evidente no caso de linhas de transmissão de dados — terminal 2 liga-se ao terminal 3 e terminal 3 liga-se ao terminal 2 —, é uma «ligação cruzada» que também se aplica às linhas *handshake*. Por outras palavras, as linhas DTR e CTS do primeiro *Spectrum* devem ser unidas respectivamente às linhas CTS e DTR do segundo *Spectrum*. Estas ligações cruzadas não são habituais. A maioria dos periféricos, impressoras, VDUs edt., já estão construídas de forma a considerar o cruzamento de ligações. Numa impressora, encontra-se o terminal 3, ainda chamado «dados RX», mas que agora é uma entrada de dados para esse aparelho. Neste caso é evidente que se deve unir o terminal 3 do *Spectrum* ao terminal 3 da impressora. O esquema do cabo de ligação *RS232* da Sinclair é o seguinte:

Spectrum	outro equipamento
dados TX	terminal 2 (dados TX)
dados RX	terminal 3 (dados RX)
CTS	terminal 5
+9V	terminal 6 (DSR)
terra	terminal 7, terra
DTR	terminal 20 DTR

e que trabalhará com a maioria das impressoras e VDUs.

De modo geral, ao construir-se um cabo que permita ao *Spectrum* funcionar com outro equipamento, é necessário conhecer pormenores acerca da disposição da ligação *RS232* desse outro equipamento. Supondo que ele utiliza uma ficha D, de 25 terminais, primeiro deve-se descobrir qual dos terminais, 2 ou 3, é o de saída (TX) do equipamento, e então uni-lo ao terminal 2 do *Spectrum*. O problema seguinte é decidir onde se devem ligar as linhas *handshake*. Habitualmente a DTR do *Spectrum* deve ser unida ou a DTR (terminal 20), DSR (terminal 6) ou RTS (terminal 4). O sinal CTS do *Spectrum* será unido a CTS (terminal 5) do outro equipamento.

Em alguns equipamentos não há linhas *handshake*. Como já se referiu, a ligação *RS232* mais simples só tem as ligações RX, TX e terra. Neste caso, a linha CTS do *Spectrum* (terminal 5) pode ficar desligada e DTR (terminal 4) deve ser unida a +9V (terminal 9): a não utilização de CTS significará que o dispositivo ao qual está ligado o *Spectrum* transmitirá dados sempre que queira, mesmo se o *Spectrum* não estiver pronto para os receber. DTR (terminal 4) une-se a +9V (terminal 9) para que o *Spectrum* transmita dados em qualquer momento. É claro que para isto funcionar o dispositivo emissor deve ser capaz de receber dados em qualquer instante! Na prática, encontram-se muitas vezes situações intermédias, com ligações *RS232* tendo, por exemplo, só uma linha CTS. Neste caso somente se ligarão as linhas *handshake* que existam e unirão as linhas de entrada restantes quer à terra quer a +9V, dependendo se elas estiverem em «baixa» ou «alta» para possibilitar transmissão ou recepção de dados.

Encontram-se, pois, muitos tipos diferentes de problemas com ligações *RS232*. Na prática as coisas não são tão más como se poderia esperar, e desde que se identifique a finalidade de cada ligação no outro dispositivo não haverá dificuldades. Muitas vezes é útil unir somente as linhas RX, TX e terra entre o *Spectrum* e o outro dispositivo, unindo as entradas das linhas *handshake* ou à terra ou a +9V de forma a que a ligação funcione sem linhas *handshake*. Pode-se então refinar a ligação unindo, uma a uma, as linhas *handshake*.

FORMATO DE DADOS «RS232»

A *RS232* é uma ligação em série, ou seja, quando se passa a

informação de um computador a um outro dispositivo ela é transmitida algarismo binário a algarismo binário (*bit* a *bit*). Embora a transmissão seja feita um *bit* de cada vez, é normal enviar um grupo de *bits* em sequência para representar um carácter. Há um certo número de escolhas quanto à maneira de transmitir os *bits*. Pode-se seleccionar uma das várias velocidades de transmissão ou taxas *baud*, isto é, o número de *bits* que se pode transmitir por segundo. Também se pode seleccionar quantos *bits* se transmitirão para representar um carácter único, quantos *bits* para assinalar o fim da transmissão de um carácter e se se vai enviar um *bit* de paridade para verificar erros de transmissão. No caso do *Spectrum* o formato usado na transmissão é

8 *bits* de dados
não existe *bit* de verificação da paridade
2 *bits* de «paragem»

e a taxa *baud* pode ser escolhida pelo utilizador. Assim, além de estabelecer a ligação eléctrica correcta entre o *Spectrum* e um outro equipamento, é também importante que esteja preparado para receber dados no formato do *Spectrum*. Na maioria dos casos isto somente envolve assegurar-se que tanto o *Spectrum* como o outro dispositivo utilizem a mesma taxa *baud*.

OS COMANDOS «RS232» DO BASIC

O BASIC *ZX* trata a ligação *RS232* como se fosse outro tipo de canal ou, para ser mais preciso, como dois novos tipos de canal. Os identificadores de canal utilizados são:

b ou B para canal *RS232*, binário, e
t ou T para canal *RS232*, de texto

Qualquer destes canais pode ser aberto a um *stream* da forma habitual. por exemplo,

OPEN #4, «b»

abrirá o *stream* 4 ao canal *RS232*, binário, e

OPEN #5, «t»

abrirá o *stream* 5 ao canal *RS232*, de texto. Uma vez associado um *stream* com um canal, podem utilizar-se os comandos I/O, já conhecidos — PRINT #,INPUT # e INKEY$ # — para receber e enviar dados.

Os dois canais b e t comportam-se da mesma maneira quando a informação transmitida ou recebida se compõe apenas de caracteres que se podem imprimir. A diferença vem da forma como elas tratam os códigos de controle do *Spectrum* e outras atribuições não normalizadas de códigos de caracteres. O canal b transmitirá o código de carácter completo de 8 *bits* do que lhe for impresso, com PRINT, mas o canal t somente enviará o código se for de um carácter imprimível ou se for capaz de converter o item numa sequência de caracteres imprimíveis. Ou seja,

PRINT #4; THEN

onde THEN, digitado com um único toque, envia CODE (THEN) ou 203 através do canal b. No entanto, se a *stream* 4 for aberta no canal t, serão enviados os códigos ASCII correspondentes às letras T, H, E, N.

As regras exactas são:

Na transmissão:

O canal b transmite o código de carácter com 8 *bits* de tudo o que lhe pedirem para imprimir, com PRINT.
O canal t não enviará os códigos de controle de 0 a 12 e 14 a 31 nem os códigos de caracteres gráficos de 128 a 164, e converterá todos os comandos de códigos entre 165 e 255 nas diversas letras ou caracteres ASCII que os compõem. O código 13 é substituído por 13, 10 (ver à frente).

Na recepção:

O canal b recebe o código completo de 8 *bits* que lhe enviem.
O canal t ignorará o oitavo *bit* de qual o código recebido, restringindo-o assim ao conjunto de caracteres ASCII de 0 a 128. Deve notar-se que o canal t é uma tentativa para reconciliar as extensões feitas no *Spectrum* ao conjunto de caracteres normalizados ASCII. Por exemplo, se a ligação *RS232* estiver ligada a uma impressora,

OPEN #4, «b»: LIST #4

listará o programa presente nesse momento; mas, como os coman-

dos serão enviados como um código de carácter único, ou não serão impressos ou farão com que a impressora faça alguma coisa estranha. Mas

OPEN #4, «t»: LIST #4.

dará uma listagem perfeitamente inteligível, porque os códigos dos comandos serão convertidos na sequência de caracteres que os representam habitualmente.

Como o canal b trabalha com todos os códigos de caracteres, serve para transmitir o conteúdo de posições de memória. Para o tornar mais fácil, os SAVE*, LOAD*, VERIFY* e MERGE* podem todos referir-se ao canal b. Por exemplo, são permitidos SAVE* «b» e LOAD* «b». É claro que sem programação especial, estes dois comandos somente permitem a troca de programas entre dois *Spectrums*. (A rede proporciona um método mais fácil de troca entre *Spectruns*, mas a ligação *RS232* tem a vantagem de servir para transferir programas através de uma linha telefónica com a ajuda de um *modem*.) Existe uma outra diferença importante entre os canais b e t: o tratamento do código de carácter ENTER. Como sempre, o canal b passa o código sem alterações, mas o canal t substitui cada ENTER (código 13) por uma sequência de dois caracteres 13, 10 que significa ENTER seguido de *line feed* [carácter ASCII de mudança de linha — CHR$(10)]. Algumas impressoras e VDUs começam automaticamente numa outra linha quando recebem um ENTER; outras necessitam também de um código de *line feed*. Se a impressora não precisa do código de *line feed* dará uma linha em branco entre cada uma das linhas de texto. Apenas se pode regular a impressora de maneira a não começar numa linha nova quando receber o código ENTER (isto é, se possível, desligar a mudança automática de linhas).

Note-se que os canais t e b também se podem utilizar com o comando MOVE. Por exemplo, para enviar dados da ligação *RS232* para o *écran* deve utilizar-se:

MOVE «b» TO #2

O AJUSTE DA TAXA «BAUD»

Já descrevemos os comandos para uso dos canais *RS232*, mas não falámos do método de ajustar a taxa *baud*. Quando se liga o

Spectrum à corrente eléctrica, a taxa *baud* é inicializada com o valor 9600. Para o alterar para outro valor deve utilizar-se

FORMAT «b»; baud

ou

FORMAT «t»; baud

onde *baud* tomará os valores 50, 110, 300, 600, 1200, 2400, 4800, 9600 ou 19200. Estes são os valores habituais da taxa *baud* encontrados em quase todo o equipamento de computação. No caso do *Spectrum*, a taxa *baud* pode ser interpretada aproximadamente como dez vezes o número de caracteres transmitidos por segundo. Assim 300 *baud* equivale aproximadamente a 30 caracteres por segundo. Como a operação *handshake* pára algumas vezes a transmissão de dados, a taxa real pode ser menor. Na maioria dos casos é aconselhável utilizar a taxa *baud* máxima que os dois equipamentos podem suportar. Uma taxa *baud* alta significa que os dados esperarão menos tempo para ser transferidos. No entanto, se não estivermos a utilizar as linhas *handshake*, o *Spectrum* não receberá dados correctamente para taxas *baud* acima de 300. Na realidade, nem sequer há a garantia de que receba todos os caracteres a 300 *baud*, mas sem a operação de *handshake* quanto mais devagar melhor!

