10

Esta obra destina-se a quem pretenda saber como trabalha o «Spectrum». Mesmo não possuindo experiência de electrónica, compreender-se-á o que se passa dentro da caixa do computador.

A primeira secção da obra trata dos conceitos fundamentais aplicáveis ao desenho de computadores. A segunda estuda a estrutura do «Spectrum» da Sinclair e, em particular, a apresentação em écran, o teclado e as capacidades de gerar som.

Programas muito práticos e úteis ilustram os assuntos tratados e consegue-se uma introdução facílima no mundo do código máquina, graças a um programa monitor e a experiências de imediata conclusão.

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

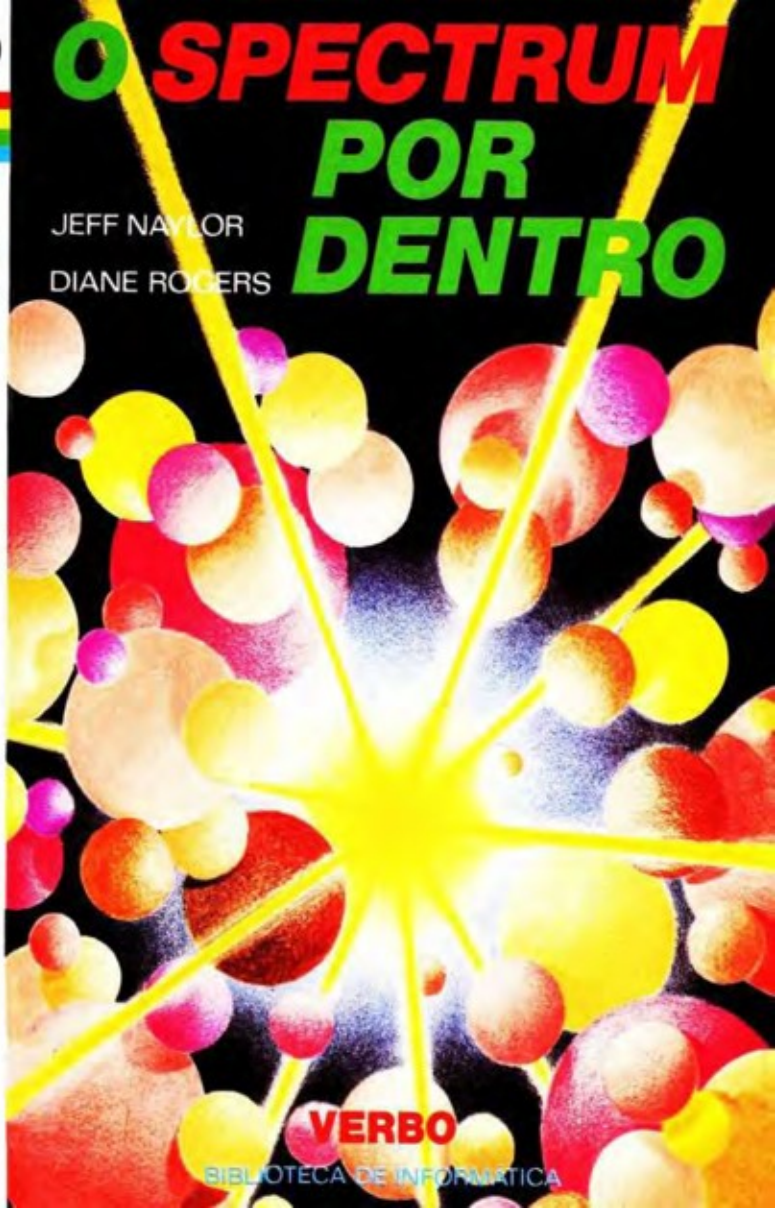
BIBLIOTECA VERBO DE INFORMÁTICA

Verbo O SPECTRUM POR DENTRO NAYLOR / ROGERS

O SPECTRUM POR DENTRO

JEFF NAYLOR

DIANE ROGERS



VERBO

BIBLIOTECA DE INFORMÁTICA

JEFF NAYLOR E DIANE ROGERS

O Spectrum por Dentro

Verbo

ÍNDICE

		Página
Sumário		7
Prefácio		9
Parte 1:	**Princípios básicos**	11
1	Electrónica digital	13
2	Números e dados	27
3	O microprocessador	34
4	Dentro da caixa	41
5	RAM e ROM	53
6	Linguagem para máquinas	58
Parte 2:	**O «Spectrum» da Sinclair**	85
7	Introdução ao *Spectrum*	87
8	O mapa de memória	95
9	O teclado	103
10	Imagens no *écran*	114
11	Som	128
12	Mais caracteres por linha	135
13	O *Spectrum* fala	153

Título do original inglês:
Inside your Spectrum

Tradução de Raul Monteiro de Sousa Machado

Editorial Verbo — Lisboa/São Paulo
Composto pela Fotocompográfica
Impresso pela Litográfica do Sul
em Abril de 1986
Ed. n.º 1681
Depósito legal n.º 10 508/86

SUMÁRIO

PARTE 1

Capítulo 1
Electrónica digital
Os fundamentos da electrónica, dando especial importância às técnicas digitais. Uma introdução à natureza da electricidade e aos circuitos simples analógicos e digitais.

Capítulo 2
Número e dados
Por que razão os computadores pensam que os homens nascem com um número errado de dedos; uma passagem pelas vias de dados; um programa de demonstração do sistema binário.

Capítulo 3
O microprocessador
Um exame externo do aparelho que comanda o computador — a UCP *Z80;* alguns dos sinais necessários para a fazer funcionar, incluindo o relógio.

Capítulo 4
Dentro da caixa
Revelação de todo o conteúdo da UCP, e uma descrição pormenorizada de como é executado um programa simples.

Capítulo 5
RAM e ROM
Os tipos de memória disponíveis, por que motivo são necessários e uma clara distinção entre os dois tipos principais.

Capítulo 6
Linguagem para máquinas
Interpretação das instruções que são tecladas como comando em BASIC; como os mesmos príncipios se aplicam ao processamento de um programa em código máquina. Uma introdução às operações em

código máquina e um glossário de mnemónicas; um programa monitor e, por fim, um desafio a escrevermos o nosso próprio programa.

PARTE 2

Capítulo 7
Introdução ao «Spectrum»
O desenvolvimento do *Spectrum:* exame minucioso das capacidades que oferece e dos circuitos electrónicos com que as realiza.

Capítulo 8
O mapa da memória
A descrição do mapa da memória.

Capítulo 9
O teclado
A estrutura do teclado, sua verificação e descodificação. Como podemos usar o teclado a partir do código máquina; e uma demonstração de um método de verificação do teclado.

Capítulo 10
Imagens no «écran»
A natureza das imagens de televisão e o modo como são apresentadas; o método pelo qual o *Spectrum* gera os sinais, e onde armazena a informação: como podemos manipular o *écran* através do código máquina.

Capítulo 11
Som
O modo como o porto BEEP forma os sons e as maneiras de o usar; o seu papel nas rotinas de gravação em fita magnética.

Capítulo 12
Mais caracteres por linha
Um programa que permite escrever mais caracteres em cada linha.

Capítulo 13
O «Spectrum» fala
Um programa que põe o *Spectrum* a dizer algumas palavras.

Prefácio

Esta obra não exige dos leitores conhecimentos prévios de electrónica ou de informática, embora seja útil possuir alguma experiência de programação em BASIC no *Spectrum;* basta ter um verdadeiro desejo de aprender. Mesmo os leitores com conhecimentos mais avançados verificarão que os primeiros capítulos levam a uma útil clarificação dos conceitos básicos e a uma preparação para os temas mais complexos que aparecem seguidamente. Obter-se-ão os melhores resultados lendo esta obra com o computador ao lado, de modo a poder fazer imediatamente a experimentação. Não se deve ter receio de experimentar as ideias que surjam: os factos que melhor se aprendem são sempre os descobertos por iniciativa do próprio estudioso.

Aviso: Muita atenção ao introduzir programas deste livro. A qualidade de impressão das impressoras de computador torna alguns caracteres muito semelhantes a outros — as vírgulas, em particular, parecem-se por vezes com pontos finais. É importante verificar se se introduziram os caracteres correctamente.

PARTE 1

Princípios básicos

CAPÍTULO 1

Electrónica digital

Ao descobrir que é a aplicação da «electrónica digital» que oferece a capacidade de trabalho dos computadores modernos, aparentemente inescrutável, não devemos temer que tal facto se torne de compreensão mais difícil do que perceber como funciona um aparelho convencional, uma simples telefonia, por exemplo. Quem já alguma vez estudou electrónica e teve dificuldade em apreender conceitos como «ondas sinusoidais», «modulação» e «capacitância», não deve preocupar-se: a teoria subjacente às técnicas digitais é muito menos complexa no que se refere a conceitos abstractos e requer menos conhecimentos de matemática. A miniaturização (muita coisa num pequeno espaço) e a velocidade de operação é que tornam os computadores modernos tão poderosos.

Qual a diferença entre a electrónica digital e a mais convencional? Em termos genéricos, um circuito digital regista somente a presença ou ausência de corrente eléctrica; por outras palavras, as partes do circuito estão simplesmente «ligadas» ou «desligadas». A quantidade exacta de electricidade presente, dentro de certos limites, é um factor de menor importância.

Os circuitos convencionais tendem a ser de muito maior precisão. Por exemplo, num sistema de alta fidelidade as quantidades de electricidade que fluem para os altifalantes reflectem exactamente as flutuações de nível sonoro que o circuito tenta imitar. Se o sistema for mal desenhado, o controle dos níveis de electricidade é pouco preciso, resultando daí apreciável distorção.

Reconhece-se muitas vezes Charles Babbage como o pai da informática moderna, pois concebeu, a partir de 1830, a primeira «máquina analítica». Era mesmo uma máquina, com rolamentos, alavancas e engrenagens interactuando de maneira bem visível. A compreensão do funcionamento de um aparelho mecânico é fácil

para o cérebro humano, pois nada há de abstracto para aprender: actuando sobre tal alavanca, esta faz funcionar aquela engrenagem, e isso acontece porque elas estão ligadas por um fio que todos podem ver.

Para se estudar um sistema electrónico é vulgar, porém, formar uma imagem da electricidade comportando-se como água ou mesmo como uns homenzinhos correndo para aqui e para ali. Não é vergonha nenhuma utilizar estas analogias, mas elas revelam-se pouco úteis na descrição de modelos já de certa complexidade. Um circuito composto por uma bateria, um interruptor e uma lâmpada pode ser explicado nesses termos, mas, experimentando aplicar essas analogias a um aparelho de televisão, ver-se-á que não só não se consegue compreender o seu funcionamento como se começará mesmo a pôr em causa o comportamento da água ou até dos homenzinhos!

A NATUREZA DA ELECTRICIDADE

Para se ter uma ideia da natureza da electricidade é necessário, em primeiro lugar, apreciar algumas realidades simples desta matéria. A comprovação de todas elas demoraria algum tempo e na análise final ainda seria preciso acreditar na palavra do autor tal como este acredita nas explicações dos cientistas. Por isso, aceitemos como verdadeiro que a matéria, quer se trate de um gás, de plástico ou de sumo de laranja, é composta por partículas minúsculas chamadas «átomos», cuja definição mais simples, embora não a mais correcta, é a de «a partícula mais pequena que existe». Em passado recente, a definição continuaria afirmando que os átomos são indivisíveis, mas sabemos hoje que tal não é o caso; e a fissão nuclear não é o propósito deste livro. Há mais de cem tipos de átomos diferentes, e alguns são mais vulgares que outros.

Os materiais formados por um só tipo de átomo são designados por «elementos», que incluem matérias tão familiares como o carbono, o ferro e o oxigénio. Os átomos de um elemento diferem dos dos outros pela sua estrutura e pelo tamanho, variando desde o simples átomo do hidrogénio até aos muito

pesados, como o do urânio. Outras substâncias são formadas por associações de diferentes átomos, por vezes como simples misturas de elementos, mas as mais das vezes resultando de interligações de átomos que formam «moléculas». Dois átomos de hidrogénio, por exemplo, unem-se a um átomo de oxigénio para dar origem a uma molécula de água.

Através de um longo processo de raciocínio teórico, seguido por experiências comprovativas, os cientistas conseguiram formar uma imagem da estrutura do átomo. E é no modo como se estruturam os diferentes tipos de átomo que reside a chave da ideia de electricidade. Cada átomo consiste num núcleo formado por determinado número de «protões» e de «neutrões», a que se chama o «núcleo indivisível» e que é a porta de acesso à ciência da física nuclear.

À volta deste núcleo orbitam os «electrões». Cada tipo de átomo necessita de um número diferente destas partículas para ter uma estabilidade ideal: o hidrogénio só precisa de um único electrão, enquanto outros necessitam de dúzias. Contudo, átomos há que podem suportar um certo desequilíbrio na sua estrutura, possuindo quer electrões em excesso quer a menos do que o normal. Como provavelmente já se adivinhou, são estes electrões que nos conduzem aos fenómenos eléctricos, desde o relâmpago até aos relógios digitais.

Se considerarmos o comportamento dos electrões numa pilha, aparelho fácil de imaginar, começaremos a aperceber-nos da utilidade da electricidade. As pilhas são construídas de maneira a representarem uma fonte de electricidade: possuem duas áreas, uma com um *superavit* de electrões e outra com um *deficit* destes: algumas pilhas, como as baterias dos automóveis, podem ser recarregadas quando o balanço de electrões se torna igual, enquanto outras, como as pilhas das lanternas, funcionam à base de produtos químicos que dificilmente são rejuvenescidos.

As duas áreas são ligadas a terminais colocados no exterior da caixa da bateria. O terminal que leva à área com o excesso de electrões é designado por «negativo», e marcamo-lo com o sinal de menos (−). À primeira vista este facto é ilógico, mas

representa tão-somente o resultado de ser assim que os primeiros investigadores marcavam as suas baterias, e a tradição enraizou-se de tal maneira que é muito difícil alterar esta situação. O terminal negativo é o ponto de «salto» para os electrões que estão «ansiosos» por passar para o outro terminal, este marcado com um sinal positivo (+).

Quase toda a gente sabe que uma das unidades usadas para medir a electricidade é o volt: este pode ser considerado como a pressão sobre os electrões para saírem dos seus átomos hospedeiros, demasiado superpovoados, de modo a que procurem uma morada mais acolhedora. Os electrões e os núcleos dos átomos comportam-se de uma forma semelhante à dos ímans: o pólo norte de um magneto é atraído pelo pólo sul de outro magneto, e vice-versa, enquanto dois pólos nortes têm o efeito inverso, repelindo-se mutuamente. Os electrões não gostam de estar perto de outros electrões: se um átomo contiver demasiados deles, tendem a fazer pressão uns sobre os outros para que se afastem e, se a oportunidade surgir, o excesso normalmente sai do átomo. Contudo, não se pense na voltagem como se fosse um mero excesso de electrões. Este é tão-somente uma indicação da capacidade da pilha e está relacionado com o tempo que ela poderá funcionar. A voltagem é a força com que um electrão tenta escapar do seu átomo.

CIRCUITOS ANALÓGICOS

É o momento de vermos o nosso primeiro circuito. Estudando a figura 1.1, reconhecem-se alguns dos componentes; a figura 1.2 é a mesma coisa desenhada com um diagrama do circuito, utilizando símbolos que tornam o desenho mais simples e o conjunto mais fácil de compreender, desde que saibamos qual o significado desses mesmos símbolos.

Como se observa, a pilha está ligada por fios às outras partes do circuito. Os electrões não se deslocam, normalmente, através do ar, a não ser com voltagem muito elevada, como no caso dos relâmpagos. Alguns materiais permitem movimentos relativa-

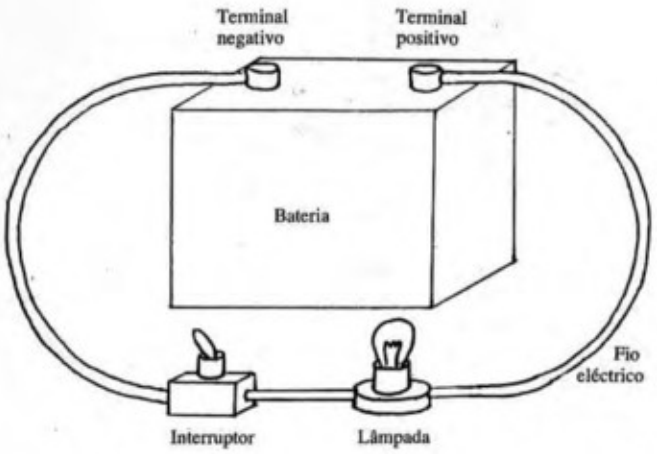


Figura 1.1

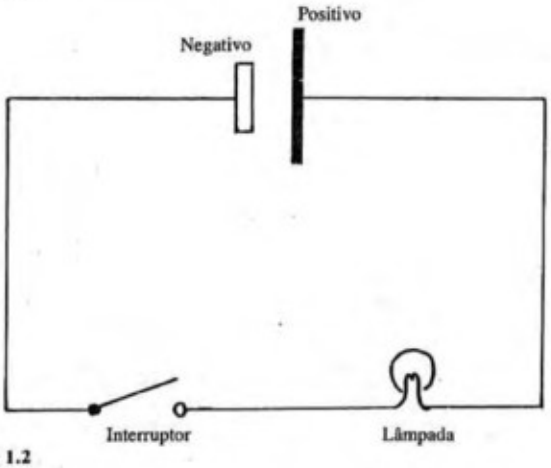


Figura 1.2

mente fáceis dos electrões livres, sendo por isso designados por «condutores»: se uma substância dificulta muito esta passagem, obrigando os electrões a lutar para conseguirem atravessá-la, estamos perante uma «resistência». Na verdade, todos os condutores possuem determinada resistividade, mas no caso dos fios de ligação, feitos de metal (em geral cobre), a resistência é tão pequena que podemos considerá-la irrelevante.

As lâmpadas contêm um tipo especial de fio, conhecido por «filamento», que possui a resistividade necessária para que os electrões só dificilmente passem pelos seus átomos, provocando o seu aquecimento até que brilhe e dê luz. O fio está encerrado numa ampola de vidro cheia de gases inertes, de modo que o filamento nada tem com que reagir, e por isso não funde. É a quantidade de fluxo através da lâmpada que determina o trabalho realizado, e portanto a intensidade de luz que o filamento produz. Este fluxo, designado por «corrente», mede-se em amperes.

O componente final é o interruptor, cuja operação é perfeitamente explicada pelo seu símbolo; proporciona uma descontinuidade mecânica no condutor, que se anula ao fechar o interruptor. Quando está aberto, a diferença total de voltagem do circuito está presente entre os seus dois contactos.

A figura 1.3 mostra o que acontece quando o interruptor se encontra fechado. A voltagem em presença provoca o movimento dos electrões através do fio, passando pela descontinuidade agora inexistente, até alcançarem o funil provocado pela natureza resistiva do filamento da lâmpada. Depois de atravessarem o filamento, os electrões estão novamente livres de movimentos, e por fim atingem o seu destino, o terminal positivo da pilha, no qual encontram um átomo que não está superpovoado, e à volta do qual podem finalmente orbitar.

Neste momento, já se deve ter uma ideia de como funciona o circuito eléctrico de uma lanterna. A este nível de complexidade há certa analogia com os fenómenos da água.