Para definir uma taxa *baud* não habitual, introduziremos, com POKE, uma constante apropriada na variável de sistema, de dois *bytes*, chamada BAUD. A constante é dada por:

(3500000/(26*taxabaud))−2.

UTILIZAÇÃO CONJUNTA DE T E B

Não há contra-indicação para a utilização simultânea dos canais t e b. Por exemplo, muitas impressoras usam códigos ASCII, na região de 0 a 31, como códigos de controle para conseguirem impressões de efeitos especiais, como caracteres alargados ou caracteres gráficos. Não considerando estes códigos, trabalha-se melhor com a impressora através do canal t.

```
10 OPEN #4,"t"
20 OPEN #5,"b"
30 PRINT #5;CHR$ c;#4;A$
```

c é um código de controle para ser enviado, inalterado, à impressora e A$ contém texto.

PRINCÍPIOS DE OPERAÇÃO «RS232»

A ligação *RS232* é manuseada pelo sistema, já conhecido desde o capítulo 5, de canais e *streams* para I/O. A única característica nova é introdução do outro tipo de descritor do canal:

Byte	utilização
0	endereço 8 (rotina de erros)
2	endereço 8 (rotina de erros)
4	identificador de canal t ou b
5	endereço da rotina de saída
7	endereço da rotina de entrada
9	11 (comprimento do descritor de canal)

Este descritor é o mais simples e curto de todos os descritores recentemente apresentados. Por esta razão é o que se utiliza mais vezes ao dar exemplos de canais definidos pelo utilizador, como se verá no exemplo do capítulo 12. É de notar que não existe *buffer* de dados; por isso tanto na recepção como na emissão não há perdas de tempo, como sucedia nas operações das *microdrives*.

A ligação *RS232* do *Spectrum* tem interesse porque é constituída, na sua maior parte, por programação. A maioria das outras máquinas utilizam circuitos especiais que aceitam *bytes* de dados e os transmitem através duma *interface RS232* sem ajuda de qualquer espécie da UCP. O *Spectrum* não contém tais circuitos especiais: é a programação que cria os impulsos necessários na linha *RS232* da mesma forma que se criaram os impulsos de som e da *cassette* (ver capítulo 8). Neste caso, a porta de I/O é a 247, e a linha RX de saída é controlada pelo estado de b0. A leitura da porta 247 devolve o estado da linha de entrada de dados TX, também como b0. As duas linhas *handshake* estão associadas com a porta 239 de I/O. A leitura desta porta devolve DTR como b3, e o estado da linha CTS é dado por b4.

A sequência de operações necessária para enviar um *byte* de dados é a seguinte:

esperar até que a linha DTR esteja «alta»;
enviar o *byte*, algarismo binário a algarismo binário, utilizando o valor armazenado em BAUD para medir o comprimento de cada impulso.

A sequência de operações necessárias para receber um *byte* de dados é um pouco mais complexa:

Examinar o valor da nova variável do sistema SER-FLG, em 23751.

Se o valor em 23751 for diferente de zero, então a posição de memória seguinte, 23752, contém o carácter desejado. Depois de pôr 23751 a zero, este é devolvido como entrada (*input*).

Se o valor em SER-FLG for zero, pôr a linha CTS no estado «alto», depois esperar pelo início da transmissão, sinalizado pelo facto de a linha de dados TX ficar «alta» para o tempo de um impulso. Seguidamente são lidos os oito algarismos binários e pesquisam-se os dois *bits* de «paragem». Depois, pôr a linha CTS no estado «baixo» para impedir a transmissão de mais dados.

Se este procedimento não conseguiu parar a tempo o dispositivo transmissor, então será lido outro carácter completo e será armazenado em SER-FLG+1. SER-FLG será posta a 1 para indicar que já está à espera de um *byte* de dados. O primeiro *byte* lido é devolvido como entrada *(input)*.

A descrição das operações de INPUT RS232 revela que os caracteres lidos são, na realidade, postos num *buffer*, com o comprimento de um único carácter. Isto torna-se necessário porque alguns dispositivos emissores não respondem suficientemente depressa ao «abaixamento» da linha CTS que assegura que não será enviado um segundo carácter.

ASSEMBLER E A «INTERFACE» «RS232»

As rotinas contidas na nova ROM de 8 k que enviam e recebem dados utilizando a ligação *RS232* podem ser utilizadas em assembler. O método é basicamente o mesmo que o utilizado na chamada

da rotina da *microdrive* (ver capítulo anterior). Chamam-se as rotinas, utilizando

RST 8
código

onde a acção resultante depende do valor de «código».

código	acção resultante
29	Ler um *byte* da ligação *RS232*. Põe-se alta a *flag carry* se o *byte* for lido antes de tempo. O resultado é devolvido no registo «A».
30	Escrever o *byte* que estiver no registo A para a ligação *RS232*.

Há ainda três rotinas gerais de I/O que podem ser úteis ao escrever programas *RS232*.

código	acção resultante
27	Ler o teclado. Esperar até que seja premida uma tecla e devolver o seu código no registo A.
28	Escrever o carácter que está no registo A para o *écran* sem contar *scrolls*.
31	Escrever o carácter que está no registo A para a impressora *ZX*.
32	Testar o teclado. Regressar com a *flag carry* alta se se premiu alguma tecla.

Na secção seguinte apresenta-se um exemplo de utilização destas rotinas.

UM VDU COM O «SPECTRUM»

O principal problema que se põe quando se quer utilizar a ligação *RS232* para qualquer outro fim que não seja guiar uma impressora é a medição de tempos. Como todos os sinais são controlados por programação, as linhas *handshake* são absolutamente essenciais

para ter confiança na operação. Há momentos em que o *Spectrum* não é capaz de receber ou transmitir caracteres.

Considere-se, por exemplo, o problema de converter o *Spectrum* num VDU. Logicamente, o problema é bastante simples. Qualquer carácter recebido através da ligação *RS232* deve ser impresso (PRINT) no *écran*, e qualquer carácter digitado no teclado deve ser transmitido pela ligação *RS232*. Em BASIC *ZX* resultará

```
10 FORMAT "t";baud
20 OPEN #4,"t"
30 LET A$=INKEY$ #4
40 IF A$="" THEN GO TO 60
50 PRINT A$;
60 LET A$=INKEY$
70 IF A$="" THEN GO TO 30
80 PRINT #4;A$;
90 GO TO 30
```

Este programa funciona bastante bem desde que as linhas *handshake* estejam a ser utilizadas, mas, mesmo assim, é um pouco lento. Esta mesma ideia pode ser implementada em assembler assim:

endereço	linguagem assembly	código	observações
23296	LOOP RST 8	207	entrada RS232
23297	29	29	
23298	JR NC,SKIP	48,2	não há entrada
23300	RST 8	207	saída no *écran*
23301	28	28	
23302	SKIP RST 8	207	testar teclado
23303	32	32	
23304	JR NC,LOOP	48,246	não foi premida nenhuma tecla
23306	RST 8	207	ler tecla
23307	27	27	
23308	RST	207	saída RS232
23309	30	30	
23310	NKEY RST 8	207	testar se se deixou de premir tecla

23311	32	32
23312	JR C,NKEY	56,252
23314	JR LOOP	24,236

Basta utilizar um programa em BASIC que carregue o código que acabámos de apresentar:

```
10 FORMAT "b";1200
20 DATA 207,29,48,2,207,28,207
,32,48,248,207,27,207,30,207,32,
56,252,24,236
30 FOR A=23296 TO 23315
40 READ D
50 POKE A,D
60 NEXT A
70 LET A=USR 23296
```

A execução deste programa ainda deixa a desejar. O teclado funciona utilizando as rotinas de teste e de leitura do teclado, mas a repetição automática fica desactivada e ainda é possível ficar «bloqueado» na rotina de leitura das teclas. A solução é escrever uma «rotina de leitura e teste do teclado» especialmente para esse fim, que imite o comportamente de INKEY$. Um problema muito mais sério é o de as rotinas *RS232* tratarem a tecla SPACE como se fosse BREAK: habitualmente, carrega-se em CAPS SHIFT e BREAK para terminar um programa, mas durante operações *RS232* bastará carregar na tecla BREAK/SPACE. Este facto torna virtualmente impossível escrever qualquer programa de comunicações a sério, tal como o programa VDU apresentado acima, a não ser que se encontre uma maneira de gerar o código de carácter correspondente a SPACE sem carregar na tecla SPACE!

A ligação *RS232* do *Spectrum* é excelente para controlar impressoras e transferir programas entre o *Spectrum* e outros computadores, mas a sua utilização noutras situações requer um esforço considerável no desenvolvimento de programação adequada. O problema de SPACE se comportar como um BREAK pode ser resolvido em novas versões da ROM de 8 K. Ao autor, esse problema parece ser mais um erro do que uma característica propositada!

A REDE SINCLAIR

Enquanto a ligação *RS232* tem por finalidade possibilitar a comunicação entre dois dispositivos, a rede permite a transferência de dados entre um número qualquer de *Spectrums*. O método de comunicação utilizado pela rede tem o mesmo formato em série que o utilizado pela ligação *RS232*, mas existe um certo número de outras condições para tornar possível a comunicação múltipla. As características do equipamento foram modificadas de modo a permitir comunicações em via dupla através de um único par de cabos. Em cada instante, somente um dos *Spectrums* do grupo transmitirá dados, enquanto um certo número dos restantes os receberá, mas qualquer uma das máquinas poderá ser o emissor desses dados.

A programação fornecida com o equipamento inclui mais duas possibilidades. Em primeiro lugar, há um modo para que qualquer *Spectrum* «requeira» a rede e se converta no seu emissor. Em segundo lugar, com cada «pacote» de dados transmitidos há um endereço que identifica a qual das máquinas se destinam. Estas duas características da programação constituem uma espécie de «regra» para os *Spectrums* que vão utilizar a rede — em linguagem especializada, constituem o chamado «protocolo de comunicações». O protocolo de comunicações da rede Sinclair não é tão complexo como o de outras redes, como a Ethernet, mas é suficiente para muitas aplicações de computadores «em grupo», como seja a educação em informática e desenvolvimento de programas.

OS COMANDOS DE REDE EM BASIC

Os acrescentamentos ao BASIC *ZX* necessários para incluir redes seguem as linhas habituais dos feitos aos comandos de canal e *stream* para permitir a utilização de *microdrives* e da ligação *RS232*. Destes dois grupos, os comandos *RS232* são mais parecidos com os comandos de rede. O canal da rede é identificado por «N» ou «n» e ainda tem de identificar a estação com a qual se vai estabelecer comunicação. Para que esta identificação seja possível, a cada *Spectrum* ligado à rede tem de ser atribuído um «número de estação» que utiliza o comando.

FORMAT «n»;NUMEST

onde «numest» é um número entre 1 e 63. Quando se liga qualquer *Spectrum* ele é inicializado como estação 1, por isso é importante que cada um dos utentes haja concordado em utilizar um único número de estação com um FORMAT adequado. Na realidade, «numest» pode ainda ser 0, utilizando-se em casos muito especiais que serão descritos mais adiante. O número de estação é armazenado na nova variável do sistema NTSTAT (23749), de modo que FORMAT «n»; numest é equivalente a POKE 23749, numest. Para achar o número de estação nesse momento, basta fazer

PRINT PEEK (23749)

Para abrir um canal de forma a enviar ou receber dados da estação «num» utilizar-se-á:

OPEN #S, «n»; num

Depois deste OPEN, pode utilizar-se o comando PRINT # para enviar dados e INPUT # e INKEY$ # para receber dados. É de notar que o comando OPEN deve ser encarado como criando um enlace de comunicações entre o *Spectrum* que utiliza o comando e a estação referida nesse comando.

Há uma dificuldade ao utilizar a rede para enviar e receber dados: a outra estação deve, obrigatoriamente, saber que se estabeleceu uma ligação entre ela e outra máquina. Se, por exemplo, abrirmos (OPEN) um canal de estação com a estação 13 e esta não estiver interessada, não existir ou estiver executando outra coisa qualquer, o *Spectrum* esperará, talvez para sempre (ou até que se carregue na tecla BREAK), tentando receber ou transmitir dados para aquela estação faltosa. Noutras palavras, a transmissão de dados na rede utiliza *handshaking* completo para assegurar a recepção, com êxito, da informação transmitida. Esta necessidade de uma estação saber o que está a fazer a outra sugere que as redes de *Spectrum* devem confinar-se a uma sala única! No entanto, é possível imaginar programação adicional em código máquina que agregue troca de mensagens e outras vantagens encontradas em outras redes.

O comando INKEY$ # é a excepção ao protocolo completo de *handshaking*, e devolverá uma mensagem nula (sem caracteres) se não estiver a ser transmitida nenhuma informação da estação à qual se refere o *stream*. Desta forma pode-se pesquisar as estações da rede para saber se algumas delas estão à espera, procurando enviar

dados à primeira estação. Caso contrário, INKEY$ # funciona da maneira habitual e devolve o carácter enviado a seguir.

Além de *handshaking*, outra característica importante dos canais de rede é que todos eles têm um *buffer*. Do mesmo modo que o canal da *microdrive*, este *buffer* faz parte do descritor do canal (ver mais adiante) mas tem só 256 *bytes*. A acção do *buffer* tem o mesmo efeito nos canais de rede do que nos canais das *microdrives*. Isto é, se escrevermos 256 caracteres num canal de rede antes de ser enviado qualquer dado para a estação receptora, podemos ler 256 *bytes* antes que a estação transmissora envie outro *buffer* de dados. Do mesmo modo, só serão enviados *buffers* incompletos como resultado de um comando CLOSE #4.

Além dos comandos relativos a canais e *streams*, pode ainda utilizar-se a rede para intercambiar programas em BASIC *ZX*. O comando

SAVE «n»; num

enviará um programa à estação «num» que, por sua vez, utilizará o comando

LOAD* «n»; org

para o receber, onde «org» é o número da estação que envia o programa. A comunicação também é feita com *handshaking* completo, e tanto a estação transmissora como a receptora esperarão até que a outra esteja pronta. Também se pode utilizar MERGE* e VERIFY* da mesma maneira.

ESTAÇÃO 0 E DIFUSÃO

Os comandos de rede descritos até aqui possibilitam a troca de dados e programas entre duas estações quaisquer. No entanto, é necessário muitas vezes transferir o mesmo programa de um *Spectrum* para vários outros, o que se consegue utilizando o número de estação 0, a «estação difusora». Informação que seja transmitida à estação 0 será transmitida imediatamente, sem *handshaking*, e um número qualquer de estações pode recebê-la. Por exemplo, quando se deseja difundir um programa, todas as estações deverão introduzir em primeiro lugar

LOAD* «n»; 0

Depois espera-se que, na estação transmissora, se introduza

SAVE* «n»; 0

e envie o programa que está na memória, nesse momento. Note-se que é importante que em todas as estações receptoras se haja introduzido LOAD* antes do envio do programa pela estação transmissora.

PRINCÍPIO DE OPERAÇÃO

A rede utiliza uma ligação de dois cabos: uma linha «sinal» que leva impulsos variáveis entre 0 e 5V, e uma linha «terra», de regresso. A parte mais difícil na operação da rede é assegurar que, em cada momento, só um *Spectrum* transmite dados. Se duas máquinas transmitirem dados simultaneamente, a condição «alta» (5 V) tem precedência sobre a condição «baixa» (0 V). Por outras palavras, se uma máquina está a tentar conduzir a rede em condição alta (5 V) e outra tenta conduzi-la em condição baixa (0 V), a rede adoptará a condição alta. No entanto, esta situação, conhecida como «contenção da rede», tem que ser evitada pela utilização do protocolo da rede. Antes que uma máquina transmita dados, deve ganhar o controle da rede e assim evitar que outra máquina qualquer a utilize.

Quando uma estação vai enviar informação, pesquisa, em primeiro lugar e durante tempo suficiente, a condição da rede, detectando impulsos, correspondentes a dados, se a rede estiver a transmitir um bloco de dados nesse instante. Se não forem detectados quaisquer impulsos, a estação transmite um *byte* único, com o número da estação. À medida que transmite cada impulso, a estação verifica a rede para assegurar-se de que os impulsos são os correctos. Se encontra discrepâncias — se, por exemplo, enviou um impulso «baixo» e a rede está em condição «alta» — significa que há outra estação a tentar, simultaneamente, controlar a rede. Quando assim sucede, a estação que detecta o erro pára a transmissão e reinicia o procedimento de reclamação de rede.

Tendo-o conseguido, envia um cabeçalho com o número da estação para a qual a informação deve ser enviada e o número da estação que deseja enviar essa informação. O *byte* enviado para ganhar o controle da rede é detectado por todas as estações que procuram ler informação da rede, e todas elas examinam o cabeçalho. A estação de número igual ao do cabeçalho, vindo de uma estação da qual espera receber informação, enviará um *byte* de reconhecimento (posto em 1). Se este *byte* for recebido pela estação transmissora, esta enviará um bloco de dados para a estação receptora. Se o *byte* de reconhecimento não for recebido, a estação transmissora repetirá toda a operação, incluindo a reclamação da rede. Este procedimento só falhará se duas estações tentarem reclamar a rede simultaneamente. Neste caso, a estação cujo número for mais pequeno será a primeira a detectar o erro e parará a transmissão. A outra estação continuará então a enviar o seu *byte* de reclamação de modo a terminar a sua transmissão de dados. Utilizando este procedimento, várias máquinas podem enviar dados através da rede, cada uma esperando a sua vez de reclamar a rede e de transmitir o seu bloco de dados.

O DESCRITOR DE CANAL DA REDE

A rede introduz mais um descritor de canal no BASIC *ZX*, cujo formato é

byte	nome	observações
0	—	endereço 8, a rotina de contole de erros
2	—	endereço 8, a rotina de controle de erros
4	—	«N», o identificador do canal
5	—	endereço da rotina de saída
7	—	endereço da rotina de entrada
9	—	276, comprimento do descritor de canal
o bloco do cabeçalho		
11	NCIRIS	o número da estação de destino
12	NCSELF	o número da estação no momento de abertura (OPEN) do canal
13	NCNUM	número do bloco de dados
15	NCTYPE	tipo do bloco de dados (0=dados 1=EOF)
16	NCOBL	número de *bytes*, relativos a dados, no bloco
17	NCDCS	soma de verificação dos dados
18	NCHCS	soma de verificação do cabeçalho

informação geral

19	INCCUR	a posição do último carácter tirado do *buffer*
20	NCIBL	número de *bytes* no *buffer* de entrada

bloco de dados

21	NCB	*buffer* de dados com 255 *bytes*

O formato deste descritor de canal é de fácil compreensão, e deve ser comparado com os descritores de canal das *microdrives* e da *interface RS232*. Notar que NCSELF contém o número da estação no momento da abertura (OPEN). Isto significa que existe a possibilidade de ter vários canais de rede abertos, cada um com um identificador de estação diferente.