O terminal negativo da bateria pode ser considerado como um tanque de água colocado num alto. A voltagem está relacionada com a altura a que está o tanque e, portanto, com a pressão que

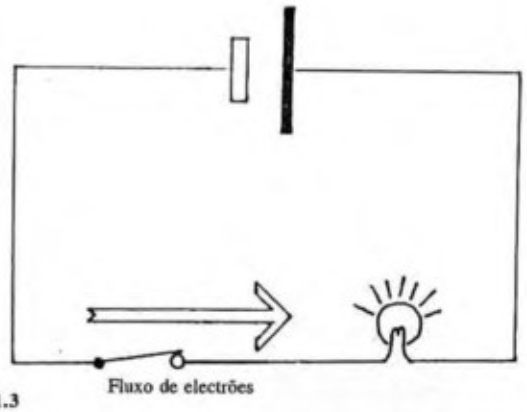


Figura 1.3

força a água através dos canos. O volume de água dentro do tanque é a capacidade da bateria. Se existir uma torneira, ela actuará como um interruptor, cortando o fluxo de água, e qualquer porção de cano muito fina restringirá a passagem do líquido, da mesma forma que o filamento da lâmpada limita a passagem da corrente.

Percebe-se agora que os três parâmetros do nosso circuito simples — voltagem, corrente e resistência — estão relacionados uns com os outros. Se ligarmos uma lâmpada com maior resistência ao circuito, o fluxo de corrente será menor; se usarmos uma pilha de maior voltagem, o fluxo será também mais forte.

Esta relação mútua determina-se matematicamente pela conhecida lei de Ohm. Esta diz que a corrente (em amperes) fluindo através do circuito é igual à voltagem (em volts), dividida pela resistência do circuito (medida em ohms). Isto significa que podemos calcular qualquer destes valores desde que conheçamos os outros dois. Em nenhuma parte deste livro serão exigidos cálculos deste tipo, mas confiamos em que, quando usarmos as expressões «voltagem», «corrente» e «resistência», os leitores se encontrarão esclarecidos e que não haverá lugar para atitudes de incompreensão.

CIRCUITOS DIGITAIS

Podemos agora deixar de lado grande parte da teoria da electricidade e dedicarmo-nos a assuntos mais práticos. Se dermos uma vista de olhos a qualquer moderno catálogo de componentes electrónicos, veremos uma secção dedicada a «circuitos integrados digitais»: estes aparecem sob designações tais como TTL e CMOS, o que pouco contribui para a sua compreensão. CMOS significa *complementary metal oxide semiconductor* (semicondutor complementar de óxido de metal), o que indica mais a maneira como são feitos do que a forma como funcionam. *Transistor transistor logic* explica um pouco melhor o que é um TTL. Estes componentes são os «tijolos» dos circuitos lógicos: alguns são já, em si, verdadeiros circuitos complexos prefabricados. Os componentes dos maiores computadores utilizam exactamente os mesmos tijolos, mas em maior número.

Para compreendermos os processos empregados num circuito digital é necessário saber algumas coisas sobre «semicondutores». Trata-se de substâncias que não aparecem na Natureza, pois são manufacturadas a partir de materiais altamente purificados, como o germânio e o silício, e que possuem propriedades específicas de grande utilidade. O componente mais simples que podemos fabricar com materiais semicondutores é o «díodo», que possui dois terminais e que só deixa passar corrente num sentido. Muito mais útil é o «transístor», cuja resistência varia quando se aplica voltagem a um terceiro terminal. Para usarmos mais uma vez a analogia da água, diríamos que o transístor se assemelha a uma torneira, em que a voltagem presente no terceiro terminal pode ser considerada como o parafuso da torneira, que controla o fluxo.

Nos circuitos analógicos, a precisão destes componentes é vital, pois a quantidade de fluxo que passa através do transístor é proporcional à voltagem de controle. Nos circuitos digitais, porém, para o estádio seguinte só interessa o facto de haver ou não voltagem presente, de modo que os transístores requerem uma manufactura menos precisa — actuam apenas como interruptores comandados electricamente.

Já falámos várias vezes sobre os circuitos estarem ligados ou desligados, e é agora a altura de mencionar outros termos. Se tivermos presente uma voltagem positiva, considera-se essa parte do circuito «alta» ou no «nível lógico um»; ao contrário, o lado negativo do circuito ou da fonte de abastecimento considera-se «baixo» ou no «nível lógico zero». Existe uma outra possibilidade, que aparece quando examinamos uma área não ligada a qualquer terminal da fonte de voltagem e que é designada por «flutuante».

Será útil descrevermos em pormenor o funcionamento de uma pastilha lógica, a porta AND (equivalente, em português, à copulativa «e») quádrupla *7409*, de duas entradas. Os nomes destes componentes indicam por vezes qual a sua função, e no caso vertente a chave reside na palavra «porta».

Uma «porta lógica» é um ponto em que se toma uma decisão. Não se conclua daqui que se trata de um raciocínio — perante um mesmo conjunto de circunstâncias, a mesma porta comportar-se-á sempre da mesma forma. O *7409* contém quatro destas portas (daí a designação de «quádrupla», no seu nome), e em cada uma existem dois terminais de entrada e um de saída.

Designa-se por «porta AND» porque se comporta da seguinte forma: se a primeira entrada for alta e se a segunda entrada também for alta, a saída será alta. Se ambas forem baixas, ou uma só for baixa, a saída será baixa.

Por que razão haverá quatro portas num só componente? É simples: o custo de produção é tão baixo que os fabricantes procuram aproveitar ao máximo o espaço disponível. A pastilha tem 14 pinos de ligação com o exterior — quatro conjuntos de três de acesso às portas e dois para o fornecimento de voltagem. A série *74* trabalha com uma voltagem positiva de 5 volts, e o terminal negativo é designado por «massa» ou «terra», ou simplesmente GND (do inglês *ground*).

Um método muito útil no estudo da lógica é o desenho de um esquema real. Esta técnica permite verificar o funcionamento de um circuito lógico, e vale a pena debruçarmo-nos sobre um muito simples para compreendermos a operação de uma porta AND.

No topo do esquema 1.1 estão os possíveis estados da primeira entrada da porta AND, e à esquerda, em baixo, os estados possíveis da segunda entrada. Seguindo a «entrada dois baixa» até ao ponto em que encontramos a «entrada um baixa», verificaremos que, nessa coluna, o cruzamento dá «baixa» — o estado da saída quando as duas entradas são baixas.

		Entrada Um: Baixa	Entrada Um: Alta
Entrada Dois	Baixa	Baixa	Baixa
	Alta	Baixa	Alta

Porta AND

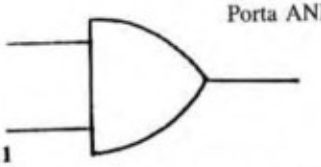


Tabela 1.1

Estudemos agora o esquema 1.2. Este descreve o comportamento de uma porta OR (em português a disjuntiva «ou»), em que a saída é alta quando a entrada um ou a entrada dois são altas. Os esquemas 1.3 e 1.4 apresentam-nos dois termos novos: NAND e NOR. Trata-se dos termos AND e OR com o prefixo N, que significa «não». Se desenharmos um esquema real para uma porta AND mas mentirmos cada vez que classificarmos uma saída, obteremos o comportamento de uma porta NAND. Uma das formas de se concretizar electronicamente este funcionamento seria a aplicação de um circuito designado por «inversor» ligado à saída de uma porta AND. Este é um dos mais simples componentes lógicos existentes, pois só tem uma entrada e uma saída — como o seu nome implica, a saída é um reflexo invertido da entrada, em que a alta é trocada pela baixa e vice-versa. Da mesma forma, podemos passar a saída de uma porta OR através de um inversor, para criarmos uma porta NOR.

Estudemos agora a figura 1.4, procurando reconhecer o que sucederá quando ligarmos o circuito, e as variantes provocadas pelos vários interruptores. O circuito representa um trinco, ou um simples *flip-flop*, e mostra como um impulso, sob a forma de uma pressão momentânea no interruptor (isto é, com um «alto» lógico aplicado por momentos a uma entrada da porta), provoca um resultado permanente. O circuito é formado por duas portas NOR, dois interruptores e dois outros componentes que impedem em grande medida a passagem da corrente, as resistências R1 e R2.

Repare-se na disposição do interruptor e da resistência. Se o interruptor estiver aberto, teremos zero volts no ponto em que o condutor que leva a uma entrada da porta está ligado ao interruptor e à resistência. Fechando o interruptor, a corrente passará através deste e da resistência, de modo que o ponto ligado à porta ficará com uma voltagem alta (positiva).

Sigamos o funcionamento das portas. Na primeira activação, e antes de qualquer dos interruptores estar fechado, ambas as

		Entrada Um: Baixa	Entrada Um: Alta
Entrada Dois	Baixa	Baixa	Alta
	Alta	Alta	Alta

Porta OR

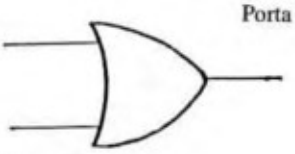


Tabela 1.2

		Entrada Um	
		Baixa	Alta
Entrada Dois	Baixa	Alta	Alta
	Alta	Alta	Baixa

Porta NAND

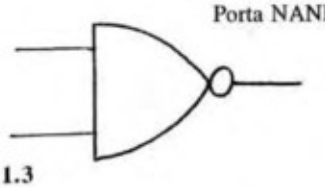


Tabela 1.3

		Entrada Um	
		Baixa	Alta
Entrada Dois	Baixa	Alta	Baixa
	Alta	Baixa	Baixa

Porta NOR

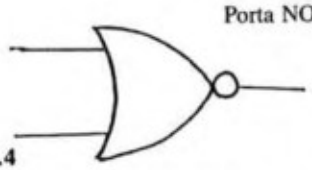


Tabela 1.4

entradas para a porta um estarão com zero volts. Comparando este estado com o esquema real da porta NOR, verificamos que temos uma voltagem alta no terminal de saída. As entradas para a

porta dois serão: baixa para o terminal ligado ao interruptor e alta para a ligada à saída um, do que resulta uma saída baixa para a saída dois. Este estado é igual ao da segunda entrada para a porta um, mantendo-o em voltagem baixa e assegurando a estabilidade do *status quo*, isto é, saída um alta e saída dois baixa.

Imaginemos agora o que sucede quando fechamos o interruptor marcado «SET». A primeira porta NOR fica com as entradas alta e baixa, do que resulta uma saída baixa. Isto vai afectar a porta dois, pois fica com as duas entradas baixas, de modo que a sua saída será alta. Esta vai por sua vez influenciar a porta um, provocando-lhe duas altas nas entradas; mas, como verificamos no esquema real, as saídas ficarão baixas. Ficamos assim com uma situação nova mas estável, em que os dois valores de saída estão permutados.

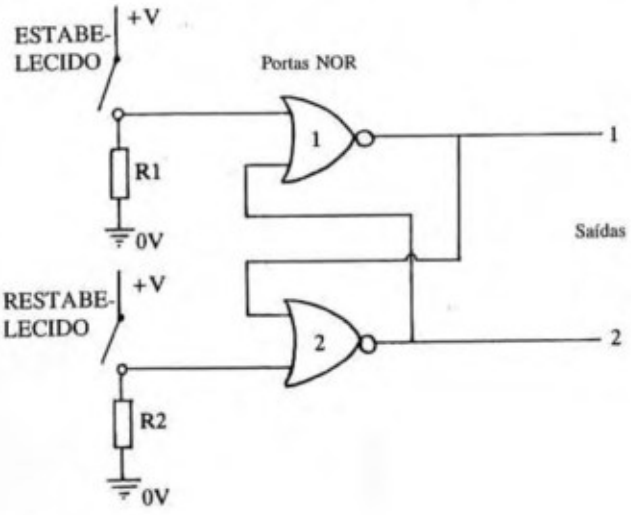


Figura 1.4

Vejamos agora o que sucede quando reabrimos o interruptor SET. A porta um tem uma entrada baixa e uma entrada alta, pelo que a saída se mantém baixa. A segunda porta não é afectada e as saídas ficam na mesma. Para que as saídas decresçam, necessitamos de premir o interruptor RESET. Tracemos os níveis lógicos que daí resultarão. Este circuito lembrar-se-á de qual foi o último interruptor premido — se foi o SET, então a saída um será alta. Podem adicionar-se circuitos extra para fazer este sistema funcionar como um armazém: na realidade lembrar-se-á de um nível lógico, uma característica que, como veremos adiante, é de extraordinária utilidade.

CAPÍTULO 2

Números e dados

Nos dias do telégrafo, as comunicações ao longo dos fios eram praticadas sob forma digital, com o código Morse, constituído por pontos e traços em lugar de uns e zeros. O processo era moroso e incómodo, pois a informação a enviar tinha de ser previamente codificada, transmitida através de uma longa série de pontos e traços e finalmente descodificada no outro extremo do fio. Se ao menos pudesse ser enviada qualquer coisa um pouco mais complexa que o sinal ligado/desligado, a transmissão teria sido muito mais fácil: por exemplo, transmitindo somente 26 sinais diferentes ao longo dos fios, todo o alfabeto poderia ser representado.

As técnicas analógicas e digitais resolveriam este problema de maneiras diferentes. Uma solução analógica poderia consistir na variação do comprimento dos traços, ou na colocação de uma voltagem de nível variável na linha; continuando nesta linha de raciocínio, acabaria por se desembocar no telefone.

A solução digital, contudo, consiste em adicionar mais linhas. Se uma linha se pode encontrar na situação de um ou de zero, duas linhas indicarão qualquer uma de quatro condições — ambas as linhas desligadas, linha um ligada e linha dois desligada, linha um desligada e linha dois ligada, ou ambas as linhas ligadas. Três linhas oferecem a possibilidade de oito permutações — as quatro das duas linhas somadas à terceira linha ligada ou desligada em cada situação.

Imaginemos um sistema de comunicações formado por cinco fios ligando dois aparelhos aptos a codificar e descodificar a informação. Isso permitiria dispor de 32 botões, etiquetados com as letras do alfabeto, para transmitir a mensagem. Um circuito digital bastante simples poderia então codificar a mensagem num único sinal de cinco linhas de ligados e desligados, transmitindo-os para o receptor. Este descodificaria o texto, podendo

responder acendendo uma determinada lâmpada. Este aparelho poderia ser designado, no calão da informática, por «sistema paralelo de cinco *bits* para transferência de dados»: isto é, passa informação (dados) de cinco *bits* (ou uns e zeros) através de cinco fios dispostos em paralelo.

NÚMEROS BINÁRIOS E HEXADECIMAIS

Consideremos agora o mesmo assunto de um ponto de vista mais teórico. A maneira como pensamos em números depende da nossa educação. Nós próprios só fomos introduzidos ao conceito de números em bases diferentes muito depois de estarmos demasiado agarrados ao sistema habitual, mas hoje as matemáticas modernas ensinam desde muito cedo os conceitos de números em bases que não a de dez. Para aqueles a quem a própria ideia de «base» causa confusão, deixamos aqui uma explicação.

Dispomos normalmente de dez números, que só precisam de um algarismo para os representar, de zero a nove. De cada vez que chegamos a dez, adicionamos para nova coluna, e por isso se diz que o sistema é de base dez. Mas porquê somente dez números? Se a raça humana tivesse evoluído com dezasseis dedos, poderíamos hoje dispor muito bem de mais seis algarismos para representar os números de 10 a 15, e então estaríamos todos a trabalhar na base 16.

Tente-se escrever números, por exemplo de 1 a 30, usando a base 16, com as letras A a F para representar os de 10 a 15, e compare-se o resultado com a tabela 2.1. Trata-se de números hexadecimais, um sistema numérico frequentemente favorecido pelos programadores de código máquina porque é mais fácil de ser traduzido para o «binário», que por sua vez é a maneira de que o computador se serve para lidar com números. Como o seu método de representação de dados só permite colunas contendo dois algarismos diferentes, o 0 e 1, o computador precisa de usar mais colunas de modo a representar um número maior que 1. Em lugar de unidades, dezenas, centenas, milhares, e por aí adiante, as colunas sobem num valor de potenciais de dois, isto é, 2, 4, 8, 16, etc. O *Spectrum*, tal como a maioria dos computadores domésticos, usa uma estrutura de dados de oito dessas colunas, e portanto pode lidar com números até 255; este é o número decimal que, transformado em número binário (ou número de base dois), se representa por 11111111.

Tabela 2.1: Conversão decimal para hexadecimal.

DEC	HEX	DEC	HEX
0	00	16	10
1	01	32	20
2	02	64	40
3	03	128	80
4	04	256	0100
5	05	512	0200
6	06	1024	0400
7	07	2048	0800
8	08	4096	1000
9	09	8192	2000
10	0A	16384	4000
11	0B	32768	8000
12	0C	65535	FFFF
13	0D		
14	0E		
15	0F		

Já aqui se disse que cada voltagem (um ou zero) se chama um *bit*, forma abreviada de dígito binário *(BInary digiT)*. As colunas ou linhas que transportam os oito *bits* distintos pelos circuitos do computador são colectivamente designadas por «vias de dados» *(data bus)*. O agrupamento de oito *bits* de informação para representar um simples dado é designado por *byte;* um microcomputador de oito *bits* trata um *byte* de dados de cada vez. Pode armazenar e comunicar com outros aparelhos, usando números dentro dos limites 0 a 255.

O interesse de se usar o sistema hexadecimal (abreviado para hex) reside no facto de um *byte* poder ser representado por um número de duas colunas, dentro do limite 00 a FF. Com um

pouco de prática, seremos capazes de converter mentalmente um número binário num número hex e vice-versa, e assim visualizarmos o modo como as linhas individuais de vias de dados se comportam, sem o inconveniente de ter de ler e escrever números de oito digitos. Se considerarmos o sistema hexadecimal absolutamente impossível, não somos obrigados a prosseguir, mas a familiarização com o sistema binário é de importância vital.

Incluímos aqui um programa que tanto serve para demonstrar a relação entre binário e decimal como para calcular a forma binária de um número decimal. Este programa, tal como todos os outros incluídos neste livro, está profusamente marcado por «instruções de anotação» (REMs), que omitiremos se quisermos poupar tempo.

Se bem que este primeiro programa não contenha código máquina, os mais avançados tê-lo-ão, o que significa que os poderemos perder completamente se os executarmos com um só erro de introdução de instruções ou de dados. A fim de se evitarem tais situações, recomendamos vivamente que se grave e verifique tudo o que demore mais de um minuto a introduzir. Como todos os programas começam na linha 10, poderemos fazê-lo usando a instrução «SAVE "nome" LINE 10»; deste modo, os programas serão executados automaticamente quando recarregados (ver o manual, capítulo 20).

```
  10 REM   Sistema Binario
  11 REM Programa 2.1
 100 REM
 101 REM  Inicializacao
 103 REM
 110 GO SUB 400: LET numero=0: G
O SUB 300
 115 REM
 116 REM      MENU
 117 REM
 120 PRINT AT 20,1;"Prima "; INV
ERSE 1;"C"; INVERSE 0;"alcular o
u "; INVERSE 1;"D"; INVERSE 0;"e
monstrar"
 130 LET a$=INKEY$: IF a$="" THE
N GO TO 130
 140 IF a$="d" THEN GO SUB 500:
GO TO 120
```



```
 150 IF a$="c" THEN GO SUB 600:
GO TO 120
 160 GO TO 130
 300 REM
 301 REM      Apresentacao de
 302 REM     um numero binario
 303 REM
 310 LET pot=128: LET resto=nume
ro
 320 FOR x=0 TO 7
 330 LET resultado=1: LET resto=
resto-pot
 340 IF resto<0 THEN LET resto=r
esto+pot: LET resultado=0
 350 LET z=x*4+1: LET cor=4-2*re
sultado: FOR y=3 TO 9: PRINT AT
y,z; PAPER cor;"  ": NEXT y
 360 PRINT AT 12,z;resultado
 370 LET pot=pot/2: NEXT x
 380 LET a$="00"+STR$ numero: LE
T a$=a$(LEN a$-2 TO LEN a$): PRI
NT AT 17,22;a$
 390 BEEP .5,numero/4: RETURN
 400 REM
 401 REM Apresentacao no Ecran
 402 REM
 410 BORDER 4: PAPER 6: INK 1: C
LS
 420 LET pot=256: FOR x=0 TO 7:
LET pot=pot/2: LET z=x*4+1: PRIN
T AT 1,z;"d";7-x;AT 2,z; INK 3;p
ot: PLOT 32*x+7,96: DRAW 0,55: P
LOT 32*x+24,96: DRAW 0,55: NEXT
x
 430 PRINT AT 0,0;"(";AT 0,31;")
": PLOT 6,174: DRAW 244,0: PRINT
 AT 0,12; INK 7; PAPER 1;"DATA B
US"
 440 PRINT AT 13,0;"(";AT 13,31;
")": PLOT 6,64: DRAW 244,0: PRIN
T AT 14,10; INK 7; PAPER 1;"NUME
RO BINARIO"
 450 PRINT AT 17,5;"Numero Decim
al--"
 460 RETURN
 500 REM
 501 REM  Demonstracao
 502 REM
 510 PRINT AT 20,1;"Prima "; INV
ERSE 1;"M"; INVERSE 0;"enu ou";
INVERSE 1;"F"; INVERSE 0;" para
congelar"
```

```
 520 FOR a=1 TO 255: LET numero=
a: GO SUB 300
 530 IF INKEY$="m" THEN RETURN
 540 IF INKEY$<>"" THEN GO TO 54
0
 550 NEXT a: RETURN
 600 REM
 601 REM    Calculo.
 602 REM
 610 PRINT AT 20,1;"Entre um val
or de ZERO a 255"
 620 INPUT "Numero Decimal ?";nu
mero
 630 LET numero=INT numero: IF n
umero>255 OR numero<0 THEN GO TO
 620
 640 GO SUB 300: RETURN
```

Depois de introduzido e gravado sem erros o programa 2.1, faz-se RUN. Aparecerá uma escolha entre uma demonstração e um cálculo.

A opção de cálculo pede um número entre 0 e 255, e depois apresenta-o sob duas formas. No topo do *écran*, existem oito linhas representando as vias de dados do computador. Se estiverem vermelhas, a voltagem é alta; se estiverem verdes (ou do mesmo tom que a moldura, se se usar um televisor a preto e branco), a voltagem é baixa. Horizontalmente, por baixo destas linhas, está uma fiada de zeros e uns, mostrando o número escolhido na sua forma binária.

No modo de demonstração, o computador percorrerá todos os números (0 a 255), e poderemos parar a listagem em qualquer altura premindo qualquer tecla que não a do *menu*, o «M».

MANUSEAR OS DADOS

Vimos já como um micro de oito *bits* trata os dados. Pode-se argumentar, com toda a razão, que o *Spectrum* também lida com palavras, e mesmo com números muito grandes ou muito pequenos. Contudo, estes são processados pelo programa incluído no computador — os números, por exemplo, são armazenados e manipulados como cinco *bytes* distintos de dados.

Mesmo a própria linguagem máquina que opera o computador emprega por vezes uma forma de código seriada; algumas das instruções da máquina aparecem como dois ou mais *bytes*, com o primeiro a dar instruções ao computador para que este se prepare para receber mais dados, e sob a forma de os tratar. Essas operações serão estudadas mais adiante. A transferência paralela de dados, por vezes, é feita para o exterior do próprio computador; por exemplo, algumas impressoras estão ligadas desta forma. Quando estão em jogo longas distâncias, ou quando a velocidade da comunicação não é tão importante, é mais usual recorrer-se a um processo de seriado. A codificação dos dados para uma forma mais conveniente pode ser processada pelo próprio programa fixo do computador.

Vimos como se processa o tratamento de dados sob a forma digital. Utiliza-se um método semelhante para dirigir o fluxo desses dados dentro do computador. Existe uma outra «via» *(bus)* dentro do *Spectrum*, designada por «via de endereço» *(adress bus)*, que mantém um número binário para indicar que área em particular pretende o computador tratar. Cada número é único, e portanto pode ser descodificado pela utilização de circuitos lógicos de forma a activar o endereço pretendido (lembremo-nos do acender de uma lâmpada no sistema de transferência de dados paralelo de cinco *bits* que mencionámos anteriormente). No próximo capítulo veremos como o computador manipula os dados usando tanto as vias de endereço como as de dados.

CAPÍTULO 3

O microprocessador

Há poucos anos, os computadores enchiam salas inteiras, e as suas diversas partes alojavam-se em módulos separados, de acordo com a função de cada uma. Ao mostrarem-nos uma dessas instalações, chamavam-nos a atenção para a «memória», aquelas enormes caixas cinzentas num dos cantos da sala, ou para o «leitor de cartões perfurados», noutro canto. O nosso cicerone teria mesmo apontado para uma das anónimas caixa cinzentas e dito «é aqui que se realiza todo o trabalho: trata-se da unidade central de processamento.»

Hoje, os computadores encolheram em tamanho, mas se olharmos para o seu interior ainda poderemos apontar para um circuito integrado e dizer: «É ali que se realiza o trabalho.» Estas UCPs ou «microprocessadores» contêm todos os componentes encontrados no antigo e enorme módulo cinzento, mas inseridos agora numa fatia de sílica de proporções diminutas.

O *Spectrum* possui um microprocessador de oito bits *Z80*, desenho desenvolvido pela Zilog no final dos anos 70. Trata-se de uma das concepções mais bem sucedidas, particularmente no campo dos microcomputadores para pequenas empresas, de modo que o usaremos como base das nossas descrições. Há outros microprocessadores mais ou menos aperfeiçoados, mas os princípios genéricos são sempre os mesmos. A colocação de todas as funções de processamento num só circuito leva-nos a considerar o microprocessador como uma espécie de «caixa preta», na qual sobressaem as saídas e entradas. É exactamente neste ponto que pretendemos começar.

«Z80» – O INTERIOR

Um microprocessador *Z80* tem 40 terminais. Se bem que alguns destes sejam para entradas e para saídas de voltagem, há-os que podem realizar ambas as funções.

Para começarmos pelos mais simples, estudemos os dois terminais (ou pinos) do circuito integrado que fornecem a energia necessária ao funcionamento. Para as operações internas e também para lançar dados para outros componentes, ele necessita de uma voltagem positiva de 5 volts, aplicada ao pino Vcc (entrada de voltagem), em contrapartida com o terminal GNR (terra ou massa). Quando está presente essa energia, uma pequena corrente atravessa o microprocessador, com uma intensidade variável de acordo com a função de momento a executar pelo *Z80*.

Adjacentes a estes terminais, encontramos os oito pinos de dados, D0 a D7, através dos quais a UCP passa *bytes* de dados de e para o mundo exterior. Todos estes são bidireccionais. Em algumas ocasiões, os oito pinos são colocados em alta ou em baixa pela UCP, de modo a colocar-se um *byte* de dados na via de dados, para ser usado em outro lugar distinto. Outras vezes, o *Z80* recebe e lê dados colocados na via por outros componentes, através dos terminais de dados; quando tal sucede, esses terminais ficam flutuantes.

Este tipo de terminal é de grande utilidade, e muito comum em outros *chips* de computadores — permite ligar muitas coisas à mesma via de dados que percorre as linhas. Aos *chips* construídos com este tipo de terminais chamamos «componentes de três estados» *(tri-state)*.

O CONTROLE DOS DADOS

Temos assim um método de passar dados para dentro e para fora do *Z80*, e precisamos agora de controlar o seu destino. Para este fim, os microprocessadores estão equipados com linhas de endereços, que lhes permitem indicar com qual dos componentes exteriores pretendem comunicar. O *Z80* tem 16 dessas linhas de endereço. Uma das possibilidades mais perfeitas desta UCP é que está desenhada para poder dirigir-se a dois tipos distintos de periféricos de maneiras diferentes: são eles os «sistemas de memória», que armazenam dados, e os «portos», que comunicam com o mundo exterior.

Dois dos terminais, o IORQ *(input/output request)* e o MREQ *(memory request)*, indicam com que classe de dispositivo a UCP pretende lidar. Se o MREQ estiver a zero volts, a UCP dirige-se à memória: as linhas de endereço mantêm um número binário de 16 *bits*, que indica qual o ponto da memória com o qual a UCP tem intenções de comunicar. Oito *bits* de dados dão-nos 256 combinações possíveis: isto é, 2 multiplicado por si próprio 8 vezes, o que é o mesmo que 2 elevado à oitava potência, que podemos escrever com 2^8; 2 elevado à 16.ª potência dá 65536, e este número é normalmente o máximo de localizações de memória (ou pontos) a que o *Z80* se pode dirigir, tal como 16 é o número de linhas de endereço. Digo normalmente, porque, com circuitos adicionais e alguns «truques», há microcomputadores (mas não o *Spectrum*) que conseguem ligar-se a vários dispositivos de memória, o que permite aumentar a capacidade de armazenamento de dados.

Vale a pena discorrermos aqui sobre uma parte do calão dos computadores, capaz de provocar alguma confusão. A publicidade e a literatura de informática dizem frequentemente que um computador tem uma determinada capacidade de memória, por exemplo, 16 k *bytes*. Como o k é a abreviatura internacionalmente aceite para o milhar, seria natural assumir que 16 k significam 16 000 *bytes*. Contudo, 1 k de memória representa na realidade 1024 *bytes*, o total de memória que pode ser atingido utilizando 10 *bits* (isto é, 2^{10}). Esta pequena discrepância torna os 64 k a capacidade máxima normal dos microcomputadores baseados no *Z80*, num total verdadeiro de 65 536 *bytes*.

Dois outros pinos, designados por RD e WR *(read* e *write*, ler e escrever), indicam aos circuitos de memória se o *Z80* precisa de dados da memória ou se está a colocar um *byte* de informação na via de dados, que pretende que seja armazenado num determinado local, especificado pela via de endereço. Estes terminais são também utilizados quando a UCP se dirige a um porto. Já vimos que a UCP comunica com a memória quando o MREQ está a 0 Volts; quando o IORQ está baixo, os pinos RD e WR desempenham a mesma função que para a memória, aplicada agora a um

porto, cujo endereço é mantido nas oito linhas de endereço baixas. Isto significa que o número máximo de portos que o *Z80* pode processar é de 256, número que se nos está a tornar familiar; e, tal como se passa com a memória, o IORQ e a via de endereço podem ser descodificados para activar os circuitos pretendidos. Contudo, podemos obter espaço adicional no endereçamento de portos, facto que será tratado na secção dedicada ao teclado.

Começa agora a revelar-se a essência de um sistema de computação, mas, para compreender melhor o funcionamento da UCP, temos ainda de considerar dois dos pinos restantes. O primeiro é designado por CLK, abreviatura de *clok* (relógio): é o «coração» do processador. Para que o *Z80* trabalhe, é necessário que este pino seja alternadamente colocado em tensão alta e baixa por uma fonte de voltagem exterior. Isto consegue-se por um circuito designado por «oscilador», que gera uma voltagem variável sob a forma de onda quadrada. A saída deste circuito é alta durante um período de tempo fixo, e então cai para baixa durante igual intervalo, repetindo-se o ciclo continuamente.

Estando interessado em saber por que razão se chama assim a esta saída, introduza-se e faça-se correr o programa 3.1. Este desenha um gráfico de corrente em função do tempo, se bem que a velocidade da simulação seja muito inferior à do funcionamento real do computador. A sequência da voltagem a subir, mantendo-se alta, e descendo de repente, para novamente voltar a aumentar, chama-se «ciclo», e a velocidade de alternação da voltagem é medida pelo número de ciclos completados num segundo. Foi Hertz quem que deu o nome a esta unidade de medida tão importante. Os amantes da alta-fidelidade saberão com certeza a frequência dos sons que os seres humanos podem ouvir: surge nos limites dos 30 Hz até mais de 16 kHz, ou dezasseis mil ciclos por segundo. A velocidade de operação do oscilador do *Spectrum* é muito maior — 3,5 MHz (M=mega — significa um milhão).

Programa 3.1. Onda quadrada

```
10 REM Programa de tracado
11 REM de Onda Quadrada
```

```
  12 REM Programa 3.1
 100 REM Inicio
 101 REM
 110 PAPER 0: INK 6: BORDER 0: C
LS
 120 PRINT "VOLTS = ": FOR x=1 T
O 6: PRINT AT x+3,0;6-x: NEXT x
 130 INK 3: PLOT 7,151: DRAW 0,-
120: DRAW 247,0: INK 6
 140 PRINT AT 1,17;"Onda Quadrad
a"
 150 DIM a$(9)
 200 REM
 201 REM  Curva de Tracado
 202 REM
 210 FOR x=0 TO 400
 211 REM
 220 REM Calculo da voltagem
 221 REM
 230 LET volts=2.5+10*SIN (x/40)
 240 IF volts>5 THEN LET volts=5
 250 IF volts<0 THEN LET volts=0
 300 REM
 301 REM Tracado dos valores
 302 REM
 310 PLOT INK 7;8+(24*x/40),32+v
olts*24
 320 REM
 321 REM Impressao de valores
 322 REM
 330 PRINT AT 0,7;volts;a$
 340 PRINT AT 21,15;x/40;a$
 350 REM
 351 REM Fim do loop
 352 REM
 360 NEXT x
 370 STOP
```

De cada vez que o microprocessador «sente» que a entrada do relógio está a ser levada de baixa para alta, executa a tarefa seguinte, de modo que o relógio actua como um metrónomo para a UCP, regulando as suas acções e forçando-a a executar o passo de programação que se segue. Com todos os seus complexos circuitos, por que razão um microprocessador não poderá regular a sua própria velocidade, ou mesmo trabalhar tão depressa quanto puder? A resposta está na necessidade dos construtores de controlarem a UCP, o que conseguem através do sinal externo do

relógio e lhes permite construírem um computador no qual todos os circuitos associados possam acompanhar o ritmo de funcionamento do microprocessador central. Dá-lhes também a possibilidade de parar o relógio num ponto qualquer, para imobilizar a acção até o oscilador ser novamente posto em funcionamento.

O pino que nos falta referir é o *reset* (restabelecer). Se o ligarmos a zero Volts, ele comporta-se exactamente como o seu nome implica — faz com que a UCP pare de executar o que quer que estivesse a fazer e obriga-a a recomeçar a partir de um ponto predeterminado.

Vejamos o que acontece quando o *Spectrum*, ou qualquer outro sistema de microcomputação baseado no *Z80*, é ligado pela primeira vez.

Todos os circuitos começam a receber a voltagem de que precisam para funcionar; o oscilador passa imediatamente a aplicar à UCP o sinal de onda quadrada do relógio. No entanto, um circuito electrónico muito simples assegura que, nos primeiros instantes após a ligação à corrente, o pino *reset* tenha uma tensão de zero Volts, o que obriga o processador a começar a trabalhar a partir do ponto correcto.

É possível verificar o que aconteceria se isto não sucedesse, ligando e desligando muito depressa o computador. Se não se der ao sinal de restabelecimento tempo suficiente para actuar, o computador pode ficar «pendurado» — mostrará uma imagem muito peculiar no *écran*, e recusar-se-á a responder ao teclado. Quando o sinal *reset* é correctamente gerado, mantém o pino respectivo nos zero Volts durante o tempo suficiente para que a UCP se prepare para trabalhar. Só a seguir o circuito de retabelecimento deixa que o pino apropriado suba para os cinco Volts, e então o *Z80* pode iniciar as suas tarefas.

Depois do *reset*, o *Z80* faz sempre a mesma coisa — coloca o número zero na via de endereço, de modo que todas as 16 linhas de endereço são colocadas em zero Volts pelos pinos de endereçamento, MREQ e RD. Assim, o *Z80* gera informação suficiente para dizer aos seus circuitos associados que necessita que os

dados colocados na localização da memória zero sejam colocados nas vias de dados, e deste modo os circuitos de memória começam a executar esta instrução. O processador espera pelo impulso seguinte do relógio (a transição baixa para alta do circuito do oscilador). Parte então do princípio de que os dados presentes na via de dados são os fornecidos pela memória, e lê-os através dos pinos de dados, D0 a D7. Aquilo que na realidade a UCP obteve da localização de memória zero é a sua primeira instrução em código máquina e, usando sempre como referência os impulsos do relógio, o processador continua a executar o resto da tarefa que as instruções seguintes lhe ditarem.

Os processos pelos quais o *Z80* armazena dados na memória e troca informação com os circuitos que são activados pelo pino IORQ devem ser considerados como variantes da anterior descrição do modo de obter dados da memória. Chegou portanto a altura de analisarmos o conteúdo da própria UCP, o que não é tão complicado como parece à primeira vista; e, estando-se interessado na natureza do código máquina e na estrutura interna do *Z80*, é assunto essencial.

CAPÍTULO 4

Dentro da caixa

O interior do microprocessador *Z80* pode ser dividido num certo número de áreas distintas. O modo como algumas delas funcionam não diz respeito à pessoa que pretenda escrever programas em código máquina, mas uma visão do conjunto será benéfica se se quiser descobrir o que na realidade acontece quando o programa está a ser executado.

OS REGISTOS

O processador contém um certo número de registos. Um destes consiste em oito circuitos do tipo *flip-flop*, cada um capaz de armazenar um *bit* de informação, e ligados de tal modo que o conjunto pode suportar um *byte* de dados — na realidade, são as células de memória próprias da UCP. Os registos estão todos ligados a uma via de dados interna. (Nota: esta não é a mesma via de dados que o *Z80* usa para comunicar com as zonas exteriores de memória e de portas.) O que surgir nestas linhas nem sempre é comunicado aos pinos de dados, se bem que estes também transportem informação de e para os terminais sempre que for necessário.

Ligado igualmente à via de dados interna está um circuito designado por «unidade aritmética e lógica», ou ULA. É nesta que se efectua todo o processamento dos *bytes* de dados e que a UCP pode adicionar ou subtrair dois números binários, bem como efectuar operações lógicas como a ligação AND entre dois *bytes*. O conjunto destas operações é governado por um circuito complexo a que se chama a «unidade de controle», que decifra as instruções em código máquina obtidas da memória e reage de acordo com elas.

A figura 4.1 merece um estudo muito atento; representa a estrutura interna do processador *Z80*. A unidade de controle pode manipular todos os registos, bem como enviar os sinais apropria-

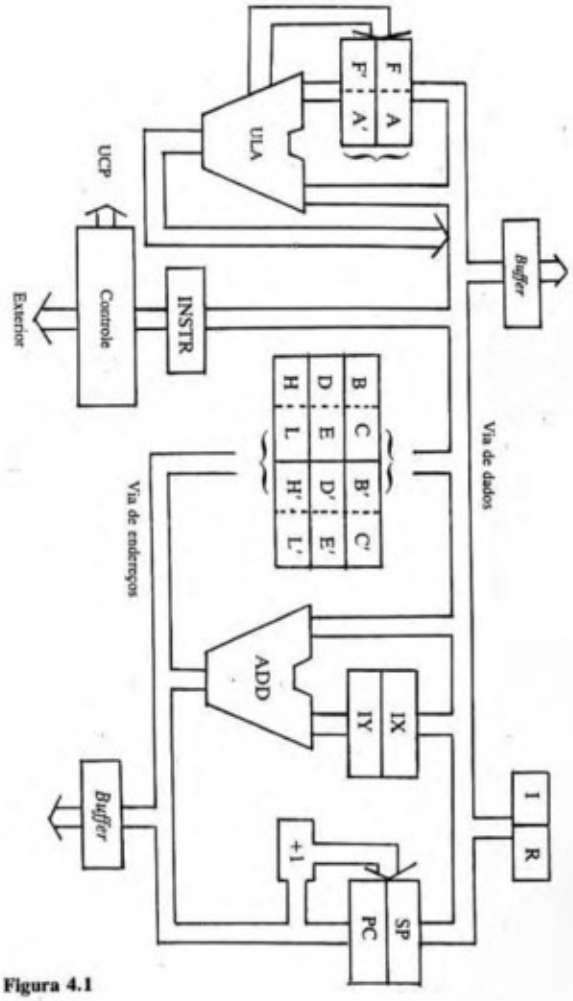


Figura 4.1

dos aos circuitos externos e aos circuitos de controle de vias. O «registo de instruções» só pode ser usado pela unidade de controle e tem a função de armazenar a última instrução obtida da memória externa. Todos os outros registos podem ser utilizados pelo programador, mas uns são mais versáteis que outros.