UTILIZAÇÃO DA REDE EM ASSEMBLY

Existem várias rotinas em código máquina na nova ROM e de 8 k que podem ser utilizadas pelo programador de linguagem assembly. O procedimento de chamada é o mesmo utilizado nas *microdrives* e ligação *RS232*, ou seja,

RST 8
código

onde «código» pode ser um dos seguintes:

código	acção resultante
45	abrir (OPEN) um canal de rede temporário. A variável de sistema D-STR1 deve conter o número da estação de destino e NTSTAT (23749) deve conter o número da estação presente nesse momento.
46	fechar (CLOSE) um canal de rede. O registo IX deve conter o endereço do descritor de canal
47	ler (READ) um registo de rede. O registo IX deve conter o endereço do descritor de canal
48	escrever um registo na rede. O registo IX deve conter o endereço do descritor de canal. A=0 escreverá dados. A=1 enviará um registo «fim de ficheiro»

Note-se que nestas observações um registo da rede é uma transacção completa, incluindo *byte* inicial de controle, cabeçalho e bloco de dados. A rotina «ler um registo da rede» deve regressar com a *flag carry* se não for recebido nenhum registo durante um tempo razoável. No entanto, parece existir um erro que corrompe o estado de *flag carry* e que torna a rotina praticamente incapaz de servir. Talvez isto sofra correcção em versões posteriores da nova ROM de 8 k.

«SPECTRUMS» DE SERVIÇO

Uma das características que se pretende ver numa rede é a comparticipação de periféricos. Evidentemente que se o mesmo programa é para ser carregado em todas as máquinas ligadas à rede, só uma delas necessitará de ter *microdrives*. Do mesmo modo, seria útil compartilhar uma impressora entre todas as máquinas da rede, coisa fácil se se designar uma máquina como «a que serve a impressora e a *microdrive*». Esta máquina executa simplesmente um programa que aceita dados da rede e os encaminha para o periférico adequado. Existem muitas formas para implementar um programa «de serviço» (pode encontrar-se um no manual da *interface* 1), mas nenhum dos métodos conhecidos pelo autor é inteiramente satisfatório. No entanto, é importante verificar que, para compartilhar periféricos entre vários *Spectrums*, um deles deverá executar o programa de serviço, o que reduz, em uma unidade, o número de máquinas disponíveis.

CAPÍTULO 12

Aplicações de programação avançada

Este capítulo final apresenta vários exemplos independentes. A maioria deles utiliza informação de capítulos anteriores, mas também se apresentará alguma matéria nova. A programação avançada tem dois aspectos. O primeiro ocupa-se da produção de bons programas, claros, de fácil utilização e sem erros. O segundo dirige-se à utilização das possibilidades oferecidas mas de forma original. No entanto, este último parte do princípio de que o leitor domina a arte de escrever programas simples, de estrutura transparente, de operação fácil e com um mínimo de erros.

O facto de fazer coisas nunca vistas com a máquina não é razão para abandonar o estilo da programação!

VECTORES DE UM «BYTE»

Algumas vezes, a necessidade de armazenar um vector com muitos números de valores pequenos torna muito ineficiente a utilização de um vector BASIC numérico. Cada elemento do vector usa cinco *bytes*, mas, se os números têm valores entre 0 e 255, cada elemento deveria ocupar teoricamente um único *byte*. Na prática é muito fácil criar vectores especiais de um *byte*, utilizando apenas PEEK e POKE. Só se precisa de uma instrução que «dimensione» o vector, reservando-lhe N *bytes*, uma função que devolva o elemento de ordem I e uma função que armazene o elemento de ordem I. O dimensionamento não é difícil, porque o comando CLEAR serve para reservar um número qualquer de *bytes* com fins especiais. No entanto, para que a rotina funcione sem modificação, quer num *Spectrum* de 16 k quer num de 48 k, deverá calcular-se automaticamente a posição mais alta de memória que está a ser utilizada. Isto pode fazer-se pesquisando, com PEEK, a variável do sistema RAMTOP em 23730. Assim, a função

```
DEF FNd(N)=PEEK 23730+256*PEEK 23731-N
```

devolve o endereço da oposição de memória que está N *bytes* abaixo da posição mais elevada em uso nesse momento, e a instrução

```
CLEAR FNd(N)
```

reservará N posições de memória para o vector de um *byte*, isto é, em que cada elemento do vector só ocupa um *byte*. As funções para armazenar e obter dados são:

```
DEF FNs(I)=PEEK 23730+256*PEEK(23731)+I
DEF FNr(I)=PEEK(FNs(I))
```

a instrução

```
POKE FNs (I),D
```

armazenará o valor D no elemento de ordem I e

```
LET D=FNr (I)
```

obterá o valor armazenado no elemento de ordem I e guardá-lo-á em D. Como exemplo de utilização destes conceitos, o programa seguinte armazena 256 números num vector de um *byte:*

```
20 CLEAR FN d(256)
30 FOR I=0 TO 255
40 POKE FN s(I),I
50 NEXT I


60 FOR I=0 TO 255
70 PRINT FN r(I)
80 NEXT I
```

Utilizando o vector de um *byte* ocupa-se 0,25 k de memória; um vector normal necessitaria 1,25 k para armazenar a mesma informação.

Pode utilizar-se a mesma técnica para armazenar números maiores do que 255, considerando mais do que uma posição de memória por elemento.

COMO PASSAR PARÂMETROS A FUNÇÕES USR

Já se demonstrou muitas vezes, em capítulos anteriores, a vantagem de implementar rotinas em código máquina através de funções USR. No entanto, a maioria dos exemplos realizou uma acção qualquer mas sem tentar devolver qualquer valor, como o faz uma função normal. Na realidade, as funções USR devolvem o número de 16 algarismos binários no par de registos BC. Por exemplo, o programa

devolverá o valor 42, se for chamado como uma função USR. O que limita a utilidade de funções USR em código máquina é a dificuldade de passar parâmetros para as rotinas. Um método já utilizado várias vezes em capítulos anteriores é utilizar posições fixas de memória como «caixas de correio». Estas passam informação a rotinas do utilizador em código máquina, introduzindo-a, com POKE, nas posições de memória antes de chamar a rotina com USR. Embora funcionando, não é suficientemente flexível nem se enquadra com a maneira por que outras funções trabalham.

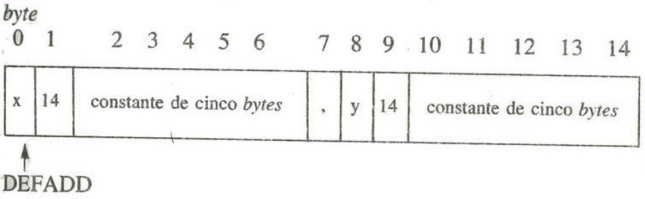
Existe uma forma de escrever rotinas em código máquina para que aceitem parâmetros normais do BASIC *ZX*. Este método baseia-se em fazer a chamada USR numa função definida pelo utilizador e com o número de parâmetros necessário. Por exemplo, se se desejar uma rotina em código máquina para adicionar dois números positivos com 16 algarismos binários poderia definir-se uma função

DEF FNa(x,y)=USR 23296

assumindo que o código máquina está armazenado no *buffer* da impressora *ZX*. O único problema que falta resolver é saber como a função USR vai ter acesso aos valores dos parâmetros «x» e «y». A solução assenta na variável de sistema DEFADD (23563), que contém o endereço do primeiro parâmetro de uma função definida pelo utilizador enquanto a função está a ser calculada. Assim, no programa

```
10 DEF FN a(x,y)=USR 23296
20 PRINT FN a(2,3)
```

DEFADD guardará o endereço do «x» na linha 10, quando a função for executada na linha 20. Isto significa que a rotina USR pode utilizar DEFADD para achar a posição de memória que guarda o «x» na linha 10. Pense-se no motivo por que a posição do nome do parâmetro de uma função tem alguma utilidade no cálculo do seu valor: quando uma função definida pelo utilizador está a ser calculada pelo BASIC *ZX*, cada um dos parâmetros utilizados é ele mesmo calculado, sendo depois armazenado em cinco *bytes* seguintes ao nome do parâmetro. Isto significa que cada um dos parâmetros é calculado na linha 20, dando o resultado 2 para x e 3 para y. (Claro que o cálculo é frequentemente muito mais complexo, envolvendo expressões numéricas completas e outras funções.) Depois, o resultado 2 é armazenado nos cinco *bytes* seguintes à letra x na linha 10, e o resultado 3 é armazenado nos cinco *bytes* seguintes à letra y na linha 10. Cada um destes cinco *bytes* é precedido por um *byte* contendo 14, o código de controle que indica que se vai seguir um número. Assim, os valores dos parâmetros não aparecem nas listagens do programa; no momento em que é chamada a rotina USR, em código máquina, a informação armazenada na linha 10 é a seguinte:



Pela utilização do valor armazenado em DEFADD, a rotina USR pode facilmente conseguir os valores dos parâmetros.

Embora seja possível escrever rotinas que processem números decimais representados em cinco *bytes*, é muito mais fácil se os valores dos parâmetros se restringem a inteiros de 16 algarismos binários. Um inteiro de 16 algarismos binários é armazenado num formato especial utilizando o segundo, terceiro e quarto *bytes*. E se forem utilizados somente inteiros positivos, o valor de 16 algarismos binários pode ser encontrado no terceiro e quarto *bytes* dos cinco.

A rotina para adicionar dois inteiros positivos de 16 algarismos binários é, agora, fácil de escrever:

endereço	linguagem assembley	código	observações
23296	LD IX,(23563)	221,42,11,92	carregar IX com o endereço do 1.º parâmetro

```
23300 LDA A, (IX+4)    221,126,4    carregar A com o 1.º byte do 1.º parâmetro
23303 ADD A, (IX+12)   221,134,12   adicionar o 1.º byte do 2.º parâmetro a A
23306 LD C,A           79           armazenar o resultado em C
23307 LD A, (IX+5)     221,126,5    carregar A com o 2.º byte do 1.º parâmetro
23310 ADC A,(IX+13)    221,142,13   adicionar o 2.º byte do 2.º parâmetro
23313 LD B,A           71           armazenar resultado
23314 RET              201          regressar ao BASIC
```

O programa seguinte, em BASIC *ZX*, carrega a rotina e exemplifica a sua utilização:

```
  10 DATA 221,42,11,92,221,126,4
,221,134,12,79,221,126,5,221,142
,13,71,201
  20 FOR A=23296 TO 23314
  30 READ D
  40 POKE A,D
  50 NEXT A

  60 DEF FN a(x,y)=USR 23296
  70 INPUT A,B
  80 PRINT FN a(A,B)
  90 GO TO 70
```

Se introduzirmos valores inteiros como resposta à linha 70, acharemos a sua soma impressa na linha 80. Talvez queiramos experimentar o uso de FNa em expressões mais complexas. Por exemplo, modifique-se a linha 80 para

```
80>PRINT FN a(A,FN a(A,A))
```

que adiciona A a A+A. O ponto a salientar é que este método de passar parâmetros a rotinas em código máquina resulta numa função que pode ser utilizada juntamente com outras funções do mesmo modo em que se usa cada uma delas. Certamente que adicionar dois inteiros de 16 algarismos binários não é uma operação muito útil para uma função em código máquina, mas, na secção seguinte, este

mesmo conceito será utilizado para agregar as funções lógicas *standard* ao BASIC *ZX*

MANIPULAÇÃO DE ALGARISMOS BINÁRIOS – AND, OR E NOT

Uma das características comuns dos programas que usam directamente os componentes físicos da máquina é a manipulação de algarismos binários. A razão reside no facto de que, muitas vezes, é o estado de um algarismo ou grupo de algarismos binários que reflecte ou controla a condição de alguma parte daqueles componentes. Outra razão para examinar ou modificar algarismos ou grupos de algarismos binários é o desejo de utilizar partes diferentes de um *byte* para armazenar informações diferentes. Por exemplo, um *byte* de atributos utiliza b7 para intermitência (ligada/desligada), b6 para brilho (ligado/desligado), de b5 a b3 e de b2 a b0 para as cores do fundo e da tinta, respectivamente.

Noutras versões do BASIC, são os operadores lógicos AND, OR e NOT que manipulam os algarismos binários, mas em BASIC *ZX* estes operadores comportam-se de outro modo. Em utilização normal do BASIC *ZX*, estes operadores operam sobre os valores 0 e 1, representando «falso» e «verdadeiro», respectivamente. Por exemplo, o resultado de x AND y é 1 se x e y forem ambos 1, e 0 se um deles for 0. Isto corresponde à interpretação habitual de palavra inglesa AND («e», em português): «x e y» é verdadeiro só se tanto x como y o forem. No entanto, o BASIC *ZX* interpreta qualquer valor diferente de zero como verdadeiro, o que dá os seguintes resultados quando x e y forem diferentes de 0 ou 1:

```
x AND y  = x se y for diferente de zero
         = 0 se y for zero
x OR y   = 1 se y for diferente de zero
         = x se y for zero
NOT x    = 0 se x for diferente de zero
         = 1 se x for zero
```

Estes resultados são úteis para escrever expressões condicionais, como se diz no capítulo 13 do manual do *Spectrum*, mas não são adequados para manipular algarismos binários.

Outras versões de BASIC implementam AND, OR e NOT com

operações *bitwise* (isto é, relativas a *bits*), que são muito mais úteis em manipulação de algarismos binários. Por exemplo, o resultado de uma operação AND *bitwise* é obtido pela aplicação do operador AND aos algarismos binários de cada um dos seus operandos: b0 do resultado é obtido a partir de b0 do 1.º operando AND b0 do 2.º operando, e assim sucessivamente. Portanto, o resultado de um AND *bitwise* a 7 e 12 é o seguinte:

```
 7          =00000111
12          =00001100
 7 AND 12   =00000100
```

ou seja, 4 em decimal. A operação AND do *Spectrum* daria o resultado 7.

A importância das operações *bitwise* reside no facto de que se pode pôr qualquer algarismo binário *(bit)* ou grupo de algarismos binários a zero fazendo um AND entre eles e uma «máscara», e também a um fazendo um OR entre eles e outra «máscara». Para ser mais preciso:

1) Para pôr a zero um grupo qualquer de algarismos binários deve construir-se uma máscara consistindo de uns em todas as posições que não sejam as que se deseje pôr a zero. Depois, deve fazer-se um AND *bitwise* entre esta máscara e o valor que contém os algarismos binários que devem ser postos a zero.
2) Para pôr a uns um grupo qualquer de algarismos binários, deve construir-se uma máscara consistindo de zeros em todas as posições exceptuando aquelas que se deseja pôr a uns. Em seguida, deve fazer-se OR *bitwise* entre esta máscara e o valor que contém os algarismos binários que devem ser postos a uns.

Por exemplo, para pôr a zero os algarismos binários de b7 a b4, teria de fazer-se um «AND *bitwise*» entre esse *byte* e a máscara seguinte:

```
b7 b6 b5 b4 b3 b2 b1 b0
0  0  0  0  1  1  1  1
```

que equivale a 15, na base decimal. Para pôr a uns os algarismos binários b7 e b6, far-se-ia um «OR *bitwise*» entre esse *byte* e a máscara seguinte:

```
b7 b6 b5 b4 b3 b2 b1 b0
1  1  0  0  0  0  0  0
```


o que equivale a 192, na base decimal.

Mas a dificuldade destes métodos origina-se em que o BASIC *ZX* não contempla os operadores *bitwise* AND, OR e NOT. Esta dificuldade ultrapassa-se facilmente com a técnica, descrita na secção anterior, de passagem de parâmetros a rotinas USR. A seguinte rotina, em linguagem assembly, executará um «AND *bitwise*» entre dois inteiros de 16 algarismos binários:

endereço	linguagem assembly	código	observações
23296	LD IX,(23563)	221,42,11,92	obter endereço de parâmetro
23300	LD A,(IX+4)	221,126,4	1.º *byte* 1.º parâmetro
23303	AND (IX+12)	221,166,12	AND com 1.º *byte* do 2.º parâmetro
23306	LD C,A	79	armazenar resultado
23307	LD A, (IX+5)	221,126,5	2.º *byte* 1.º parâmetro
23310	AND (IX+13)	221,166,13	AND com 2.º *byte* do 2.º parâmetro
23313	LD B,A	71	armazenar resultado
23314	RET	201	regressar ao BASIC

Se compararmos esta rotina AND com a rotina de adição de inteiros de 16 algarismos binários *(bytes)* dada anteriormente, verificaremos, como única diferença, que as instruções ADD foram substituídas por AND. Semelhantemente, pode conseguir-se uma rotina OR *bitwise* alterando as duas instruções AND para

```
OR (IX+12)      221,182,12
```

e

```
OR (IX+13)      221,182,13
```

Para completar o conjunto, apresentamos, a seguir, uma rotina NOT para um único parâmetro de 16 algarismos binários:

endereço	linguagem assembly	código	observações
23296	LD IX, (23563)	221,42,11,92	obter o endereço do parâmetro

23300	LD A,(IX+4)	221,126,4	carregar 1.º *byte*
23303	CPL	47	complementar (NOT) A
23304	LD C,A	79	armazenar resultado
23305	LD A, (IX+5)	221,126,5	carregar 2.º *byte*
23308	CPL	47	complementar (NOT) A
23309	LD B,A	71	armazenar resultado
23310	RET	201	regressar ao BASIC

Embora apresentássemos as três rotinas para serem carregadas no início do *buffer* da impressora, elas são independentes dessa posição e podem ser carregadas em qualquer posição da memória. O programa seguinte, em BASIC, carrega o código máquina das três rotinas no *buffer* da impressora, e define três funções:

FNa (x,y) que executa um AND *bitwise* entre x e y.
FNo (x,y) que executa um OR *bitwise* entre x e y.
FNn (x) que executa um NOT *bitwise* de x.

```
  10 DATA 221,42,11,92,221,126,4
,221,166,12,79,221,126,5,221,166
,13,71,201
  20 DATA 221,42,11,92,221,126,4
,221,182,12,79,221,126,5,221,182
,13,71,201
  30 DATA 221,42,11,92,221,126,4
,47,79,221,126,5,47,71,201
  40 FOR A=23296 TO 23348
  50 READ D
  60 POKE A,D
  70 NEXT A

 100 DEF FN a(X,Y)=USR 23296
 110 DEF FN o(X,Y)=USR 23315
 120 DEF FN n(X)=USR 23334

 130 INPUT A,B
 140 PRINT FN a(A,B),FN o(A,B),F
N n(A)
 150 GO TO 130
```


Como exemplo de utilização das funções AND, OR e NOT para simplificar problemas, considere-se a separação dos dados fornecidos pela função ATTR. No capítulo 6, este problema foi resolvido por técnicas de manuseio de algarismos binários baseadas em multiplicações e divisões por potências de dois. Multiplicar por dois equivale ao deslocamento, um lugar para a esquerda, do modelo binário que representa um dado valor, e agregando um zero à direita. Isto é equivalente ao que se passa quando se multiplica um número decimal por 10. Da mesma forma, dividir por dois e tomar a parte inteira (INT) é equivalente ao deslocamento do modelo binário uma casa para a direita, perdendo o anterior valor de b0. Utilizando estas operações de deslocamento, é possível isolar grupos de algarismos binários de um dado *byte*, e até se pode pôr 0 ou 1 num dado algarismo binário, mas este processo é habitualmente muito complexo. Mas a utilização das funções lógicas *bitwise* torna muito fácil o isolamento de partes de um *byte*. Por exemplo, isolar a cor da tinta (b2, b1, b0) de ATTR é agora simples:

```
tinta=FNa (BIN 111, ATTR (linha, coluna))
```

Isolar a cor do fundo (b5, b4, b3) também é fácil:

```
fundo=INT (FNa (BIN 111000, ATTR (linha, coluna))/8)
```

Finalmente, o brilho e a intermitência são dados por:

```
brilho=INT (FNa (BIN 1000000,ATTR(linha,coluna))/64)
```

e

```
intermitência = INT (FNa (BIN 10000000, ATTR (linha,
coluna))/128)
```

CANAIS DEFINÍVEIS PELO UTILIZADOR E A «INTERFACE» 1

No capítulo 5, já se referiu como agregar canais definíveis pelo utilizador ao *Spectrum* básico. No entanto, a adição da *interface* 1 e da nova ROM de 8 k introduz um formato alargado para outros descritores de canais. Com a *interface* 1 ligada, o descritor mais pequeno de canal corresponde aos 11 *bytes* que descrevem um canal *RS232*. Certamente que não se comete qualquer erro ao modificar o endereço da rotina de controle de I/O no descritor de canal — o

primeiro exemplo dado no capítulo 5 funciona com a *interface* 1 ligada. No entanto, se criarmos um descritor de canal completo, será melhor fazê-lo enquadrar-se nos formatos alargados, introduzidos pela ROM de 8 k.