O registo PC (contador de programa ou *program counter*) guarda o endereço da próxima instrução a interpretar. Restabelecida a UCP, este registo é carregado com o valor zero, que é a localização da primeira instrução. Obtida esta, a unidade de controle adiciona automaticamente o valor um ao valor do registo PC, o que significa que estará a armazenar o endereço a partir do qual receberá os dados subsequentes. Se o programador não utilizar certas instruções de «salto» para recarregar o PC com um valor diferente, os dados em código máquina serão sempre lidos sequencialmente. O PC é um registo de 16 *bits*: pode fornecer qualquer um dos 64 k endereços à via de endereçamento.

O bloco principal de registos de utilização surge-nos em dois grupos, ambos usando as mesmas iniciais como nome, mas com um deles referenciado pelo sinal ' para indicar que se trata do grupo alternativo. Em dado momento, só um destes conjuntos está activado; instruções especiais em código máquina permitem determinar qual dos grupos está ligado à via de dados interna, de modo a poder reter ordens subsequentes. Para todos os efeitos, o grupo alternativo pode ser considerado como um útil espaço de armazenamento, particularmente se quisermos parar o processamento de uma tarefa e executar uma outra (caso em que os valores nos registos podem ser preservados colocando-os no outro grupo, até se estar pronto a regressar à primeira tarefa).

O «acumulador», ou registo A, é o mais versátil dos registos de utilização, porque lhe podem ser aplicadas todas as funções aritméticas e lógicas. Um valor guardado em qualquer outro registo de utilização ou armazenado na memória externa pode ser adicionado, subtraído ou aplicado nas formas AND, OR ou XOR (OR exclusivo) ao A, que passará a conter o resultado da operação.

Determinados resultados dos cálculos da ULA revestem-se de

particular interesse, por exemplo se o seu produto final é zero. Para sua própria utilização, ou para servir os propósitos do programador, ela anota electronicamente algumas destas ocorrências e guarda-as como *bits* simples de dados no registo F, por vezes designado pelo seu nome completo, *flags* (bandeiras ou etiquetas). Este nome provoca uma analogia um tanto ou quanto estranha para um computador — imagine-se que uma pequena bandeira é içada num mastro marcado por «Z» todas as vezes que um cálculo resulta em zero. Entre outras coisas, o registo F também toma nota de todas as operações que dão um determinado «resto», isto é, se a resposta excede um valor de oito *bits*.

Os outros registos de oito *bits*, de utilização genérica, os B, C, D, E, H e L, servirão como espaços de armazenamento de valores a que a UCP necessite de chegar rapidamente. Podem também funcionar como registos de 16 *bits* emparelhando-os em registos BC, DE e HL — particularidade muito útil quando pretendemos manipular valores de endereço. Quando aos pares, suportam operações aritméticas limitadas, cujos resultados vão parar ao registo HL. Um outro papel atribuído aos pares de registos é o de «apontadores de memória»: isto significa que, por exemplo, podemos carregar o registo A com determinado valor contido numa localização da memória, cujo endereço será preservado no registo HL.

Existem dois registos especiais de 16 *bits* cuja finalidade é reter endereços de memória importantes — isto é, importantes para o programador. Se este carregar os registos IX ou IY com um endereço apropriado, terá imediato e fácil acesso aos endereços de memória dentro de um limite situado entre as 128 localizações abaixo e as 127 acima desse endereço, que para tal basta ser retido em qualquer desses «registos de índice». Dois dos registos menos utilizados são os designados por I e R. O I, ou *interrupt vector* (vector de interrupção), altera o comportamento da UCP quando esta sofre uma interrupção, assunto que trataremos mais tarde. O R, ou *refresh* (de refrescar), é utilizado com frequência quando o *Z80* trabalha em conjunto com «memórias dinâmicas». Também este caso será explicado mais adiante.

O último registo que iremos considerar é o SP, ou *stack pointer*. Este é outra forma de «ponteiro de memória», e permite a aplicação de técnicas de programação já bastante complexas. Os leitores familiarizados com a programação em BASIC saberão o que é uma sub-rotina, através da qual se pode fazer saltar o programa para uma determinada linha; o programa lembra-se de onde veio e para lá volta sempre que encontra uma instrução RETURN. O mesmo princípio pode ser utilizado no código máquina *Z80*, com o endereço de retorno guardado na localização de memória indicada pelo ponteiro de pilha. Este registo tem outras aplicações, que serão descritas no capítulo dedicado ao código máquina.

A partir deste ponto, já se compreenderá o modo como a UCL troca informações com o mundo exterior, para além de haver uma ideia geral da sua estrutura interna. Passemos agora revista, lentamente, aos processos de execução de um programa muito simples, que para o que pretendemos estará armazenado na memória externa, começando no endereço zero. A tabela 4.1 dá os valores do exemplo.

Tabela 4.1

Endereço	Conteúdo
0	62
1	8
2	33
3	0
4	88
5	61
6	202
7	15
8	0
9	54
10	0
11	35
12	205
13	5
14	0
15	116

EXECUÇÃO DE UM PROGRAMA

A partir do momento em que o pino *reset* deixa de ser mantido em baixa pelo circuito de restabelecimento, o *Z80*, durante os dois primeiros ciclos do relógio, vai buscar um *byte* de dados ao endereço de memória registado no contador do programa. Esse *byte* passa ao longo da via de dados interna até ao registo de instruções, onde é armazenado para vir a ser usado pela unidade de controle. Estes dados constituem a primeira instrução em código máquina do programa (ou código *op*, abreviatura de código de operação) — e a unidade de controle descodifica-se e reage de acordo com o que ela determina.

No nosso programa de exemplificação, o primeiro código *op* é o decimal 62, que diz à unidade de controle para ir buscar o próximo grupo de dados guardados na memória, colocando-os no registo A. Decorrem dois ciclos enquanto esta instrução é «compreendida». Entretanto, a unidade de controle aumenta o valor do contador do programa em uma unidade, passando portanto a reter o valor um. Na realidade, a unidade de controle pode perfeitamente executar outras tarefas relacionadas com os circuitos dinâmicos de memória ao mesmo tempo, mas isto não nos diz respeito.

De volta ao nosso exemplo, a UCP, «consciente» de que aquilo que está colocado no endereço um não é outra instrução, mas dados, executa outra operação de busca à memória e regista o valor obtido no registo A, conforme lhe tinha ordenado o código *op* anterior. Aumenta simultaneamente o contador do programa. Qualquer que seja o dado retido no endereço um, neste caso o decimal 8, é sempre armazenado no registo A. A UCP completou assim a tarefa determinada pela primeira instrução, de modo que vai buscar a próxima à memória, utilizando o endereço (neste caso, 2) registado no PC. O código *op* seguinte do programa é imediatamente transferido para o registo de instruções e vai provocar nova actuação. É o decimal 33, que indica à UCP que deve colocar os dois *bytes* de dados seguintes, respectivamente, nos registos L e H. Portanto, o registo L contém o que estava armazenado no endereço 3 (0); e guarda em H o dado

contido no endereço 4 (decimal 88). Lembremo-nos de que se adiciona uma unidade ao PC de cada vez que um *byte* é obtido da memória.

Chegamos assim à instrução contida no endereço 5, que é obtida da mesma maneira. Trata-se do decimal 61, que representa a tarefa simples de subtrair um ao valor do registo A; depois de ser executada, em A fica o valor decimal 7. Sigamos para o endereço 6, que guarda o valor decimal 202. Este é um código *op* condicional, e podemos descrevê-lo como «vai buscar os próximos dois *bytes* de dados deste programa e, se (IF) o resultado da última operação aritmética ou lógica for zero, guarda os dados obtidos no contador do programa». A UCP está aqui a tomar a sua primeira «decisão», baseada no conteúdo do registo F. Até aqui, só fizemos a UCP realizar uma operação aritmética, um passo de programa atrás, quando subtraiu 1 ao conteúdo do registo A. Contudo, o resultado foi 7 e não 0. Portanto, se bem que os próximos dois *bytes* sejam obtidos dos endereços 7 e 8, nada é executado com eles, pois a bandeira Z não está activada. Em breve veremos o que acontece neste último caso.

O PC contém agora o valor 9; por isso a UCP carrega o conteúdo deste endereço, que é o decimal 54, no registo de instruções. A ordem é «vai buscar o próximo *byte* do programa, considera-o como dados, e coloca-o no endereço de memória representado pelo conteúdo do par de registos HL». Deste modo, o valor zero é obtido do endereço 10 e a unidade de controle dedica-se à tarefa de o ir armazenar na memória.

O primeiro passo é a colocação do conteúdo do registo L nas oito linhas de endereço baixas, e o conteúdo do registo H nas oito linhas de endereço altas. Isto significa que a via de endereço contém o valor decimal 22528, que é 256 vezes o valor de H (88) somado ao conteúdo de L, zero neste caso. É então que a unidade de controle coloca a linha MREQ baixa, activando os sistemas de memória externos, e coloca na via de dados o valor que acabou de obter da área de memória do programa, que é zero. Espera por um ciclo do relógio para que tudo se ajuste, e então coloca a linha WR (escrever) em estado baixo. A memória externa considera

isto como um sinal para armazenar os dados na via de dados e o endereço na via de endereços. Qual é o resultado? Os dados que estavam guardados no endereço de memória 22528 foram agora apagados e substituídos pelo dado zero.

Com o PC agora em 11, a UCP atingiu a instrução «decimal 35», que adquire e a faz reagir adicionando um ao valor do registo HL. O valor do endereço 12 é a versão «incondicional» de uma que já encontrámos anteriormente, no endereço 6. Desta vez, diz-se à UCP para carregar o PC com os dois *bytes* seguintes. Por isso, como as localizações 13 e 14 contêm 5 e 0, respectivamente, estas são adquiridas, e ao PC é somado o valor 5 (a parte baixa do PC é carregada com o primeiro *byte* obtido, e a parte alta com o segundo).

De onde virá a próxima instrução? O endereço prévio do PC foi reescrito, por isso a próxima ordem vem, não do endereço 15, mas do 5. A UCP executou uma instrução de salto: carrega-se com o código *op* do endereço 5, que é 61, instrução esta que já tinha sido executada anteriormente. O programa criou um ciclo (*loop*) — continuará a executar os códigos *op* do endereço 5 até ao 12, se bem que não o faça indefinidamente, devido à instrução condicional do endereço 6. De cada vez que se cumpre o ciclo, o valor do registo A reduz-se de uma unidade. Depois da oitava execução do código *op* 61, o registo A terá o valor zero, de modo que quando a UCP verificar que está o *bit* zero no registo *flags*, aperceber-se-á de que, finalmente, a operação prévia deu um resultado zero.

Esta realidade implica a passagem à segunda parte da operação ditada pelo código *op* do endereço 6 — se recapitularmos a explicação veremos que, no caso de o resultado ser zero, o PC terá de ser carregado com 15. Depois de executar o ciclo oito vezes, o programa salta para a instrução colocada em 15. Acontece que esta é a última, e vai indicar à UCP que o programa foi completado.

Usando a tabela 4.1 para avançar através de cada passo do programa, com cuidado, acabaremos por compreender perfeitamente o que está a acontecer dentro do processador. Nesta altura,

não é preciso compreender em pormenor o próprio programa, sendo mais mais importante visualizar quais os passos dados pela estrutura interna do processador *Z80*, de modo a compreender as instruções que ele obtém da área de memória externa que contém o programa.

Ora bem, o programa funcionará? E que faz ele na realidade? Para responder a estas questões, porque não pô-lo a correr no *Spectrum?* Para isso, é necessário, em primeiro lugar, carregá-lo na memória. Como não podemos carregar dados nos primeiros 16 k da memória, pois estes, no *Spectrum*, estão cheios com o programa próprio do computador; temos de o fazer numa área mais alta da memória. Por consequência, e como a primeira instrução não fica no endereço zero, não poderemos servir-nos do pino de restabelecimento para levar o contador do programa até esse endereço. Contudo, há uma maneira de tornear este problema.

Uma característica do *Spectrum* é poder executar código máquina a partir do BASIC, através da função USR. Isto consta do programa 4.1, sob a forma «RANDOMIZE USR 28672», e funciona da seguinte maneira: o endereço que o programa em BASIC atingiu durante a sua execução é armazenado no endereço indicado pelo ponteiro de pilha, e o contador do programa é carregado com o valor 28672, de modo que a UCP começa a correr o programa guardado neste endereço. Quando, por fim, a UCP encontra o código *op* «decimal 201», recarrega o valor original do PC da área de memória apontada pelo ponteiro de pilha (PP). Isto assegura que a UCP regressa ao programa em BASIC que estava a executar. O BASIC aproveita assim o valor que está contido no registo BC da UCP, e processa-o como se fosse o resultado da função USR.

A consequente prática deste facto é que, quando ditamos á instrução RAND USR e logo a seguir um endereço de memória válido, em qualquer programa em BASIC que escrevermos, o *Spectrum* vai até esse endereço e executa qualquer código máquina que aí esteja, até encontrar a instrução em código máquina RET (decimal 201); alimenta então o gerador de

números aleatórios com o valor guardado no registo BC. É uma forma muito conveniente de chamar uma rotina; se em vez disso usarmos o PRINT USR, o resultado do registo BC aparecerá no *écran*.

Tabela 4.2. Código máquina para o programa 4.1.

Endereço	Código hex	Assembler
7000	3E08	LD A,08h
7002	210058	LD HL,5800h
7005	3D	DEC A
7006	CA0F70	JP Z,700Fh
7009	3600	LD (HL),00h
700B	23	INC HL
700C	C30570	JP 7005h
700F	C9	RET

Introduza-se e grave-se agora o programa 4.1. Quando o fazemos executar, ele vai em primeiro lugar colocar o código máquina das instruções DATA na memória, começando no endereço 28672. Pede-nos então para premir uma tecla e, quando o fazemos, salta para o programa e executa-o. Deverá aparecer uma linha grossa e preta no canto superior esquerdo do *écran*. Repare-se na rapidez com que ela aparece; o código máquina colocou o valor zero em sete locais da memória, de 22528 a 22535. Escolhemos estes principalmente porque constituem a primeira parte do chamado «arquivo de atributos». Este controla as cores e os locais em que aparecem no *écran*, característica que será explicada nos capítulos dedicados à apresentação de dados no pequeno *écran*. Poderemos fazer a experiência de mudar o valor do registo A (o segundo número da instrução DATA), o valor carregado no ficheiro de atributos (o décimo primeiro da declaração DATA), ou mesmo o ponteiro de memória do próprio registo HL (mas neste caso teremos de apagar a linha 60, antes que actuem outros valores). Se tornarmos o HL mais baixo do que 16348 não obteremos qualquer resultado, e se lhe aplicarmos um valor superior a 23296 não só não veremos nada como

poderemos provocar um *crash* do computador, pois interferiremos com o seu funcionamento. Este «desastre» não provocará qualquer avaria no *Spectrum*, mas talvez nos obrigue a desligar e a ligar novamente a máquina, para prosseguirmos.

Na verdade, este programa em código máquina não é só muito simples; é também tosco. Consegue provar, no entanto, que poderemos tornear o sistema operativo do *Spectrum* e explorar as potencialidades da programação em código máquina.

Programa 4.1. Demonstração de código máquina

```
  10 REM   Demonstracao de
  11 REM   Codigo Maquina
  12 REM Programa 4.1
  20 REM    Iniciacao
  21 REM
  30 PAPER 7: INK 0: BORDER 7: C
LS
  40 REM
  41 REM  Verificacao de Dados
  42 REM
  50 RESTORE : LET sum=0: FOR x=
0 TO 15: READ a: LET sum=sum+a:
NEXT x
  60 IF sum<>1183 THEN PRINT "Os
 dados estao errados."'"Verifiq
ue se os introduziu"'"correctame
nte": STOP
  70 REM
  71 REM Colocacao do Codigo na
  72 REM       Memoria
  73 REM
  80 RESTORE : FOR x=28672 TO 28
687: READ byte: POKE x,byte: NEX
T x
  90 REM
  91 REM  Espere pela accao da
  92 REM        tecla
  93 REM
 100 PRINT AT 16,6;"PRIMA UMA TE
CLA"
 110 IF INKEY$="" THEN GO TO 110
 120 REM
 121 REM Salto para a Rotina
 122 REM
 130 RANDOMIZE USR 28672
 140 REM
 141 REM    Regresso
 142 REM
```

```
 150 PRINT AT 16,6;"Programa Com
pleto"
 160 STOP
 500 REM
 501 REM    Codigo Maquina
 502 REM
 510 DATA 62,6,33,0,88,61,202,15
 520 DATA 112,64,0,35,195,5,112,
201
```


CAPÍTULO 5

RAM e ROM

Nos capítulos anteriores, mencionámos frequentemente a «memória externa», sem qualquer definição ou clarificação. Agora, que já percorremos os principais processos de operação do *Z80*, é, porém, tempo para esse esclarecimento. O *Spectrum* usa a memória para armazenar os dados de que necessita para funcionar, quer se trate do código máquina, que ele utiliza mal é ligado, quer de um simples programa deste livro ou da última versão dos Invasores Espaciais.

TIPOS DE MEMÓRIA

Os microcomputadores usam essencialmente dois tipos de memória. A «Read Only Memory» (memória exclusiva de leitura), ou ROM, não permite ao microprocessador alterar os seus dados — como o seu nome implica, só pode ser lida. O outro tipo é designado por «Random Access Memory» (memória de acesso aleatório), ou RAM, o que é uma designação ligeiramente confusa porque a sua característica principal é a possibilidade de se conseguir alterar o conteúdo sob controle do computador. E, por outro lado, esta situação não é tão simples como aparenta: tanto na RAM como na ROM aparecem vários tipos de dispositivos de memória. Comecemos por analisar o tipo de ROM do *Spectrum*, usado para guardar o programa de código máquina com que já vem fornecido e que lhe permite funcionar.

Esta ROM possui 14 pinos de vias de endereço, etiquetadas de A0 a A13. Quando o terminal CS (Chips Select, «selecção de *chip*») da memória é colocado em baixa tensão, o conteúdo de um registo de oito *bits* dentro do *chip*, e que corresponde ao endereço do momento presente nos pinos de endereço, passa para os pinos de dados do dispositivo de memória (e por consequência na via de dados do computador). Como cada endereço diferente é capaz de atingir áreas separadas de armazenamento de oito *bits*,

com endereços binários de 14 *bits*, a ROM é capaz de memorizar 16484 (16 k) *bytes* diferentes de dados. Estes existem na realidade e formam-se através de ligações simples na altura do fabrico da própria ROM.

Imaginemos que o endereço é descodificado por circuitos lógicos de tal forma que haja uma saída, e só uma, de uma porta lógica, em estado alto. Se esse endereço tivesse sido designado para armazenar o número 255, ou 11111111 binário, então a saída da porta estaria ligada a todos os oito pinos de dados da ROM. Se os dados armazenados representassem o número 129, que é 10000001 em binário, a voltagem alta à saída da porta só estaria presente nos pinos de dados 0 e 7. As linhas de dados estão normalmente baixas, mas as portas lógicas podem colocá-las em estado alto, de modo que os pinos reflectem sempre os dados armazenados num endereço que é indicado pela informação contida na via de endereços. Quando uma porta lógica não for seleccionada por um endereço apropriado, a sua saída ficará flutuante, de maneira que não exercerá influência na via de dados. Quando o pino CS não está activado por um nível lógico de voltagem zero, toda a saída do *chip* será flutuante.