O descritor de canal para o nosso novo canal deve ter o formato seguinte:

byte	
0	endereço da rotina de saída
2	endereço da rotina de entrada
4	nome do canal com uma letra
5	40 endereço da rotina de erros na ROM de 8 k
7	40 endereço da rotina de erros na ROM de 8 k
8	11 comprimento do descritor de canal

A única diferença entre este descritor e o do canal *RS232* é a utilização dos primeiros quatro *bytes* para armazenar os endereços das rotinas de controle de I/O, e a dos *bytes* 5 a 7 para guardar o endereço da rotina de erros na ROM de 8 k. A explicação para este facto é que quaisquer rotinas de I/O escritas por nós irão ser armazenadas na RAM e não na nova ROM de 8 k.

Além da alteração de formato, o descritor de canal terá também que ser armazenado na área da memória reservada a canais e não no *buffer* da impressora, como no capítulo 5. Para o conseguir, terá que arranjar-se um espaço de 11 *bytes* na área de canais, utilizando-se a rotina MAKESP (5717) na ROM de 16 k. Isto produzirá uma área de RAM destinada a qualquer finalidade, deslocando para cima, e pela quantidade necessária, todas as áreas de RAM já utilizadas e corrigindo todas as variáveis do sistema afectadas por esta alteração. O espaço a criar passa no par de registos BC, e o endereço da primeira posição de memória da área livre passa no par de registos HL. Assim

```
LD BC,100
LD HL,23700
CALL 5717
```

criará uma área livre com 100 *bytes*, começando em 23700. Quando se agrega um novo descritor de canal, o espaço necessário adicional é posicionado no fim de todos os outros descritores de canal. Assim, a área destinada a um canal definível pelo utilizador deve

ser criado começando em um menos o endereço armazenado na variável do sistema PROG.

A rotina seguinte criará o espaço necessário de 11 *bytes* e inserirá um novo descritor de canal:

endereço	linguagem assembly	código	observações
23296	LD HL, (23635)	42,83,92	PROG está em 23635
23299	DEC HL	43	HL=fim da área de canais
23300	PUSH HL	229	guardar HL
23301	LD BC,11	1,11,0	espaço necessário
23304	CALL 5717	205,85,22	criar espaço
			mover o descritor de canal para dentro da área de canais
23307	LD HL,23338	33,42,91	mover o descritor
23310	POP DE	209	de canal dado no
23311	PUSH DE	213	fim desta rotina
23312	LD BC,11	1,11,0	para dentro da área
23315	LDIR	237,176	livre de 11 *bytes*
			calcular deslocamento para a tabela de *streams*
23317	POP HL	225	calcular valor do
23318	LD BC,(23631)	237,75,79,92	«deslocamento» para
23322	AND A	167	a tabela de *streams*
23323	SBC HL,BC	237,66	
23325	INC HL	35	
			armazenar na tabela de *streams*
23326	LD (23582),HL	34,30,92	armazenar elemento no
23329	RET	201	*stream* 4
			rotina de controle de saída
23330	OUT LD BC,245	1,254,0	rotina de saída
23333	OUT (C),A	237,121	
23335	RET	201	

rotina de controle de entrada

23336	IN RST 8	207	rotina de entrada
23337	DEFB 18	18	erro de dispositivo ilegal

descritor de canal

23338	DEFB 34	34	endereço da rotina OUT
23339	DEFB 91	91	
23340	DEFB 40	40	endereço da rotina IN
23341	DEFB 91	91	
23342	DEFB «E»	69	identificador de canal
23343	DEFB 40	40	rotina de erros
23344	DEFB 0	0	
23345	DEFB 40	40	rotina de erros
23346	DEFB 0	0	
23347	DEFB 11	11	comprimento do descritor de canal
23348	DEFB 0	0	

As rotinas de saída e entrada utilizadas pelo descritor de canal são as mesmas que foram utilizadas no capítulo 5, que enviam dados para o porto associado à moldura. Embora esta rotina abra o *stream* 4 por defeito, pode alterar-se isto armazenando o endereço de um elemento diferente da tabela de *streams* nas posições de memória 23327 e 23328.

O programa em BASIC *ZX* apresentado a seguir carrega a rotina em código máquina e dá um exemplo da sua utilização:

```
  10 DATA 42,83,92,43,229,1,11,0
,205,85,22
  20 DATA 33,42,91,209,213,1,11,
0,237,176
  30 DATA 225,237,75,79,92,167,2
37,66,35
  40 DATA 34,30,92,201
  50 DATA 1,254,0,237,121,201
  60 DATA 207,18
  70 DATA 34,91,40,91,69,40,0,40
,0,11,0
  80 FOR A=23296 TO 23348
  90 READ D
 100 POKE A,D
 110 NEXT A
 120 LET S=4: GO SUB 1000
 130 PRINT #4;0;
 140 PRINT #4;7;
 150 GO TO 130

1000 LET A=23674+2*3
1010 POKE 23327,A-INT (A/256)*25
6
1020 POKE 23328,INT (A/256)
1030 LET A=USR 23296
1040 RETURN
```

As linhas de 10 a 110 carregam, da maneira habitual, o programa em código máquina: a sub-rotina 1000 abre o *stream* S ao novo descritor de canal, e as linhas 130 a 150 escrevem, com PRINT #, os valores 0 e 7 no canal da moldura, fazendo-o intermitentemente a preto e branco. É de notar que este método de agregação de um canal definível pelo utilizador funciona com a *interface 1* ligada ou desligada.

JUNÇÃO DE COMANDOS AO BASIC «ZX»

É fácil criar novos comandos de BASIC *ZX* desde que tenhamos uma *interface* 1 ligada ao *Spectrum* e sejamos bons programadores em linguagem assembly do *Z80*. A chave para acrescentar os nossos próprios comandos reside na forma como se manuseiam erros uma vez ligada a *interface* 1. Quando ocorre um erro, usa-se um comando RST 8 para chamar a rotina de erros. No entanto, e como já se descreveu, essa chamada é interceptada pela *interface* 1 e a nova ROM de 8 k é paginada. Esta examina a natureza do erro e verifica se o comando que o originou pode ser tratado correctamente por ela — isto é, se é um dos novos comandos implementados pela nova ROM de 8 k. Se for, será chamada a rotina em código máquina apropriada, desenvolvendo-se depois o controle à ROM de 16 k. Se o comando não for reconhecido pela ROM de 8 k, o controle será devolvido à ROM de 16 k no endereço dado pela nova variável de sistema, VECTOR (23735). Habitualmente esta variável contém o endereço de uma rotina final de tratamento de erros, mas pode alterar-se este endereço de forma a transferir o controle para uma rotina fornecida pelo utilizador que faz uma última tentativa para reconhecer e implementar o comando, seja ele qual

for, que tenha sido rejeitado tanto pela ROM de 16 k como pela ROM de 8 k.

A alteração do endereço em VECTOR intercepta, efectivamente, o tratamento normal de instruções incorrectas do BASIC *ZX* ou do BASIC *ZX* alargado. Isto implica que qualquer comando adicionado desta forma ao BASIC deve, obrigatoriamente, causar um erro. Por exemplo, poderíamos acrescentar novos comandos tais como:

```
*T
ASN
PAUSE*
```

cada um dos quais causaria um erro porque não seria reconhecido por nenhuma das ROMs. Isto garante que o seu processamento será transferido para a rotina para a qual VECTOR «aponta».

Na prática, adicionar comandos é bastante complexo, sendo essencial um conhecimento profundo da organização da ROM de 16 k. Se se desejar alargar o BASIC *ZX*, não há forma de evitar a utilização de muitas das suas rotinas. No entanto, quando se transfere o controle para a nova rotina utilizando VECTOR, ainda está paginada a nova ROM de 8 k. Para chamar rotinas na ROM de 16 k termos de utilizar

```
RST 16
DEFW endereço
```

onde «endereço» é o endereço da rotina contida na ROM de 16 k que desejamos utilizar. Todos os registos são devolvidos com os valores aí deixados pela ROM de 16 k. Enquanto a ROM de 8 k estiver paginada, todos os endereços RST serão diferentes do que seria de esperar se estivesse paginada a ROM de 16 k. Os mais importantes são:

```
RST 32  relata um erro da ROM de 8 k;
        o código de erro segue-se a RST 32
RST 40  relata um erro da ROM de 16 k;
        o código de erro é armazenado em ERRNO
RST 48  cria novas variáveis de sistema
```

As rotinas que implementam novos comandos têm sempre a mesma estrutura geral:

1) Um verificador do sintaxe.
   Verifica se o novo comando tem a forma correcta. Se não tem, deverá ser relatado um erro, saltando à posição 496. A verificação do sintaxe deve ser feita até ao fim da linha e deixar CH-ADD apontando para o fim da linha. O fim da instrução deve ser testado pela chamada da sub-rotina 1463 na ROM de 8 k. Se só estiver a ser verificada a sintaxe, o controle não regressará desta sub-rotina, mas se o programa estiver a ser executado (RUN) o controle passará à segunda metade da rotina.
2) Um módulo de execução (RUN *time module*).
   Esta parte da rotina é a que realmente faz o trabalho necessário à implementação do novo comando. Quando o módulo de execução está terminado, deve devolver o controle ao BASIC *ZX* saltando para 1473 na ROM de 8 k.

Duas rotinas indispensáveis, contidas na ROM de 16 k, para escrever novos comandos são

endereço	função
24	passar o carácter presente nesse momento na linha BASIC para o registo A
32	passar o carácter seguinte ao presente nesse momento na linha BASIC para o registo A. Chamadas sucessivas a esta rotina fazem avançar o carácter, inspeccionando assim toda a linha

A rotina «carácter seguinte» salta automaticamente sobre espaços em branco e códigos de controle, devolvendo assim o carácter seguinte «útil».

Como exemplo simples de adição de um comando, a rotina seguinte implementa o comando

```
PAUSE*
```

que parará o processamento até que se prima numa tecla qualquer

endereço	linguagem assembly	código	observações
		verificação de sintaxe	
23296	RST 16	215	obter código de comando
23297	24	24,0	

```
23299 CP 242          254,242     PAUSE?
23301 JP NZ,ERR       194,240,1   erro
23304 RST 16          215         obter carácter seguinte
23305 32              32,0
23307 CP 42           254,42      é '*'
23309 JP NZ,ERR       194,240,1   erro
23312 RST 16          215         mover para o fim da instrução
23313 32              32,0
23315 CALL CKEND      205,183,5   verificar se fim da instrução

                  módulo de execução

23318 LOOPXOR A       175         pôr A a zero
23319 IN A, (254)     219,254     inspeccionar teclado
23321 AND 31          230,31      guardar só os 5 bits mais baixos
23323 SUB 31          214,31      A=31 se nenhuma tecla foi
                                  premida
23325 JP Z,LOOP       202,22,91   ciclo até tecla ser premida
23328 JP COMEND       195,193,5   regressar à ROM de 16 k
```

Na verificação do sintaxe testa-se a existência do comando PAUSE, e depois a existência do carácter «*». Desde que sejam encontrados, o controle é transferido para CKEND, que só devolve o controle à rotina se o programa em BASIC estiver a ser executado (RUN). A segunda parte da rotina (módulo de execução) forma unicamente um ciclo até uma tecla ser carregada, e depois regressa, via COMEND, que pagina a ROM de 16 k e permite continuar a execução do programa BASIC. O seguinte programa, em BASIC, carrega o código máquina e introduz, com POKE, o novo valor em VECTOR.

```
  10 DATA 215,24,0,254,242,194,2
40,1,215,32,0,254,42,194,240,1,2
15,32,0,205,183,5
  20 DATA 175,219,254,230,31,214
,31,202,22,91,195,193,5
  30 FOR A=23296 TO 23330
  40 READ D
  50 POKE A,D
  60 NEXT A
  70 POKE 23735,0
  80 POKE 23736,91
```


Depois de executar este programa, o comando PAUSE* será aceite em qualquer programa, tendo como efeito a paragem da execução até que se carregue numa tecla qualquer.

Rotinas para agregar outros comandos novos ao BASIC tomam esta mesma estrutura — um verificador da sintaxe e um módulo de execução (RUN-*time*) mas, normalmente, o módulo de execução será muito mais complexo de que o apresentado no exemplo.

UM PROGRAMA PARA ESTATÍSTICAS

Nos últimos exemplos utilizou-se bastantes vezes a linguagem assembly do *Z80*. Para ilustrar como pode ser útil o conhecimento do funcionamento interno do *Spectrum*, mesmo em programas BASIC *ZX* aparentemente imediatos, apresenta-se no exemplo seguinte um programa de estatística que edita os dados, calcula valores estatísticos, traça histogramas, grava e lê dados na fita magnética.

O primeiro problema destina-se a armazernar os dados que se querem analisar.

O método mais evidente é utilizar um vector numérico. Este vector pode facilmente ser gravado, com SAVE, e lido, com LOAD, permitindo analisar e editar rapidamente tantos dados como seja possível guardar na RAM disponível. No entanto, a utilização de um vector também tem desvantagens. A primeira é que quando se carrega um vector utilizando

LOAD «nomedofich» DATA D()

o número de elementos do vector não está imediatamente acessível.

Quando os dados são gerados pelo programa, não é difícil ir contando o número de elementos numa variável N, por exemplo. O problema é saber o valor de N quando se está a ler o vector gravado na fita magnética. Uma solução seria armazenar N num dos elementos do vector antes de ele ser gravado, mas isto é uma complicação desnecessária no esquema de armazenamento de dados. Utilizando o formato de armazenamento de vectores dado no capítulo 4 (ver figura 4.1), é possível escrever poucas linhas de BASIC *ZX* para pesquisarem, com PEEK, a dimensão do vector. Pergunta-se é como encontrar a posição da memória onde se inicia o vector. Uma maneira seria escrever uma rotina em código máquina que pesquisasse a área das variáveis à procura do vector, mas há

uma maneira muito mais simples. A variável do sistema DEST (23629) guarda o endereço da variável de destino durante uma assignação. Assim, se desejamos encontrar o endereço do primeiro elementos do vector D, basta fazer

LET T=D(1)
LET D(1)=PEEK 23629+256*PEEK 23630

Depois destes comandos, os dois seguintes

LET N=PEEK(D(1)−1)+256*PEEK(D(1))
LET D(1)=T

armazenarão a dimensão do vector em N, e em D (1) voltarão a pôr o seu valor inicial

O único problema restante é como adicionar dados a um vector já existente. Se o vector guardar N números, e desejarmos agregar-lhe mais M números, o vector terá que ser alargado para DIM D (N+M) sem perda de nenhum dos dados originais. Isto também poderia ser obtido utilizando uma rotina em código máquina, mas, uma vez mais, o BASIC *ZX* é suficiente. Para alargar o vector D para N+M primeiro dimensiona-se um vector DIM E(N) e copiam-se todos os dados existentes de D para E. Depois redimensiona-se D para DIM (N+M) e copiam-se então todos os dados de E para D, ficando M elementos livres, prontos para receber novos valores. Finalmente redimensiona-se o vector E para DIM E (1) para libertar o espaço que ele ocupava. Talvez não seja o método mais rápido, mas é muito simples!