Poderemos, pois, imaginar que, se todos os endereços contivessem o valor 255, seria necessário efectuar mais de 130 000 ligações. Os fabricantes conseguem na realidade alcançar este total, através de um processo fotográfico que imprime as ligações na sílica do próprio *chip*. O original para esta operação é extraordinariamente caro, mas depois de pronto torna a produção de cada *chip* muito barata; é por isso que as ROMs de origem fotográfica (Mask Etched Read Only Memories) se tornam economicamente rentáveis para produção de *chips* em larga escala.

Vale a pena mencionar aqui dois outros tipos de ROM. A «Programmable Read Only Memory» (memória de leitura exclusiva programável), ou PROM, é um dispositivo que sai da linha de produção completamente em branco. Virá a ser programado por uma máquina que gera voltagens altas destinadas a «queimar» determinados circuitos de ligação, de uma maneira parecida

com o comportamento de um fusível submetido a uma voltagem demasiado alta. Um modelo aperfeiçoado, a «Erasable Programmable Read Only Memory» (memória de leitura exclusiva reprogramável), ou EPROM, armazena os dados sob a forma de cargas eléctricas. Não só podemos programá-lo como também «apagar» o seu conteúdo, pela aplicação da luz ultravioleta sobre uma pequena janela situada na face superior.

Os dois tipos de memória acima descritos são principalmente usados na produção de protótipos e em linhas de fabrico de pequenas dimensões, e tem frequentemente «pinos compatíveis» com as ROMs normais, significando este facto que entrarão nas mesmas bases de ligação, ficando os pinos nos mesmos lugares. Os primeiros protótipos do *Spectrum* estavam equipados com EPROMs, de modo que podiam ser exaustivamente testados antes de o programa contido nas EPROMs passar para o dispendioso processo de manufactura das ROMs definitivas, que as substituíam nos modelos de série.

PORQUE HÁ TIPOS DIFERENTES DE MEMÓRIA

A característica mais importante das ROMs é que elas retêm os dados sem interrupção, quer estejam ou não alimentadas por corrente, isto porque, se bem que necessitem de determinada voltagem para fazer funcionar os circuitos de descodificação que contêm, podem manter os dados sob uma forma «não volátil».

Esta característica é essencial para qualquer computador, pois o facto de o ligarmos e desligarmos repetidas vezes não impede que, no primeiro caso, passe imediatamente a dispor de um conjunto de instruções prontas a actuar. As memórias que só podem conter dados pré-programados de nada servem se quisermos empregar memórias externas que o computador manipule. Mesmo o próprio sistema operativo da máquina, que comanda o teclado e o *écran* e que interpreta os programas que lhe introduzimos, precisa de mais registos que aqueles que o *Z80* pode fornecer; este último tem de reter, por exemplo, a informação de quando imprimiu qualquer coisa no tubo visor. É por isso que os computadores são dotados de memórias de acesso aleatório, nas

quais inserimos dados para futura utilização. Esta designação deriva do facto de que temos acesso a essas memórias quer para leitura quer para escrita de dados, e por uma forma à nossa escolha (a expressão «acesso aleatório» também é aplicável a vários tipos de ROMs, já aqui mencionadas, mas é uso aplicá-la só a memórias que podem ser escritas).

Podemos considerar o tipo mais simples de RAM como um dispositivo do tipo ROM, com linhas de endereço e circuitos de descodificação, mas em que, em lugar das ligações, cada pedaço de informação é arranjado com uma disposição do tipo *flip-flop*, em que a saída representa cada um dos *bits* de dados. Esta saída é encaminhada para a linha de dados apropriada quando introduzimos o endereço correcto na via de endereço. Por outro lado, a via de dados é bidireccional, de modo que, com a ajuda de dois pinos que comunicam ao *chip* se deve receber ou transmitir informação, a UCP pode enviar dados endereçados até à memória, ao mesmo tempo que estabelece os valores apropriados para os *flip-flops* de acordo com os dados enviados.

Este tipo de memória chama-se «RAM estática», porque, enquanto receber uma determinada voltagem, manterá sem alteração o padrão dos *bits* que o seu circuito interno está a armazenar. Porém, esta espécie de memória é de produção cara, e é difícil de ser fabricada em forma compacta, de modo que não constituirá surpresa para o leitor ficar a saber que o *Spectrum* está equipado com uma espécie diferente, a que se chama «memória dinâmica».

Esta armazena dados sob a forma de uma carga eléctrica, que, infelizmente, começa a enfraquecer ao fim de um curto espaço de tempo, de modo que a memória necessita da oportunidade de «refrescar» essas cargas. Fá-lo, ao receber um pedido de leitura de cada um dos *bits* antes que os seus valores se percam.

Como atrás realçámos, o *Z80* foi equipado de modo a gerar sinais de refrescamento durante os dois ciclos do relógio que ocorrem imediatamente a seguir ao cumprimento de uma instrução de obtenção de dados da memória. O registo R é utilizado para fornecer endereços sequenciais aos circuitos de memória,

que os vai usar em conjugação com o pino de refrescamento. Do ponto de vista do utilizador, não há diferença entre o funcionamento das memórias de acesso aleatório estáticas ou dinâmicas.

A preocupação principal neste capítulo tem sido a de estabelecer uma diferenciação entre a RAM e a ROM. Contudo, os mais cépticos poderão verificar essas diferenças seguindo o procedimento que vai ser descrito.

Determine-se qual o conteúdo de um endereço da ROM, por exemplo o da localização zero, escrevendo a instrução PRINT PEEK 0. Tente-se a seguir alterar o valor do conteúdo para, por exemplo, 1, escrevendo POKE 0,1. Obrigando o computador a executar novamente um PRINT PEEK 0, descobrir-se-á que estes esforços foram vãos. Repitam-se estas operações para o primeiro endereço da RAM, que é o 16 384, isto é, fazemos: PRINT PEEK 16 384, depois POKE 16 384,1 e por fim PRINT PEEK 16 384.

A prova ficará imediatamente à vista. No *Spectrum* de 16 k, os endereços superiores a 32 678 não reagirão, pois não existem localizações de memória acima dessa área.

CAPÍTULO 6

Linguagem para máquinas

De nada servirá sentarmo-nos em frente do computador e escrever «Toca uma música alegre»: ele pura e simplesmente não perceberá o que estamos a dizer. Podemos escrever um programa para tocar qualquer música, mas em primeiro lugar teremos de aprender a fazer com que o computador o compreenda. Para cobrir o espaço que separa a linguagem humana da do código máquina, desenvolveram-se muitas linguagens intermediárias e entendidas simultaneamente por nós e pelo computador. O BASIC, a linguagem que o *Spectrum* compreende, é a abreviatura da expressão «Beginner's All-purpose Symbolic Instruction Code» (código de instruções simbólicas de uso geral para iniciados), e é mais fácil de aprender que qualquer outra linguagem para computadores. Se bem que os «profissionais» tenham por vezes uma atitude um pouco depreciativa em relação ao BASIC, este, muito flexível, é universalmente considerado como a linguagem mais popular. Poderemos dedicar-nos a outras linguagens, de uso mais elegante e que correrão mais depressa no computador, mas, excepto por motivos masoquistas, é preferível por agora continuar a utilizar o BASIC.

É até natural que o leitor já esteja familiarizado com esta linguagem, pois não é o propósito deste livro a exploração dos pormenores mais convenientes para a escrita de programas. Quem ainda precisar de estudar o seu próprio BASIC, encontrará no manual do *Spectrum* a melhor ajuda. Além disso, as livrarias tem a pratelaeiras cheias de obras sobre técnicas de programação: e, antes de prosseguirmos, teremos de saber muito bem como programar em BASIC.

A INTERPRETAÇÃO DO BASIC

Como é que o BASIC trabalha? O *Z80* só compreende aquilo que os *bytes* de dados que lhe são introduzidos a partir da memória lhe determinam que faça, mas não reconhece o significado de

instruções tais como PRINT AT 2,5; «OLÁ». Imagine-se que escrevemos dois programas que fazem o altifalante reproduzir marchas e acordes alegres; organizando-os como sub-rotinas, poderemos escrever, por exemplo GOSUB 100 PARA «Parabéns a você». Se, no início do programa, tivermos escrito qualquer coisa como:

```
INPUT A$: IF A$="Toca-me um acorde alegre" THEN
GOSUB 100: STOP
```

ao fazer correr o programa, e se escrevermos «Toca-me um acorde alegre», o computador conseguirá finalmente compreender-nos. Adicionando outras linhas, por exemplo «Toca-me uma marcha», poderemos mesmo confundir um leigo, levando-o a pensar que o computador tem gosto musical! Porém, a máquina está a interpretar as instruções, não através de um processo de raciocínio, mas simplesmente através de comparações das ordens com uma lista que ele já possui, e só obedecerá caso essas ordens correspondam às que ele está preparado para reconhecer. O programa de código máquina próprio do *Spectrum* contém um «interpretador de BASIC», que opera acompanhando as linhas de instruções. Durante a execução, ele utiliza os valores das «palavras-chave» que vai encontrando para descobrir onde se situam, dentro da ROM, as ordens necessárias ao cumprimento da tarefa, e então coloca essas sub-rotinas num processo de GOSUB. Note-se que em código máquina, em lugar da instrução GOSUB, se usa a palavra CALL. Antes de nos debruçarmos sobre matéria tão delicada como o código de máquina, terá utilidade explicarmos o que sucede quando ligamos o *Spectrum* e escrevemos:

```
PRINT AT 2,5; "OLÁ"
```

e premimos a tecla ENTER.

Quando o ligamos, o *Z80* executa um *reset* e a seguir cumpre os códigos *op* armazenados, começando no endereço zero. Estes códigos são muito próprios: preparam as áreas de memória,

organizam as «variáveis de sistema» e, de forma genérica, põem a casa em ordem. Depois de completar esta rotina de «inicialização do sistema», o *Z80* salta para o sistema fechado (em ciclo) de operação principal, onde aguarda que se prima uma tecla. Quando tal sucede, ele reconhece o facto como um número de linha ou como uma palavra-chave — premindo-se a letra «P» o sistema operativo vai à procura da forma de traduzir a ordem PRINT, e coloca esta palavra no *écran*. Armazena também o «símbolo» relevante numa área de memória posta de lado para esse fim e a que costuma chamar-se o *edit buffer:* cada palavra em BASIC tem o seu próprio valor simbólico, e o que corresponde a PRINT é o décimal 245. O sistema operativo segue depois procedimento idêntico para a instrução AT.

Temos a seguir o «2,5»; esta ordem envolve mais trabalho, e os números em questão são armazenados como seis *bytes* de dados (na realidade, isto não é necessário para 2 e 5, números pequenos, mas é importante quando se trata de números maiores). A pontuação e o conjunto de palavras entre as aspas são colocadas no *buffer* sob a forma dos seus códigos ASCII (que poderemos verificar no manual de instruções do *Spectrum*).

Quando finalmente premimos a tecla ENTER, o sistema operativo coloca um código de ENTER no fim do *buffer* e verifica a «sintaxe» de tudo o que escrevemos. Se não houver erros, isto é, se não houver qualquer combinação ilícita de códigos na linha, o processador passa à tarefa seguinte; caso contrário, coloca um ponto de interrogação sobre o local em que houve violação das ordens e não prossegue enquanto não efectuarmos a correcção.

Se tudo estiver bem, efectua-se uma verificação: o sistema operativo vê se os dados do *buffer* começam por um número de linha. Nesse caso, o conjunto dos dados é transferido para uma área de memória preparada para armazenar programas em BASIC, de modo a serem executados mais tarde. Como a linha que escrevemos é um comando directo, o sistema operativo passa o controle para o interpretador de BASIC, que assim analisa o primeiro valor — 245. Reconhece-o como uma ordem para

passar o controle para a rotina de saída existente no endereço 10h. Esta rotina trata então os dados recebidos, colocando-os no *écran* e cumprindo a ordem «AT 2,5;», através da análise dos respectivos símbolos, o que leva ao ajuste das coordenadas de impressão. Quando se alcança o símbolo ENTER, a rotina de saída volta a passar o comando para o interpretador. Este, verificando que a tarefa está completa, imprime «OK 0.1», e leva o *Z80* a regressar ao ciclo de operação principal. Este exemplo é muito simples, mas demonstra bem a importância da «programação de sistema» *(system software)*: sem o sistêma operativo (SO) e sem o interpretador de linguagem BASIC, o microcomputador não teria qualquer utilidade.

CÓDIGO MÁQUINA

Tratemos agora do enigmático código máquina. Pode dar-se o caso de alguns dos leitores estarem já familiarizados com a linguagem do *Z80*, e nesse caso reconhecerão o prazer que dá o facto de se conseguir escrever um programa que trabalhe sem problemas. As complicações maiores são as provocadas pelo elevado número de instruções necessárias para se conseguir realizar alguma coisa com significado e pela natureza aparentemente abstracta do código máquina. Contudo, há mnemónicas que ajudam a relembrar as funções dos vários códigos. Reparando na listagem do capítulo 4, vemos que o decimal 62 significa «carregue A com o número seguinte», e que se transforma em LD A,N. Estas mnemónicas são conhecidas pela designação genérica de «linguagem assembler», pelo facto de existirem programas disponíveis que traduzem as palavras nos códigos correctos e que portanto conseguem «assemblar» automaticamente um programa em código máquina.

Para auxiliar os que pretendam utilizar de futuro o código máquina, inclui-se a seguir uma descrição das mnemónicas da linguagem assembler. Não vale a pena tentar aprendê-las todas de uma só vez; aprendamos primeiro as mais vulgares, como LD, JR, ADD, SUB e CALL; o estudo dos códigos mais obscuros é mais fácil quando os encontramos no contexto de um programa.

GLOSSÁRIO DE MNEMÓNICAS DE CÓDIGO MÁQUINA

ADC Esta instrução tanto se aplica ao registo A como a HL. Quando aplicada ao A, significa «adicionar o conteúdo do *byte* simples especificado e o conteúdo da *carry flag* a A e deixar o resultado em A». Também se podem adicionar os *bytes* mantidos nos registos de *bytes* simples GP (*general purpose*, «de utilização genérica»), apontados para a memória pelos registos HL (ou registos de índice) ou os que são imediatamente armazenados como dados no programa. Quando aplicada ao registo HL, os conteúdos dos pares de registo (BC, DE, HL) ou os do ponteiro de pilha são adicionados ao HL, junto com uma unidade (1) se o *carry* estiver estabelecido, e com o resultado mantido em HL. Por exemplo, ADC A,D adiciona D a A, junto com o *bit* de *carry*.

ADD Semelhante a ADC, mas não inclui o *bit* de *carry*. Usa-se em outros registos de índice, além dos A e HL.

AND É uma operação lógica que pode ser executada no registo A. Retira um *byte* da memória (apontado pelo HL ou pelos registos de índice), da memória do programa, ou de um único registo GP, e executa um teste lógico AND a cada *bit* contido em A e no *bit* correspondente do *byte* especificado, deixando o resultado em A. Por exemplo, AND B faria uma comparação lógica do conteúdo de B com A. Se A contivesse 01010101 e B contivesse 00001111 (binários), o resultado colocado em A seria 00000101.

BIT Aplicando esta instrução testa-se qualquer *bit* dos registos GP e da memória se forem apontados pelos registos HL ou de índice, colocando a bandeira *(flag)* zero de acordo com eles. Por exemplo, BIT 3, (HL) testará o *bit* 3 do *byte* mantido no endereço HL, e se for zero a bandeira Z será estabelecida; se for um, Z será restabelecida. Os *bits* são sempre numerados de 0 a 7. Este código tem pelo menos dois *bytes* de comprimento, começando o primeiro sempre em EDH.

CALL Equivalente a GOSUB em BASIC, a instrução CALL leva o conteúdo do registo PC para a pilha e depois carrega o PC com os dois *bytes* seguintes da memória do programa, fazendo com que o *Z80* trabalhe a partir desse endereço. Veja-se RET, para descobrir como regressar desta instrução! CALL pode ser condicional — por exemplo, CAL Z só acontece se a bandeira zero estiver estabelecida.

CCF Significa «bandeira de *carry* complementar». O valor da bandeira de *carry* é invertido, isto é, zero transforma-se em um e vice-versa.

CP Significa «comparação». *Bytes* que sejam apontados na memória, os da próxima localização do programa, ou os registos GP, são subtraídos ao valor do acumulador e as bandeiras são estabelecidas de acordo com a operação, mas restabelece-se o valor original de A. Se A contiver 20h, então CP 20 estabelecerá a bandeira zero, mas o valor de A será mantido em 20.

CPD Trata-se de uma instrução complexa. Significa «Compare e decremente» e é útil para procurar num bloco de memória um *byte* que tem de ser comparado com o do registo A. CPD compara o *byte* apontado por HL com A e estabelece as bandeiras zero e de sinalização de forma correspondente (sinal=1 se o *bit* 7 do resultado for alto). É então que a instrução decrementa HL e BC, que actua como um contador, e se alcançar zero a bandeira P/V (neste caso, de *overflow*) é estabelecida.

CPDR «Comparar e decrementar com repetição». Esta instrução continuará a executar CPD até que o registo BC atinja

zero ou até que a operação CP estabeleça a bandeira Z.

CPI «Comparar e incrementar». É semelhante à instrução CPD, à excepção de que se soma 1 ao registo HL em vez de se subtrair.

CPIR «Comparar e incrementar com repetição». Tal como em CPDR, excepto que incrementa HL. Estas instruções de procura de blocos são muito úteis para encontrar dados de determinado valor. Terminadas as instruções, o registo HL manterá o endereço da primeira comparação ou o fim do bloco se nenhum dos valores indicados tiver sido encontrado.

CPL Significa «complemento», e só se aplica ao registo A. Todos os *bits* são invertidos, de modo que 00110101 tornar-se-à em 11001010.

DAA Esta instrução provavelmente só se empregará com o método de armazenamento de números designado por «decimal codificado em binário». Um registo de oito *bits* é dividido em dois grupos de quatro *bits;* cada grupo só poderá existir dentro dos limites de 0 a 9. O «acumulador decimal de ajuste» reajustará automaticamente os valores do registo A depois de uma adição ou subtracção, cumprindo as regras de codificação binária do decimal.

DEC «Decremento»: subtrai um ao valor ao qual é aplicado. A instrução DEC aplica-se a qualquer registo de oito ou de dezasseis *bits,* ou a um *byte* da memória apontado pelos registos HL ou de índice. Um aspecto importante, que se não se levar em conta conduzirá a resultados frustrantes nas primeiras tentativas de programação: o decremento de registos de dezasseis *bits* (HL, DE, BC) não terá qualquer efeito sobre o registo de bandeiras.

DI *Disable Interrupt;* levará a UCP a ignorar quaisquer

«interrupções disfarçadas» recebidas através do pino INT. Tratamos das interrupções noutra parte deste livro.

DJNZ Esta instrução, de grande utilidade, significa *Decrement and Jump if Not Zero* (decrementar e saltar se não for zero). Permite usar o registo B como um contador. Reduz B de uma unidade e, se (IF) o resultado não for igual a zero, executa um «salto relativo» (ver JR), ditado pelo *byte* seguinte da memória do programa. Se se atingiu o valor zero, a operação segue para a próxima instrução.

EI *Enable interrupt:* cancela a instrução DI.

EX *Exchange:* permite trocar dois determinados registos. Executa EX DE, HL, o que troca os respectivos conteúdos; faz igualmente EX aos dois *bytes* apontados pelo SP com os contidos em HL ou nos registos de índice; ou executa EX AF, AF', o que passará o comando para o par alternativo AF.

EXX Esta instrução chama para funcionamento os registos alternativos BC', DE' e HL', depois do que todas as operações se dirigem para estes. Para voltar ao modo original, basta aplicar novamente a ordem EXX.

HALT Executa precisamente o que o seu nome indica (alto): a UCP passará a executar NOP durante todo o tempo em que contiver uma instrução HALT, e só cessará ao ser-lhe aplicado um sinal de interrupção.

IM0 Veja-se o capítulo dedicado ao teclado para uma explicação completa sobre as interrupções. A IM0 coloca o *Z80* num modo de interrupção não utilizado pelo *Spectrum*, e no qual a UCP espera uma instrução (provavelmente um RST) que será colocada na via de dados pelo dispositivo que provocou essa interrupção.

IM1 Modo correcto para o *Spectrum*. A instrução IM1 provoca sempre a execução da ordem RST 38h desde que se receba uma interrupção.

IM2 Modo de interrupção que reaje a qualquer interrupção chamando uma sub-rotina cujo endereço foi previamente armazenado na memória. A localização procurada para se descobrir este endereço é determinada como se segue: o *byte* de ordem mais alta é considerado como o conteúdo do registo I, e o *byte* de ordem baixa é retirado da via de dados.
Nota: todas as instruções de modos de interrupção acima descritas alteram o processo de se lidar com uma interrupção disfarçada. As interrupções não disfarçadas provocam sempre uma operação do tipo RST 66h e, como o seu nome indica, não são afectadas pela instrução DI.

IN Esta interrupção lê um *byte* dos circuitos externos que têm acesso por portos. Fá-lo colocando o endereço de porto nas oito linhas de endereço mais baixas, activando IOQR e RD e depois colocando o *byte* recolhido da via de dados no registo especificado. A IN pode assumir duas formas: IN A,(porto), que tem o número do porto contido no *byte* seguinte da memória do programa e em que A mantém o resultado; e IN reg(C), na qual o *byte* contido em C é utilizado como endereço do porto e em que *reg* pode ser qualquer dos registos GP.

INC «Incremento»: adiciona um ao valor contido num determinado registo; aplicam-se as mesmas regras que para a instrução DEC.

IND
INDR
INI
INIR Grupo de instruções de entrada que servem para as aplicações específicas de IN, tal como o grupo CPD faz para as comparações e como o grupo LDD executa para as operações de carregamento. IND transfere o conteúdo

do porto especificado pelo registo C para a localização de memória cujo endereço é apontado pelo par de registos HL. Este é então decrementado, o mesmo sucedendo ao registo B. Se este último chegar a zero, estabelece-se a bandeira Z. INDR é a versão de repetição de IND e continua a transferir dados do porto (C) para a área de memória através da qual o ponteiro de HL está a ser decrementado, fazendo-o até que B atinja zero. As instruções INI e INIR funcionam como as IND e INDR, respectivamente, com a diferença de que HL é incrementado em lugar de ser decrementado.

JP *Jump* (salto); carrega o contador do programa com um novo valor, do que resulta uma operação de deslocamento do cumprimento do programa para outro passo qualquer. Existem três alternativas: saltar para o endereço indicado pelos dois *bytes* seguintes do programa; ou executar um JP condicional, que só sucederá se um teste de bandeira se revelar verdadeiro (isto é, JP NC, 6000h carregará o PC com 6000h, mas só se a bandeira de *carry* não estiver estabelecida); ou, finalmente, executar um salto indexado, que carrega o PC com o valor contido em HL, IX ou IY.

JR *Jump relative* (salto relativo). Modifica o conteúdo do registo PC por adição ou subtracção de um valor de sete *bits* ao *byte* mais baixo do PC. O valor requerido é armazenado no *byte* seguinte da memória do programa. Se o *byte* contiver menos de 127, é adicionado ao valor do PC, mas só depois do incremento automático que se segue à operação de recolha. Deste modo, JR 0 não terá qualquer efeito, enquanto JR 1 fará a UCP omitir um *byte* antes de ir buscar o seu seguinte código *op*.

Se o *bit* 7 do«*byte* de deslocamento» for 1 (por outras palavras, se o número for maior que 127), este valor é tratado por uma convenção conhecida com o «comple-

mento de 2», que permite gerar números negativos. Por exemplo, FEh (254 dec) tem um complemento de 2 que é −2. Se complementarmos (veja-se CPL) o *byte* de deslocamento e adicionarmos um, chegaremos a um valor que deverá ser subtraído ao PC. Deste modo, JR FEh (254 dec) obriga a UCP a dar um salto de regresso de duas localizações, onde irá encontrar a mesma instrução, criando-se assim um ciclo sem fim.

JR serve também como instrução condicional, mas testa apenas um número limitado de bandeiras, JR Z, JR NZ, JR C e JR NC. Contudo, JR é muitas vezes preferível a JP porque, para além de ser um *byte* mais curto, o seu uso evita que o programador escreva código que é obrigado a residir numa área particular da memória. JP necessita de ser acompanhada pelo endereço do momento, enquanto que a instrução JR só modifica o valor existente no PC.

LD Este é o mais comum dos códigos de operação: *load* (carregar). À primeira vista, parece que podemos carregar tudo com qualquer coisa, mas na realidade existem certas limitações! Na sua forma mais simples, LD A,B pegará no valor armazenado em B e colocá-lo-á em A, apagando o *byte* previamente existente em A. Percorramos agora sistematicamente todos os códigos de carregamento disponíveis.

Podemos carregar qualquer registo GP de oito *bits* com dados «imediatos»: isto é, dados armazenados no endereço seguinte da memória do programa.

Podemos carregar com dados imediatos todos os registos de 16 *bits* (à excepção de PC), desde que eles estejam armazenados nos dois *bytes* seguintes do programa. Há adiante uma nota sobre o modo como o *Z80* trata valores de dois *bytes*.

A memória externa, que mantém os seus endereços em HL ou nos registos de índice (IX e IY mais um

deslocamento, que explicaremos mais tarde), pode também ser carregada com dados imediatos, isto é, LD (HL), FFh.

É possível carregar o conteúdo de qualquer registo GP de oito *bits* com o conteúdo de um outro registo.

Podemos carregar os dois registos especiais de oito *bits*, I e R, com o valor contido em A, e este pode ser carregado com os valores dos dois primeiros.

Só se podem transferir valores de 16 *bits* dos registos duplos para o SP (contudo, observe-se em EX). Para transferir, por exemplo, o conteúdo de BC para DE, aplica-se LD D,B e depois LD E,C.

Podemos carregar em qualquer registo GP de oito *bits* os dados indicados pelo endereço contido em HL ou nos registos de índice. Podemos carregar na memória externa o conteúdo de qualquer registo GP de oito *bits* desde que o endereço de localização seja mantido em HL ou nos registos de índice.

O registo A pode também usar BC e DE como ponteiros de memória: é assim que LD A,(BC) carregará o conteúdo do endereço mantido em BC no acumulador, e LD (DE), A carregará o endereço contido em DE com o valor de A.

As instruções LD A,(ADDR) e LD (ADDR),A demonstram a flexibilidade extra de A. Aqui ADDR é um valor de dois *bytes* armazenado como dado imediato nas duas localizações de programa logo a seguir. Este valor é adquirido e usado como ponteiro de memória, de modo que LD A,(0000h) carregará A com os *bytes* de dados armazenados na localização zero.

Por fim, esta técnica de usar dados imediatos como ponteiros de memória pode ser aplicada aos registos de 16 *bits*. Por exemplo, LD (ADDR), BC colocará o conteúdo de BC nas duas localizações indicadas por ADDR e ADDR+1.

A lista de LDs é extensa mas não infinita! Por

exemplo, não se pode executar LD (ADDR), outro ADDR. Esta operação requer qualquer coisa como LD HL, outro ADRR seguida de LD (ADDR), HL. Repare-se também que nenhuma instrução *load* altera as bandeiras.

LDD Trata-se de uma instrução complexa semelhante a CPD. Leva a UCP a transferir os dados contidos no endereço armazenado em HL para o endereço guardado em DE. Decrementa então os registos BC, DE e HL. Se BC se tornar zero, estabelece-se a bandeira P/V.

LDDR Semelhante a LDD, mas com repetição automática se BC não for igual a zero. Este código *op* transfere um bloco de dados de uma área da memória para outra, carregando da forma seguinte: HL com o último endereço do bloco que vamos mover; DE com o último endereço do destino que vamos preencher; BC com o número de *bytes* que vamos transferir. LDDR cumpre a movimentação pretendida e os dados passam a estar em ambas as áreas de memória escolhidas.

LDI Semelhante a LDD mas com um incremento em vez de um decremento nos registos HL e DE.

LDIR Versão auto-repetitiva de LDI. Escolhendo LDDR ou LDIR podemos encher a nova área, respectivamente, «do topo para baixo» ou «do fundo para cima», o que é importante se os blocos se sobrepõem.

NEG Código de operação exclusivamente de acumulação. Transforma o valor de A no valor negativo em «complemento de dois» do conteúdo original, por complementação e adição de uma unidade. Desde que não esteja a lidar com a aritmética do complemento de dois, um programa pode usar esta instrução como forma expedita de subtrair um valor a zero.

NOP *No operation:* quando a UCP adquire este código, apenas incrementa o PC, como sempre. Instrução surpreendentemente útil, em especial durante a concepção de um programa.

OR Código de operação lógico, que pode pegar num *byte* de dados imediatos, no conteúdo de um registo GP ou num valor de memória apontado por HL, IX ou IY, e realizar um teste *bit* a *bit* do tipo ORing entre o *byte* e o registo A. O resultado é deixado em A, estabelecendo-se apropriadamente as bandeiras. Por exemplo, se A contém o binário 00001111, OR83h (10000011 binário) terá como resultado, em A, 10001111.

OUT Versão *write* (escrever) de IN. Aplicam-se as mesmas regras, excepto que os dados são enviados do registo para o porto especificado.

OUTD Grupo de instruções que é o complemento do grupo IND.
OUTI Faz-se sair para o porto (C) o conteúdo da localização de
OTDR memória apontado por HL. HL é então decrementado ou
OTIR incrementado; B é decrementado; e, *se* a instrução for do tipo repetitivo, mantém-se a acção até que B alcance o valor zero.

POP Código de operação associado com a pilha, que é a área de memória externa reservada pela UCP para seu uso próprio: guarda aí os endereços de retorno das instruções CALL. POP pega nos dois *bytes* previamente armazenados nessa área e carrega-os no par de registos especificado (AF, BC, DE, HL, IX, IY). O ponteiro da pilha é então incrementado duas vezes, ficando apontado para os dois *bytes* seguintes a serem submetidos a uma POP. Mais adiante trataremos das operações de pilha.

PUSH Esta instrução inverte a operação POP: decrementa o SP duas vezes, depositando o conteúdo do par de registos

especificado no endereço de memória criado durante o processo.

RES *Reset* para zero. Repõe a zero o valor de qualquer *bit* guardado nos registos GP ou numa localização de memória apontada quer por HL quer pelos registos de índice. RES 1,A fará com que o *bit* 1 do registo A passe a ser zero.

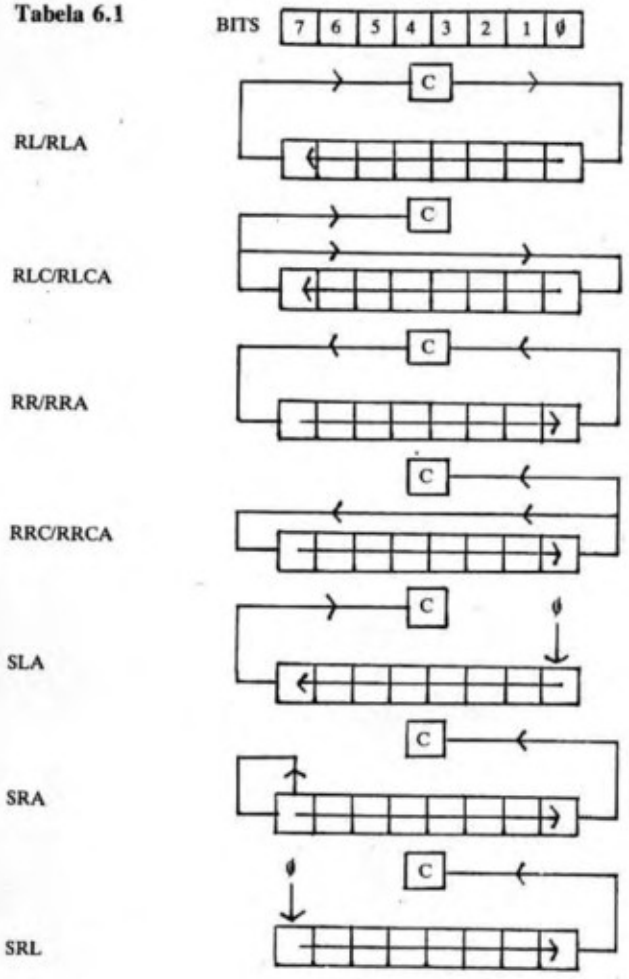
RET *Return* (regresso) de uma instrução CALL: é usada no final de uma sub-rotina de modo a regressar à rotina original. RET vai recuperar o endereço que tinha sido levado para a pilha pelo código *op* CALL, e coloca-o no PC, causando portanto um salto para essa localização. Cuidado com a intromissão na pilha ou no SP durante a sub-rotina: pode fazer com que o endereço correcto fique irrecuperável. Esta particularidade pode ser utilizada de propósito, por exemplo para determinar o valor do PC.

RETI Instrução RET especial, para uso no fim de uma sub-rotina chamada em resposta a uma interrupção disfarçada. Actua com uma RET normal; mas sinais enviados através da via de comando também avisam o dispositivo de interrupção de que esta foi «revisada».

RETN Regresso *(return)* de uma interrupção não disfarçada.

RL É o primeiro de um conjunto de códigos de «rotação e troca», que se descrevem melhor por meio de um esquema: a tabela 6.1 dá uma explicação mais completa. RL roda um *byte* de dados deixados na bandeira de *carry*. O *bit* mais alto, *bit* 7, é colocado nessa bandeira, e o *bit* 6 salta para ocupar o lugar deixado pelo primeiro. Todos os outros *bits* se deslocam um furo, e o *bit* 0 é carregado com o antigo valor do *bit* de transporte. Desde que não seja oportunamente especificado, todos os códigos de rotação e troca podem ser aplicados a qualquer registo

Tabela 6.1



GP ou a um *byte* da memória apontada por HL ou pelos registos de índice.

RLA Excepção à regra anterior. RLA só pode ser aplicada ao registo A. Comporta-se da mesma maneira que RL, exceptuando o facto de não afectar nenhuma bandeira além da de transporte *(carry)*. As rotações que acabem com A, todas de um *byte* de comprimento, são de mais rápida execução que as outras.

RLC «Rotação para a esquerda circular», semelhante a RL. Diferencia-se desta pelo facto de que o *bit* 7 é copiado para o *bit* 0 ao mesmo tempo que o *carry*, cujo valor inicial se perde.

RLCA Versão circular de RLA.

RR «Rotação para a direita». É uma instrução RL em sentido contrário!

RRA Versões de rotação para a direita de RLA, RLC e RLCA.
RRC
RRCA

RRD Estas instruções não estão na ordem alfabética porque
RLD não são códigos de rotação normais. RLD significa *rotate left decimal* (rotação para a esquerda decimal). Usa-se em conjunto com a aritmética de codificação decimal binária ou em certos casos em que se pretenda rodar um valor em quatro *bits*. O registo HL deve apontar para a localização de memória que se quer rodar; o grupo de quatro *bits* mais baixo do acumulador é colocado no grupo mais baixo dessa localização de memória, e o conteúdo desse grupo é colocado nos quatro *bits* mais altos, cujo conteúdo prévio é transferido para o grupo

mais baixo do registo A. RRD é uma instrução semelhante, excepto que roda na direcção contrária.

RST Não é uma instrução de rotação! Significa *restart* (recomeço), e compreende oito códigos de operação RST, cada um deles executando uma ordem CALL para um dos seguintes endereços específicos: 0, 8, 16, 24, 32, 40, 48 e 56 decimal. O código RST só tem um *byte* de comprimento, e é usado intensivamente na ROM para chamar rotinas de utilização frequente.

SBC *Subtract with carry*, subtracção com transporte. Semelhante a ADC, com a diferença de que neste caso se realiza uma subtracção.

SCF *Set the carry flag*, estabelecer a bandeira de transporte. Provoca a introdução do valor 1 na bandeira de transporte, qualquer que seja o seu valor anterior.

SET Ver RES. SET torna o *bit* especificado em 1.

SLA *Shift left arithmetic*. É o primeiro de três códigos de troca. Ver RL e a tabela 7.

SRA *Shift right arithmetic*, troca para a direita aritmética.

SRL *Shift right logic*, troca para a direita lógica.

SUB Código *op* exclusivo de oito *bits*, que subtrai ao registo A o conteúdo de um registo GP, ou da memória externa apontada pelo HL ou pelos registos de índice. Deixa o resultado em A e estabelece as bandeiras. Se pretendermos executar uma instrução SUB de 16 *bits*, podemos usar SBC desde que se restabeleça previamente a bandeira de transporte; a forma mais rápida de o conseguir é AND A.

XOR Esta instrução de tonalidade exótica significa «OU exclusivo» e é do mesmo tipo que as operações AND e OR. Tem o resultado lógico um se os dois *bits* testados forem diferentes entre si. Deste modo, dois zeros e dois uns darão sempre um resultado zero. Esta instrução aparece frequentemente na forma XOR A, porque esta é a maneira mais fácil de levar o acumulador a zero.

REGISTOS DE ÍNDICE, BANDEIRAS E PILHA: NOTAS A ACOMPANHAR O GLOSSÁRIO

Usamos muitas vezes os registos de índice, IX e IY, como ponteiros de memória. Os códigos de operação que os utilizam apresentam a forma «LD A,(IX+*byte* de deslocamento)», em que o «*byte* de deslocamento» é um número em complemento de dois (ver JR) adicionado ao registo de índice, antes que esse valor seja usado como ponteiro de memória para acesso a determinada localização da memória. Deste modo, se carregarmos um registo de índice com um número adequado no início do programa, podemos ter acesso rápido e simples a um bloco de memória constituído por 255 *bytes* e centrado num valor seleccionado.

O *software* de sistema do *Spectrum* usa o registo IX para ter acesso às variáveis de sistema, de modo que tem sempre de ser restabelecido no seu valor correcto antes que se verifique um *return* para o sistema operativo. Os valores dos códigos *op* podem ser calculados a partir do seu código (HL) equivalente, pois DDh é o prefixo de IX e FDh é o de IY. O *byte* de deslocamento é inserido depois do valor do código *op* (HL), portanto, para transformar OR (HL) em OR (IX+2), por exemplo, o código de um *byte* B6h tornar-se-ia no código de três *bytes* DD B6 02 h.

Uma palavra agora sobre bandeiras *(flags)*. A bandeira de transporte é útil para testar se um valor é maior do que outro — comparando A com 10 estabelece a *carry* em 1 se A contiver menos do que 10. A bandeira P/V tem várias funções : testa os *overflows* quando se executam operações aritméticas em complemento de dois; reflecte a «paridade» do resultado de uma operação lógica, com um número par de uns no acumulador estabelecendo a bandeira; e é usada pelos códigos *op* de bloco como bandeira zero suplementar. Lembremos também que é possível alterar acidentalmente os valores das bandeiras quando usamos as instruções POP AF e EX AF AF'.

Um dos aspectos confusos do código do *Z80* é o método de armazenar números de dois *bytes* na memória externa. O *Z80* coloca o *byte* baixo na primeira localização e o *byte* alto no endereço seguinte. No caso de LD HL,0100, este valor será armazenado na memória como 21,00,01; o segundo *byte* representa os oito *bits* daquele valor.