Agora que se resolveram estes dois problemas, apresenta-se o programa de estatística:

```
  10 REM programa de estatistica

 500 CLS
 510 PRINT TAB 5;"Estatistica"
 520 PRINT AT 6,0
 530 PRINT "(1) Introduzir novos
 dados"
 540 PRINT "(2) Gerar dados alea
toriamente"
 550 PRINT "(3) Alterar dados"
 560 PRINT "(4) Gravar/Carregar
dados"
 570 PRINT "(5) Executar calculo
s"
 580 PRINT "(6) Tracar histogram
a"
 590 PRINT "(7) Sair"
 600 PRINT AT 21,0;"Introduza o
numero pretendido"
 610 INPUT sel
 620 IF sel=1 THEN GO SUB 3000
 630 IF sel=2 THEN GO SUB 1000
 640 IF sel=3 THEN GO SUB 4000
 650 IF sel=4 THEN GO SUB 1500
 660 IF sel=5 THEN GO SUB 5500
 670 IF sel=6 THEN GO SUB 6000
 680 IF sel=7 THEN STOP
 690 GO TO 500

1000 CLS
1010 PRINT "Dados aleatorios"
1020 PRINT "Quantos valores ";
1030 INPUT n
1040 PRINT n
1050 DIM d(n)
1060 PRINT AT 3,0;"Dados fraccio
narios ou inteiros,f/i ";
1070 INPUT a$
1080 IF a$<>"f" AND a$<>"i" THEN
 GO TO 1060
1090 PRINT a$
1100 LET t=0
1110 IF a$="i" THEN LET t=1
1120 PRINT AT 5,0;"valor mais ba
ixo ";
1130 INPUT l
1140 PRINT l
1150 PRINT "valor mais alto ";
1160 INPUT h
1170 PRINT h
1180 IF h>=l THEN GO TO 1210
1190 PRINT "mais alto<mais baixo
!"
1200 GO TO 1120
1210 FOR i=1 TO n
1220 LET d(i)=RND*(h-l+t)+l
1230 IF t=1 THEN LET d(i)=INT d(
i)
1240 PRINT "dado ";i;" = ";d(i)
1250 NEXT i
1270 GO TO 8900

1500 CLS
1510 PRINT "Gravar ou carregar d
ados g/c"
1520 INPUT a$
```

```
1530 IF a$<>"c" AND a$<>"g" THEN
 GO TO 1500
1540 IF a$="c" THEN GO TO 1600
1550 PRINT "Nome do ficheiro"
1560 INPUT f$
1570 SAVE f$ DATA d()
1580 GO TO 8900
1600 PRINT "Quer mesmo carregar
dados?"
1610 INPUT a$
1620 IF a$="n" THEN GO TO 8900
1630 IF a$<>"s" THEN GO TO 1600
1640 PRINT "Nome do ficheiro "
1650 INPUT f$
1660 LOAD f$ DATA d()
1670 LET t=d(1)
1680 LET d(1)=PEEK 23629+256*PEE
K 23630
1690 LET n=PEEK (d(1)-1)+256*PEE
K d(1)
1700 LET d(1)=t
1710 RETURN

2000 LET m=0
2010 LET s=0
2020 LET l=d(1)
2030 LET h=1
2040 FOR i=1 TO n
2050 LET m=m+d(i)
2060 IF l>d(i) THEN LET l=d(i)
2070 IF h<d(i) THEN LET h=d(i)
2080 NEXT i
2090 LET m=m/n
2100 FOR i=1 TO n
2110 LET s=s+(d(i)-m)*(d(i)-m)
2120 NEXT i
2130 LET s=s/(n-1)
2140 RETURN

2500 CLS
2510 PRINT "numero de valores= "
;n
2520 PRINT "maximo= ";h
2530 PRINT "minimo= ";l
2540 PRINT "amplitude= ";h-l
2550 PRINT "media= ";m
2560 PRINT "variancia= ";s
2570 PRINT "desvio padrao= ";SQR
(s)
2580 GO TO 8900

3000 CLS
```



```
3010 PRINT "Input de dados"
3020 PRINT "Quantos valores ?";
3030 INPUT n
3040 PRINT n
3050 DIM d(n)
3060 FOR i=1 TO n
3070 PRINT AT 21,0;"valor ";i;"
= ";
3080 INPUT d(i)
3090 PRINT d(i): PRINT
3100 NEXT i
3110 PRINT "input de dados compl
eto"
3120 GO TO 8900

4000 CLS
4010 PRINT TAB 5;"Alterar dados"
4020 PRINT AT 5,0
4030 PRINT "(1) lista de dados"
4040 PRINT "(2) alterar dados"
4050 PRINT "(3) rejeitar dados"
4060 PRINT "(4) acrescentar dado
s"
4070 PRINT "(5) voltar ao menu p
rincipal"
4080 PRINT AT 21,0;"Introduza o
numero pretendido"
4090 INPUT ed
4100 IF ed=1 THEN GO SUB 4200
4110 IF ed=2 THEN GO SUB 4500
4120 IF ed=3 THEN GO SUB 4600
4130 IF ed=4 THEN GO SUB 4800
4140 IF ed=5 THEN RETURN
4150 GO TO 4000

4200 CLS
4210 PRINT "comeco da lista em ?
";
4220 INPUT l
4230 PRINT l
4240 PRINT "fim da lista em (-1
lista ate ao fim) ? ";
4250 INPUT h
4260 PRINT h: PRINT
4270 IF h<0 THEN LET h=n
4280 IF l>h THEN GO TO 4200
4290 IF l>n OR h>n OR l<1 OR h<1
 THEN GO TO 4200
4300 FOR i=l TO h
4310 PRINT "valor ";i;" = ";d(i)
4320 NEXT i
4330 GO TO 8900
```

```
4500 CLS
4510 PRINT "alterar que valor? "
;
4520 INPUT i
4530 IF i<1 OR i>n THEN GO TO 45
00
4540 PRINT i
4550 PRINT "valor actual = ";d(i
)
4560 PRINT "novo valor = ";
4570 INPUT d(i)
4580 PRINT d(i)
4590 GO TO 8900

4600 CLS
4610 PRINT "rejeitar a partir de
 ";
4620 INPUT l
4630 PRINT l
4640 PRINT "acabando em ";
4650 INPUT h
4660 PRINT h
4670 IF h<l THEN GO TO 4600
4680 IF h>n OR h<1 OR l>n OR l<1
 THEN GO TO 4600
4690 PRINT
4700 PRINT "rejeitar de ";l;" at
e ";h
4710 PRINT "correto?";
4720 INPUT a$
4730 PRINT a$
4740 IF a$(1)<>"s" THEN RETURN
4750 FOR i=h+1 TO n
4760 LET d(l+i-h-1)=d(i)
4770 NEXT i
4780 LET n=n-h+l-1
4790 RETURN

4800 CLS
4810 PRINT "quantos valores mais
 ?";
4820 INPUT m
4830 PRINT m
4840 DIM e(n)
4850 PRINT "...a arranjar espaco
..."
4860 FOR i=1 TO n
4870 LET e(i)=d(i)
4880 NEXT i
4890 PRINT "...quase pronto..."
4900 DIM d(n+m)
4910 FOR i=1 TO n
```



```
4920 LET d(i)=e(i)
4930 NEXT i
4940 PRINT "pronto"
4950 DIM e(1)
4960 FOR i=n+1 TO n+m
4970 PRINT "valor ";i;" = ";
4980 INPUT d(i)
4990 PRINT d(i)
5000 NEXT i
5010 LET n=n+m
5020 GO TO 8900

5500 CLS
5510 PRINT "...a calcular..."
5520 GO SUB 2000
5530 GO TO 2500

6000 CLS
6010 PRINT "quantas classes ? ";
6020 INPUT m
6030 PRINT m
6040 PRINT "valor maximo= ";
6050 INPUT h
6060 PRINT h
6070 PRINT "valor minimo= ";
6080 INPUT l
6090 PRINT l
6100 IF h<l THEN GO TO 6000
6110 LET d=(h-l)/m
6120 GO SUB 7000
6130 FOR i=1 TO m
6140 PRINT AT 21,0;INT (l*100)/1
00;TAB 6;
6150 IF h(i)=0 THEN GO TO 6190
6160 FOR j=1 TO h(i)/f*25
6170 PRINT CHR$ 143;
6180 NEXT j
6190 PRINT : PRINT
6200 LET l=l+d
6210 NEXT i
6220 PRINT : PRINT
6230 GO TO 8900

7000 DIM h(m)
7010 FOR i=1 TO n
7020 LET j=(d(i)-l)/(h-l)*m+1
7030 LET j=INT j
7040 IF j<1 OR j>m THEN GO TO 70
60
7050 LET h(j)=h(j)+1
7060 NEXT i
7070 LET f=0
```

```
7080 FOR i=1 TO m
7090 IF f<h(i) THEN LET f=h(i)
7100 NEXT i
7110 RETURN

8900 PRINT
8910 PRINT AT 21,0;"prima uma te
cla para continuar"
8920 IF INKEY$="" THEN GO TO 892
0
8930 RETURN
```

Este é o maior programa apresentado neste livro; contém muitas técnicas e ideias que só se tornam importantes em programas extensos! Em particular, deve notar-se a utilização de *menus* que permitem ao utilizador seleccionar a opção desejada, e a verificação das entradas de dados (INPUT) para evitar que dados incorrectos sejam introduzidos no programa, causando erros que interromperiam a sua execução. A utilização repetida de sub-rotinas torna o programa de compreeensão, generalização e manutenção fáceis. Para melhor análise do programa, apresenta-se seguidamente uma tabela das sub-rotinas utilizadas:

número de linha	finalidade
500- 690	*menu* principal
1000-1270	gerar dados aleatórios
1500-1710	gravar e ler os dados — SAVE/LOAD
2000-2140	calcular valores estatísticos
2500-2580	imprimir resultados
3000-3120	entrada de dados
4000-4150	*menu* de edição
4200-4330	listar dados
4500-4590	alterar dados
4600-4790	eliminar dados
4800-5020	adicionar dados
5500-6230	traçar histograma
7000-7110	calcular frequências
8900-8930	carregar em qualquer tecla para continuar

As modificações a fazer a este programa para que funcione com dados armazenados em *microdrives* são simples: substituir as instruções SAVE... DATA e LOAD... DATA para SAVE*... DATA e LOAD*... DATA, e alterar a forma dos nomes dos ficheiros utilizados. No entanto, organizam-se melhor os dados nas *microdrives* utilizando ficheiros PRINT. Em vez de armazenar todos os dados num vector, pode utilizar-se um ficheiro PRINT para que se leia e processe só um único dado de cada vez. Aumenta-se assim a quantidade de dados que podem ser processados, pois agora o limite dessa quantidade é dado pela capacidade da *microdrive* e não pelo espaço de memória restante, onde se podem armazenar variáveis. A desvantagem desta estratégia é o tempo. De cada vez que se lê um dado é necessário ler o ficheiro completo, além de que a adição de dados novos é ainda mais morosa.

UTILIZAÇÃO DA «INTERFACE» 2

A *interface* 2 é uma componente muito simples, que permite a utilização de *joysticks* e *cartridges* ROM, programadas, com o *Spectrum*. Do ponto de vista do utilizador, o mais importante da *interface* 2 é a introdução de uma norma para a utilização do teclado em jogos dinâmicos. Os *joysticks* são ligados como se fossem um conjunto «gémeo» das teclas da fila de cima. Assim:

sentido	*joystick* 1	*joystick* 2
	tecla	tecla
para a esquerda	6	1
para a direita	7	2
para baixo	8	3
para cima	9	4
disparar	10	5

Embora se possam ler estas teclas utilizando a função INKEY$, há vantagem em usar IN 61438 para ler o *joystick* 1 e IN 63486 para ler o *joystick* 2. As funções lógicas definíveis pelo utilizador podem utilizar-se para verificar em que teclas se carregou. Por exemplo

```
500 LET A=IN 61438
510 IF FN a(a,BIN 1)=0 THEN PRI
NT "fogo";
```

```
 520 IF FN a(a,BIN 10)=0 THEN PR
INT "cima";
 530 IF FN a(a,BIN 100)=0 THEN P
RINT "baixo";
 540 IF FN a(a,BIN 1000)=0 THEN
PRINT "direita";
 550 IF FN a(a,BIN 10000)=0 THEN
 PRINT "esquerda";
 560 PRINT
 570 GO TO 500
```

imprimirá as palavras apropriadas quando se carrega numa das teclas. É de notar que se deve juntar este programa à definição das funções lógicas dada anteriormente neste capítulo.

CONCLUSÃO

São inumeráveis os modos pelo quais o conhecimento do funcionamento interno do *Spectrum* pode tornar-se útil. O conselho mais importante do autor quanto a projectos de programação é que sejam levados a sério! Sucede muitas vezes iniciar-se um projecto sem objectivos bem definidos e pô-lo de lado quando as coisas se tornam difíceis. Começando-se com uma ideia clara daquilo que o programa deve fazer, e definindo especificações suficientemente pormenorizadas, transpor as dificuldades será um desafio compensador. Não desistam — tentem isolar os vossos problemas e escrever rotinas para investigar o que estiver a suceder. Um programa que depois de finalizado execute o que foi planeado será um prémio suficiente para o esforço de cada um.