Chegamos por fim à pilha *(stack)*. Imaginemo-la como um monte de cartas; colocar um valor na pilha significa que ele irá para o topo do monte, de modo que se tivermos de retirar (ou POP) a carta do topo, esta seria sempre a última que colocámos ali, ou a última à vista se a carta anterior tivesse sido retirada. O ponteiro da pilha contém o endereço da próxima carta a submeter a um POP. No *Z80*, o ponteiro da pilha (SP) é decrementado antes de um PUSH, de modo que a pilha cresce sempre «do topo para baixo».

PROGRAMA MONITOR

Se quisermos escrever programas complexos, será bom investimento a compra de um programa assembler e um livro de referência. Para despertar o gosto, contudo, serão suficientes o glossário antes descrito e a lista de códigos de operação do capítulo «Conjunto de caracteres» do manual do *Spectrum*.

O programa 6.1, em BASIC, serve para carregar dados decimais ou hexadecimais na memória, para além de permitir verificar o que se fez, alterar, gravar e executar qualquer código máquina criado por nós. O programa possui as suas próprias instruções de operação; por isso é preciso o maior cuidado ao escrevê-lo, gravando-o logo a seguir. Depois de executado um programa, ele imprimirá o valor deixado no registo BC; portanto

pode carregar-se este com qualquer valor que quisermos examinar. É útil explorar o que fazem os vários códigos de operação, com especial incidência nos condicionais e nos de rotação.

Programa 6.1. Monitor

```
  1 REM PROGRAMA MONITOR
  2 REM
  5 REM O SIMBOLO DA LIBRA DEVE
  6 REM SER LIDO COMO CARDINAL
  7 REM (conforme a impressora)
  8 REM
 10 GO TO 9000
 11 REM
 12 REM  Formatacao em 1 byte
 13 REM
 20 IF NOT h THEN LET a$="  "+S
TR$ a: LET a$=a$(LEN a$-2 TO ):
RETURN
 30 LET a$="  "
 40 FOR x=LEN a$ TO 1 STEP -1:
LET b=a-INT (a/16)*16: LET a$(x)
=CHR$ (b+48+7*(b>9)): LET a=INT
(a/16): NEXT x: RETURN
 50 REM
 51 REM  Formatacao em 2 bytes
 52 REM
 60 IF NOT h THEN LET a$="    "
+STR$ a: LET a$=a$(LEN a$-4 TO )
: RETURN
 70 LET a$="    ": GO TO 40
100 REM
101 REM   Input n
102 REM
110 LET a$="Numero invalido;ten
te de novo"
120 REM
121 REM  Comece aqui
122 REM
130 LET n=0: INPUT (a$+" "); LI
NE b$
140 FOR x=1 TO LEN b$: LET b=CO
DE b$(x)-48: IF h THEN GO TO 170
150 IF b<0 OR b>9 THEN GO TO 11
0
160 LET n=INT n+b*10↑(LEN b$-x)
: NEXT x: RETURN
170 IF b<0 OR b>9 AND b<17 OR b
>22 THEN GO TO 110
180 LET b=b-7*(b>9): LET n=INT
n+b*16↑(LEN b$-x): NEXT x: RETUR
N
```



```
200 REM
201 REM   Obtencao de l
202 REM
210 GO SUB 120: IF n<0 OR n>655
35 THEN LET a$=y$: GO TO 210
220 RETURN
1000 REM
1001 REM   EXAMINE
1002 REM
1010 CLS : PRINT INVERSE 1;"M";
INVERSE 0;"enu,"; INVERSE 1;"N";
 INVERSE 0;"ova localizacao,"; I
NVERSE 1;"A"; INVERSE 0;"lterar,
"; INVERSE 1;"6&7"; INVERSE 0;"s
croll": LET a$=z$
1020 GO SUB 200: IF n>65516 THEN
 LET a$="Muito alto; reentre": G
O TO 1020
1030 LET c=0: LET l=n: PRINT AT
2,0;: FOR y=0 TO 19: LET p=y: GO
 SUB 1210: NEXT y: GO SUB 1190
1040 LET a$=INKEY$: IF a$="A" TH
EN GO TO 1100
1050 IF a$="6" THEN GO TO 1130
1060 IF a$="7" THEN GO TO 1160
1070 IF a$="M" THEN CLS : RETURN

1080 IF a$="N" THEN GO SUB 1200:
 LET a$=z$: GO TO 1020
1090 GO TO 1040
1100 LET a$="Novo valor"
1110 GO SUB 120: IF n>255 THEN L
ET a$=y$: GO TO 1110
1120 POKE l+c,n: LET p=c: GO SUB
 1210
1130 IF c<>19 THEN GO SUB 1200:
LET c=c+1: GO SUB 1190: GO TO 10
40
1140 IF l>65515 THEN GO TO 1040
1150 LET l=l+1: RANDOMIZE USR 23
335: GO SUB 1190: LET p=c: GO SU
B 1210: GO TO 1040
1160 IF c<>0 THEN GO SUB 1200: L
ET c=c-1: GO SUB 1190: GO TO 104
0
1170 IF NOT l THEN GO TO 1040
1180 LET l=l-1: RANDOMIZE USR 23
350: LET p=c: GO SUB 1210: GO TO
 1040
1190 PRINT AT c+2,0;"Endereco>";
AT c+2,22;"<Contem": RETURN
1200 PRINT AT c+2,0;TAB 9;AT c+2
```

```
,22,: RETURN
1210 LET a=l+p: GO SUB 50: PRINT
 AT p+2,9;a$;"     ";: LET a=PEE
K (l+p): GO SUB 20: PRINT a$: RE
TURN
2000 REM
2001 REM  ENTRADA DO CODIGO
2002 REM
2010 CLS : PRINT "O Modo 2 permi
te entrar o codigo";AT 4,2;"ENTR
E UM VALOR FORA DO LIMITE"'TAB 7
;"PARA VOLTAR AO MENU";AT 10,2;"
Endereco actual = "
2020 LET a$=z$: GO SUB 200: LET
l=n
2030 LET a=l: GO SUB 50: PRINT A
T 10,20;a$: LET a$="Codigo": GO
SUB 120: IF n<0 OR n>255 THEN RE
TURN
2040 POKE l,n: LET l=l+1*(l<6553
5): GO TO 2030
3000 REM
3001 REM  GRAVACAO DO CODIGO
3002 REM
3010 CLS : PRINT "O Modo 3 permi
te gravar o codigo"
3020 LET a$=z$: GO SUB 200: LET
l=n: LET a=n: GO SUB 50: PRINT A
T 6,4;z$;" = ";a$
3030 LET a$="Numero de bytes": G
O SUB 200: LET s=n: LET a=n: GO
SUB 50: PRINT AT 8,2;"Compriment
o do bloco = ";a$
3040 INPUT "Nome do registo ";a$
: SAVE a$CODE l,s
3050 PRINT AT 12,3;"REBOBINE A F
ITA e VERIFIQUE"'TAB 6;a$;'': VE
RIFY a$CODE l,s
3060 INPUT "O.K. --- ENTER para
o MENU";a$: RETURN
4000 REM
4001 REM  EXECUCAO DO CODIGO
4002 REM
4010 CLS : PRINT "O Modo 4 execu
ta o Cod.Maquina"
4020 LET a$=z$: GO SUB 200: LET
a=n: GO SUB 50: PRINT AT 8,3;"O
codigo sera executado"'"  a part
ir de";AT 8,12;a$;AT 12,3;"PRIMA
:-"'"     R para executar o Codig
o";AT 14,3;" N para alterar o en
dereco";AT 16,3;" M para o MENU"
```



```
4030 LET a$=INKEY$: IF a$="M" TH
EN RETURN
4040 IF a$="N" THEN GO TO 4010
4050 IF a$<>"R" THEN GO TO 4030
4060 CLS : PRINT USR n: INPUT "C
ODIGO executado; prima ENTER";a$
: RETURN
5000 REM
5001 REM  MUDANCA DE BASE
5002 REM
5010 CLS : PRINT AT 3,2;"O Progr
ama Monitor trabalha"'"     agora
 com numeros de base": IF h THEN
 GO TO 5030
5020 LET h=1: PRINT AT 8,12;"DEZ
ASSEIS": GO TO 5040
5030 LET h=0: PRINT AT 8,14;"DEZ
";
5040 INPUT "Prima ENTER para o M
enu";a$: RETURN
6000 REM
6001 REM  CONVERSAO DE NUMEROS
6002 REM
6010 LET t=h: CLS : PRINT AT 4,6
;"Para converter:-";AT 8,5;"Hex
em Decimal prima H";AT 10,5;"Dec
imal em Hex prima D";AT 15,7;"M
para o MENU"
6020 LET a$=INKEY$: IF a$="" THE
N GO TO 6020
6030 IF a$="M" THEN LET h=t: RET
URN
6040 IF a$="D" THEN GO TO 6070
6050 IF a$="H" THEN GO TO 6100
6060 GO TO 6020
6070 LET a$="Decimal": LET h=0
6080 GO SUB 200
6090 LET a=n: LET h=1: GO SUB 60
: PRINT #1;"Hex = ";a$: GO TO 60
20
6100 LET a$="Hex": LET h=1
6110 GO SUB 200
6120 PRINT #1;"Decimal = ";n: GO
 TO 6020
8000 REM
8001 REM      MENU
8002 REM
8010 CLS : PRINT AT 1,6;"MONITOR
 DE CODIGO MAQUINA": PLOT 46,158
: DRAW 205,0
8020 PRINT AT 3,10;"Prima o nume
ro para";AT 4,11;"a funcao desej
```

```
ada"
8030 RESTORE : FOR x=1 TO 6: REA
D a$: PRINT AT x+2+6,5;x;" ----
 ";a$: NEXT x
8040 LET a=CODE INKEY$ -48: IF a
<1 OR a>6 THEN GO TO 8040
8050 GO SUB a*1000: GO TO 8000
8060 DATA "Exame da Memoria","En
trada de Codigo","Gravacao de Co
digo","Execucao do Codigo","Muda
nca de Base","Conversao de Numer
o"
9000 REM
9001 REM  INICIALIZACAO
9002 REM
9010 INK 1: PAPER 6: BORDER 0: P
OKE 23658,8
9020 LET h=0: LET z$="Endereco i
nicial": LET y$="Fora dos limite
s;tente de novo"
9030 RESTORE 9200: FOR x=23296 T
O 23364: READ a: POKE x,a: NEXT
x
9100 GO TO 8000
9200 DATA 245,230,24,246,64,103,
241,245,230,7,15,15,15,198,9,111
,241,201
9210 DATA 205,0,91,6,8,197,1,13,
0,213,229,237,176,225,209,36,20,
193,16,241,201
9220 DATA 62,2,205,0,91,235,60,2
05,18,91,254,21,32,244,201
9230 DATA 62,21,205,0,91,235,61,
205,18,91,254,2,32,244,201
```

Agora, um desafio! Escrever, sem consultar a tabela 6.2, um programa que dispare uma seta através da linha superior do *écran*. Para tal, é preciso saber o seguinte:

1. RST 10h (código D7) chamará a rotina de impressão da ROM, e colocará o símbolo do código mantido no registo A no *écran*, na posição de impressão actual.

2. Pode-se alterar a posição actual de impressão (que é automaticamente actualizada pela rotina de impressão) enviando 16h (o código AT) de A para a rotina de impressão (RST 10h de novo). Neste caso deve-se fazê-lo seguir-se pelas novas coordenadas de

impressão; por exemplo, LD A,16h RST 10h, LD A,0, RST 10h, LD A,0, RST 10h colocará a posição de impressão no canto superior esquerdo.

3. O código para a seta poderá ser 3Eh (o símbolo de «maior que»).

4. Se quisermos ver a seta a viajar teremos de incorporar um ciclo de retardamento de 16 *bits!*

A «resposta» dada na tabela 6.2 não é a única hipótese de resolver este problema. Há obrigação de a melhorar quando dominarmos bem esta matéria. Será divertido!

Tabela 6.2

Endereço	Código hex	Assembler
7F80	012000	LD BC,0020h
7F83	110100	LD DE,0001h
7F86	6A	LD L,D
7F87	CDA97F	CALL 7FA9h
7F8A	3E3E	LD A,3Eh
7F8C	D7	RST 10h
7F8D	61	LD H,C
7F8E	10FE	DJNZ −2
7F90	25	DEC H
7F91	20FB	JR NZ,−5
7F93	6A	LD L.D
7F94	CDA97F	CALL 7FA9
7F97	3E20	LD A,20h
7F99	D7	RST 10h
7F9A	6B	LD L,E
7F9B	CDA97F	CALL 7FA9h
7F9E	3E3E	LD A,3Eh
7FA0	D7	RST 10h
7FA1	14	INC D
7FA2	1C	INC E

7FA3	7B	LD A,E
7FA4	FE20	CP 20h
7FA6	20E5	JR NZ,−27dec
7FA8	C9	RET
7FA9	3E16	LD A,16h
7FAB	D7	RST 10h
7FAC	AF	XOR A
7FAD	D7	RST 10h
7FAE	7D	LD A,L
7FAF	D7	RST 10h
7FB0	C9	RET

PARTE 2

O Spectrum da Sinclair

CAPÍTULO 7

Introdução do «Spectrum»

Na primeira parte deste livro, descrevemos os conceitos gerais dos microcomputadores, a partir de princípios básicos. É tempo agora de examinar minuciosamente o funcionamento do *Spectrum* da Sinclair. Existem dois modelos, mas a única diferença entre eles é que um tem 16 k de RAM e o outro 48 k. Esta capacidade de memória adicional permite a utilização de programas maiores, e um armazenamento «a bordo» de quantidades mais vastas de dados úteis.

O *Spectrum* foi um dos primeiros computadores a cores de preço acessível para o comprador de posses médias em toda a Europa. Grande parte da sua concepção assentou na dos seus predecessores, o *ZX80* e o *ZX81*, que também utilizam o mesmo microprocessador, o *Z80*.

O *ZX80* foi o primeiro computador verdadeiro posto à venda por menos de 100 libras; não conseguia produzir uma imagem de TV e computar ao mesmo tempo, tinha 4 k de ROM que englobava o sistema operativo e uma versão simples de BASIC e possuía só 1 k de RAM, uma ninharia comparando-a com os padrões actuais, e modesta já em 1980. Mesmo com estas limitações, foi um sucesso, pois interessava as pessoas que pretendiam um computador sem ser de «prateleira», para iniciados e a um preço correspondente.

O *ZX80* foi rapidamente substituído pelo *ZX81*, modelo muito melhorado e que se vendeu aos milhões, com 8 k de ROM e um modo de operação que mantinha a apresentação em *écran* à custa de um funcionamento muito lento. Estes melhoramentos foram acompanhados por uma redução de 20 libras no preço, e em breve o computador preto e silencioso, com o seu teclado branco de membrana, imaginado pela Sinclair, se vendia em números nunca imaginados, para além de contribuir para o aparecimento de novos mercados, não só de computadores como de *software*, de livros e de revistas.

AS POSSIBILIDADES DO «SPECTRUM»

O lançamento do *Spectrum* aconteceu em Maio de 1982. Verificou-se logo que representava um enorme avanço sobre o seu predecessor, e em termos de valor de aquisição estava muito à frente da concorrência — o modelo de 16 k era só 5 libras mais caro que um *ZX81* com a extensão de RAM. Vejamos agora algumas das notáveis possibilidades oferecidas pelo *Spectrum*.

1. Uma apresentação vídeo de alta resolução, que pode ser estabelecida em qualquer vulgar aparelho doméstico de televisão. Possui uma resolução de 256 por 192 pontos, e imprime 32 caracteres em cada uma das suas 24 linhas de texto. Cada espaço de carácter pode se apresentado em uma de duas cores diferentes, designadas por PAPER e INK, e para cada uma destas existem oito cores possíveis: preto, azul, vermelho, magenta, verde, *ciano*, amarelo e branco. Além disto, cada espaço admite ser posto em BRIGHT, que aumenta a luminosidade, ou em FLASH, que alterna os valores atribuídos a PAPER e a INK a um ritmo predeterminado. O *software* permite desenhar com a precisão de um ponto, ou *pixel*, nas 22 linhas superiores do *écran*, e oferece formas gráficas incorporadas e prontas a usar *(user-defined)*. Os dados do *écran* são mapeados em memória, o que significa que a imagem é copiada da RAM através da via de endereços, o que ocupa 6 k de memória que de outra maneira estariam à disposição do programador.

2. Um comando BEEP, que permite a criação de tons simples com o BASIC, e de efeitos sonoros mais complexos a partir de programas em código máquina. O som emerge de um pequeno altifalante instalado dentro da caixa. Se bem que normalmente passivo, a saída *ear* (monitor) pode alimentar um altifalante exterior.

3. Uma *interface* torna possível a utilização de um vulgar gravador de *cassetes* como meio de arquivo de dados. Programas em BASIC, dados e código máquina podem ser gravados em *cassettes* normais e carregados de volta ao computador através das ligações *ear* e *mic*, utilizando os fios fornecidos com o computador. A porta da *cassette* tem nos seus circuitos um

«disparador Schmitt», que melhora a sua prestação e permite que os dados sejam transferidos a uma velocidade de cerca de 1200 *baud* — o que é rápido em comparação com outras máquinas. A produção de dados seriados é tratada pelo *software* de sistema, que também aceita *verify* (verificação), um comando que nos certifica de que uma gravação foi feita em boas condições.

4. Dentro da ROM, o sistema operativo da máquina é acompanhado por um interpretador de BASIC. Desenvolveu-se aqui o BASIC aplicado no *ZX80* e no *ZX81*, conferindo-lhe listas *(arrays)* multidimensionadas, um avaliador de expressões completas, vírgula aritmética flutuante e verificação de sintaxe em cada entrada de linha. A divisão de séries é completada pela função TO — por exemplo, se A$ é «viva», então A$ (2 TO 4) será «iva». As linhas em BASIC são introduzidas através do teclado por um método de «tecla única»: cada tecla tem até cinco significados para o sistema operativo, dependendo do ponto da linha BASIC em que a tecla é premida, auxiliado pelo uso das teclas de *shift*. Como dialecto de BASIC, este método é excelente para os iniciados.

5. Vem preparado para operar uma impressora de baixo custo, de «portas mapeadas». A impressora é controlada por *software*, o que simplifica a necessária *interface*, e era já usada com o *ZX81*.

6. O teclado é formado por teclas móveis de borracha, separadas, como as das máquinas de escrever. Era característica parecerá pouco importante, mas os utilizadores do *ZX81* consideram-na um melhoramento valioso.

7. Vem preparado para operar um novo tipo de dispositivo de arquivo de dados de alta capacidade, designado por *microdrive*, que mais não é do que um sistema de leitura de fita magnética de alta qualidade, operando a elevada velocidade. Este meio aproxima-se das características das *floppy disk drives* e custa metade do preço. Recentemente, apareceram para o *Spectrum* verdadeiros sistemas de *floppy disk*, que muito melhoram as capacidades de trabalho deste computador.

Existem no mercado inúmeros periféricos para o *Spectrum*, e a própria Sínclair Research fabrica vários. A *Interface 1* é necessá-

ria para controlar os *microdrives* e tem incorporada uma *interface RS 232* e possibilidades de trabalho em rede *(network)*. A *Interface 2* recebe programas em *cartridges* de ROM, e aceita dois telecomandos *(joysticks)*.

As especificações acima descritas, associadas aos baixos preços praticados pela Sinclair, levaram esta firma britânica ao domínio do mercado dos microcomputadores, pelo menos na Grã-Bretanha. Dias antes do Natal de 1983, a milionésima máquina saiu das linhas de produção, e na altura não existiam sinais de perda de popularidade por parte deste pequeno mas potente computador.

A ESTRUTURA DO «HARDWARE»

A estrutura do *hardware* do *Spectrum* está representada na figura 7.1. Se compararmos esta com a disposição dos circuitos colocados dentro da caixa, julgaremos à primeira vista que não existe qualquer semelhança, mas na realidade, à parte alguns pequenos componentes que podem desfocar a imagem, reconhecemos as partes principais.

A mais evidente é o próprio microprocessador. Trata-se de *Z80 A* — uma versão do *Z80* capaz de funcionar a uma frequência de relógio mais alta que a do modelo normal. Uma larga via de dados de oito *bits* corre ao longo da placa dos circuitos, do microprocessador para os outros componentes principais, sendo ocasionalmente protegida de conflitos por resistências e díodos apropriados. A via de endereços de 16 *bits* é muito parecida.

A parte principal do sistema que ainda não referimos é praticamente constituída por um circuito integrado de 40 pinos ou terminais. É a ULA, *chip* de que se pode dizer que faz, com algumas excepções, tudo o que a UCP não é capaz de executar. É, na realidade, um «cavalo de trabalho» preparado na fábrica, que fornece os meios necessários à composição de uma imagem de vídeo — encarrega-se das formas mais exóticas de descodificação de endereços e também descodifica todos os endereços de portos. Fornece o sinal do relógio e muitas mais coisas.

As iniciais ULA significam *Uncommitted Logic Array*; trata-se

de um método de produção, a partir de uma substância estandardizada, de circuitos integrados preparados para desempenhar um papel específico. Os fabricantes de *chips* produzem um circuito que contém todos os «tijolos» de construção necessários a um dispositivo complexo — portas, *flip-flops*, inversores, *buffers*, etc. Deixa-se a tarefa final de os interligar através de um método de gravura fotográfica, sendo acertada com os desejos específicos do comprador. Este método torna muito mais barata a produção de um *chip* para desempenhar determinadas funções, sendo preferido ao da associação de componentes discretos, especialmente quando se requer um largo número destes.

Não existe nada de «mágico» dentro da ULA da Sinclair. As suas funções poderiam ser desempenhadas por um conjunto de *chips* lógicos, se bem que o espaço ocupado por estes exigisse uma enorme placa de circuitos; os factores tamanho e custo jogam em favor dos *chips* feitos «a pedido». À ULA está ligada tanto a via de dados como a de endereços, e também recebe a maior parte dos sinais de comando do *Z80*. Com toda esta informação, gera mais sinais específicos próprios, com os quais vai comandar todo o computador.

Muito próximo, na placa de circuitos, está um «cristal». É o componente externo do oscilador situado dentro da ULA: o oscilador gera o sinal de relógio para a UCP e para a própria ULA. Outras ligações para o «cavalo de batalha» incluem o *jack* de entrada *ear*, através do qual podemos introduzir dados seriados a partir de um gravador de *cassettes*. As saídas incluem os sinais de vídeo e de cor, a alimentação do terminal *mic* para gravação de dados e os impulsos que activam o altifalante e o teclado.

O componente seguinte mais evidente da placa de circuitos é a *chip* da ROM. É capaz de armazenar 128 k *bits*, isto é, 16 k *bytes*. É a residência do *software* do sistema, permanentemente gravado em ligações de sílica. Este *chip* está também ligado às vias de dados e de endereços; os sinais de funcionamento deste componente provêm da ULA.

A máquina de 48 k tem 16 *chips* de RAM, e a sua irmã mais

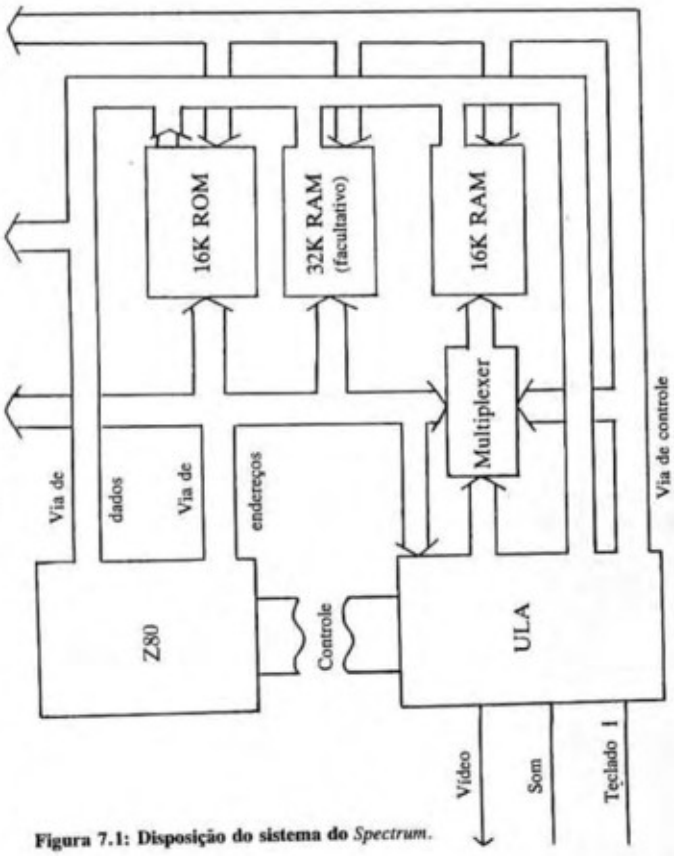


Figura 7.1: Disposição do sistema do *Spectrum*.

pequena somente 8. O modo como os *chips* a mais do modelo maior estão dispostos depende da idade da máquina. O complemento normal de 16 k *bytes* de RAM é constituído por oito *chips* dinâmicos RAM *4116* de 16 k *bits*, dispostos de modo que cada

chip contribui para uma linha da via de dados: deste modo, um *byte* de dados armazenado num determinado endereço terá cada um dos seus *bits* individuais guardado num *chip* RAM diferente.

Estes primeiros 16 k de RAM são tratados através das suas próprias vias de endereços, porque o *chip* da ULA precisa de aceder a essa área da memória independentemente da UCP, de modo a coligir dados que deverão ser enviados para o *écran*. Para facilitar esta tarefa, usam-se *chips* designados por *multiplexers (74 LS 157)*. Estes tem duas saídas para cada *bit* da via de endereços, uma para a via do *Z80* e outra que vem directamente da ULA. Um sinal de controle da ULA determina qual deles deverá ser passado para as *chips* de RAM.

Alimentar uma televisão doméstica requer um bom número de circuitos electrónicos. Para além da necessidade de se ter de sincronizar os impulsos de cor e de luminância com os sinais respectivos, estes devem ser codificados num sinal de vídeo composto, que também tem de ser «modulado». Este processo imita a maneira de funcionar de um emissor de TV e, por consequência, a área do circuito que efectua a «transmissão» está fechada numa caixa de metal isoladora, de modo que o computador não apareça incluído nas ondas radiofónicas. Uma versão idêntica mas mais fraca do que a dos sinais captados pela antena da sua TV passa então para o *écran* através do cabo e da entrada de recepção do aparelho. Como sabemos, as diferentes emissoras de televisão emitem em canais distintos; o *Spectrum*, tal como a maior parte dos computadores, transmite no canal 36.

A maioria dos componentes que ainda não vimos dedica-se quer a exercer uma função de tampão *(buffer)* entre os sinais provenientes das áreas principais do circuito, quer a fornecer as voltagens correctas sem as quais os circuitos integrados não poderiam funcionar.

A voltagem fornecida pela alimentação externa é de 9 volts (via transformador), mas muitos dos *chips* só precisam de 5 volts para funcionar, e uma tensão superior provocaria danos irremediáveis. Outros dos *chips* precisam de +5, +12 e −12 volts, de

acordo com os respectivos terminais GRN (massa ou terra).

Cabe aqui uma palavra de aviso para os inevitáveis eternos curiosos: abrir a máquina para uma investigação interna começará por invalidar a garantia do computador oferecida pelos fabricantes ou vendedores e só não provocará danos se houver imenso cuidado. Quem conseguir conter-se deve esperar que a garantia acabe para então proceder às suas explorações.

Cinco parafusos de cabeça em cruz (tipo *phillips*), colocados na parte inferior da caixa, mantêm fechado o computador. Depois de se desligar a ficha e de desapertar os parafusos, invertem-se as faces do aparelho antes de levantar a tampa superior! É preciso muito cuidado! O teclado está ligado à parte inferior por dois cabos de fita muito frágeis, que só podem ser retirados das suas fichas muito suavemente, permitindo depois uma remoção completa da tampa superior. Observa-se então à vontade o interior do computador. Contudo, não se pode deixar cair nada lá para dentro, especialmente objectos de metal. Um outro parafuso mantém o circuito impresso ligado à tampa inferior, mas por baixo daquele pouco mais há que linhas de soldadura em cobre.

Ao montar de novo o computador, é preciso cuidado com os cabos de fita que vão dar ao teclado, pois a sua fragilidade fá-los partir à menor tentativa de os dobrar. Muita atenção: nunca deixar nada dentro da máquina!

CAPÍTULO 8

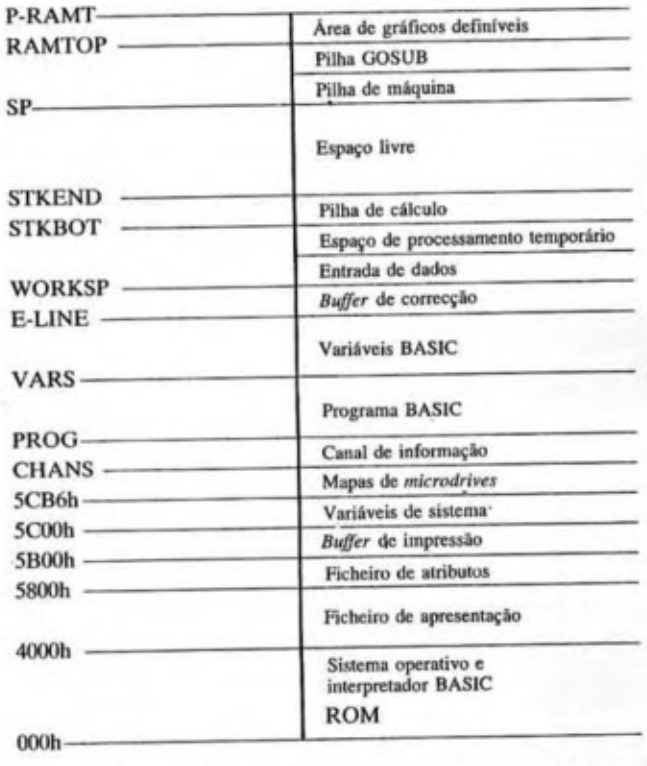
O mapa de memória

A limitação mais importante imposta pela maioria dos microprocessadores ao desenho de computadores é a do tamanho da via de endereços. Com 64 k *bytes* à sua disposição, os primeiros microcomputadores tinham espaço de sobra, mas, com a quebra de preços nos *chips* de memória, aliada à procura crescente de maiores capacidades de armazenamento de dados, os produtos mais recentes utilizam quase sempre todos os endereços disponíveis. O *Spectrum* de 48 k entra nesta categoria. A disposição dos endereços de memória é, pois, designada por «mapa de memória», e no *Spectrum* este mapa está distribuído da forma que se segue.

Entre os endereços decimais 0 e 16383 (0000h a 3FFFh), localiza-se uma ROM de 16 k que mantém o *software* de sistema (incluindo as rotinas que fornecem a saída e a entrada via teclado, *écran* e altifalante) e que inclui ainda o interpretador de linguagem BASIC. A RAM encontra-se contida entre a localização 16384 e o topo do mapa. No computador de 16 k a RAM só chega ao endereço 32767 (7FFFh), ficando livres os restantes endereços. A máquina de 48 k tem a RAM em todos os endereços disponíveis até ao 65535 (FFFFh); mas nem todas as localizações RAM estão à nossa disposição em qualquer dos dois computadores. Se estudarmos o mapa de memória da figura 8.1, veremos que, de 4000h para cima, existe um grande espaço ocupado antes da localização PROG, que é o local de armazenamento do início de qualquer programa em BASIC. Repare-se que a figura 8.1 não está desenhada a escala.

Os dados dos *pixels* do *écran* são armazenados entre 4000h e 57FFh, espaço a que se chama «arquivo de apresentação» *(display file)* — que mantém a informação de memória sobre se um qualquer ponto do *écran* é papel *(paper)* ou tinta *(ink)*. O espaço ocupado é grande — 6 k *bytes* —, mas representa uma penalização para a obtenção de gráficos de boa qualidade.

Figura 8.1
Mapa de memória do *Spectrum*
Variáveis de sistema ou endereços hexadecimais



Começando em 5800h, existe outro arquivo para fins de apresentação, o «arquivo de atributos». Vale a pena relembrar aqui que a resolução de cores do *Spectrum* é baixa, sendo suficientes 300h *bytes* para os dados que levam a *chip* ULA a gerar a parte respeitante à cor de uma dada imagem.

Logo a seguir aos atributos, aparece um bloco de memória com 100h de capacidade, designado por «*buffer* de impressão», que em conjunto com os programas faz funcionar a impressora de baixo custo existente para as máquinas da Sinclair. É neste *buffer* que uma linha de texto é transformada em oito linhas de 256 *pixels*, dando origem às formas dos caracteres. Estes dados são então enviados para a impressora, sob a forma de uma corrente seriada de ONs e OFFs, que vai controlar as agulhas de impressão à medida que estas correm sobre a superfície do papel.

AS VARIÁVEIS DE SISTEMA

O bloco de endereços seguintes, a que se chama as «variáveis de sistema», encarrega-se do funcionamento dos programas de sistema próprios da máquina. Qualquer programa totalmente armazenado em ROM está limitado pelo facto de que não pode guardar variáveis dentro da sua área de memória, à excepção dos poucos *bytes* que consegue manter nos registos do processador e que não são usados pelo programa em si. Pode, é claro, alojar os *bytes* na pilha, mas reavê-los de forma aleatória é totalmente impossível. Por consequência, a área de memória compreendida entre 5C00h e 5CB6h é posta de lado para armazenamento de informações tais como as das cores a empregar na impressão do *écran* ou as localizações de memória das variáveis dos programas em BASIC; ou até à do começo do próprio programa BASIC (esta será guardada noutro local quando se ligam outros equipamentos ao *Spectrum*).

Encontramos uma lista das variáveis de sistema no manual; a relação indica sucintamente quais os dados que elas armazenam. Várias delas podem ser alteradas pelo programador, com a mira de obter determinados efeitos, mas algumas, se mudadas, causarão o *crash* irremediável do programa e da própria máquina. Por exemplo, a variável RAMTOP, localizada no endereço 5C82h, pode ser sujeita a um POKE de valor inferior, para enganar o computador sobre a existência de certas localizações de memória — normalmente, quando se executa a instrução NEW, limpa-se toda a memória, mas com este artifício as localizações

acima do seu novo RAMTOP não serão afectadas. Por outro lado, se aplicar um POKE de valor diferente ao PROG (o endereço em que começa o programa em BASIC), o interpretador não conseguirá encontrar esse programa quando fizer o RUN, provocando um *crash*. Com sorte, talvez se faça um POKE de um valor correspondente ao do início de uma das linhas do programa, mas mesmo assim todas as linhas anteriores ficarão perdidas.

Vale a pena referir que os valores mantidos pelas variáveis de sistema de dois *bytes* são armazenadas através dos próprios métodos do *Z80* — isto é, o *bit* menos significativo em primeiro lugar. Damos um exemplo deste facto, que também demonstra a vantagem de se usarem números hexadecimais.

PROG armazena um endereço que é um número de 16 *bits*, pelo que necessita de dois *bytes* de espaço de memória, 5C52h e 5C43h. Se o BASIC estiver no seu lugar normal, um PEEK sobre esses dois valores deverá dar um resultado padrão. Introduzindo-se PRINT PEEK 23635 e PRINT PEEK 23636, deverá obter-se, respectivamente, 203 e 92. Para descobrir que endereços representam estes valores, será preciso multiplicar o segundo valor por 256: PRINT 256*PEEK 23636 dará 23552. Somando agora o valor da primeira localização, introduzindo PRINT 256*PEEK 23636+PEEK 23635, o resultado terá de ser 23755, isto é, (256*92)+203.

Se o *Spectrum* conseguisse ser persuadido a fornecer as suas respostas em hexadecimal (e existem programas que oferecem esta possibilidade), o resultado de um PEEK sobre a variável de sistema RAMTOP seria CBh e 5Ch. Ora nós podemos multiplicar um número hex por 256 adicionando simplesmente dois zeros a esse número: 5C torna-se 5C00h, que somado a CBh dá 5CCBh. Qual é a versão décimal de 5CCBh? Lendo simplesmente ao contrário os dois *bytes* hex, chegamos ao valor real de dois *bytes* armazenado na RAMTOP. É verdade que de nada serve usar números hex numa máquina que os não pode compreender, mas através da aplicação de um programa «monitor» evitaremos muitos cálculos difíceis. Quem quiser ter mesmo a

certeza de que 23755 é na verdade o ponto de partida de um programa em BASIC, tente o procedimento a seguir descrito.

Introduz-se uma simples linha de BASIC num computador previamente limpo — por exemplo: 100 REM Primeira e única linha — e faça-se um POKE de zeros aos dois primeiros *bytes* da área do programa. Isto é, faça-se POKE 23755,0 e POKE 23756,0. Lista-se agora o programa, e vê-se qual a rotura causada. Isto sucede porque cada linha é armazenada com os dois primeiros *bytes* a representar o número dessa linha. Fazendo-se-lhe um POKE com outros valores, descobrir-se-á que, para aumentar potencialmente a confusão, esses números de linha serão memorizados da «maneira correcta ao contrário», ou seja, com o *byte* mais significativo em primeiro lugar.

Regressemos ao mapa da memória, para examinar o que fica entre o fim das variáveis de sistema e o início de qualquer programa em BASIC. Encontramos aí uma área de RAM designada por «mapa de *microdrives*» e que é inexistente se estes não estiverem aplicados ao computador. Por fim, mesmo antes de chegarmos à área dos programas BASIC, surge-nos uma pequena tabela de dados conhecida por «canal de informação». O *Spectrum* utiliza um método de entrada e de saída de dados que atribui um número de canal a cada um dos seus tipos de comunicação disponíveis.

O dispositivo que de momento constitui o «fluxo» activo é o que recebe e envia dados. Por exemplo, se o *écran* for o fluxo activo, os dados armazenados para esse canal de saída na tabela de «canais de informação» serão usados por uma rotina de saída multifuncional, que descobrirá o paradeiro das rotinas específicas dedicadas à saída de *écran*. Se abrirmos o fluxo para a impressora, o comando PRINT levará directamente para o *buffer* da impressora os dados que normalmente iriam para os ficheiros de apresentação da transmissão de TV.

Este método parecerá uma complicação excessiva das tarefas a executar, mas torna-se útil quando se trata de juntar mais canais ao sistema. O conjunto das rotinas I/O (*input/output*, entradas/saídas) já existe, de modo que, analisando a informação do

fluxo em dado momento, pode mandar-se cada uma delas processar o seu caso especial. A tabela forma-se durante o processo de inicialização do sistema, através da verificação de quais os canais presentes (incluindo os periféricos), adaptando-se à realidade que se lhe deparar. Experimente-se o programa seguinte, que provoca uma saída para um canal que não é o da parte normal do *écran*. O símbolo Y (cardinal), que representa o «fluxo», é introduzido pela tecla 3 em modo capital e *extended*.

```
10 PRINT Y 1; «Normalmente isto estaria no topo»
20 IF INKEY$=""THEN GO TO 20
```

O programa ilustra o que sucede quando fazemos sair dados através da selecção do canal um, que é a parte de baixo do *écran*, normalmente reservada para pedidos de INPUT e para as mensagens de erro. Se tentarmos outros números, receberemos em princípio um aviso de erro, pois não existe informação disponível para a rotina de saída, de modo a que esta consulte a tabela à procura de outros canais.

BASIC E VARIÁVEIS

Alcançamos assim a parte da memória que armazena qualquer programa em BASIC. O tamanho desta área, bem como da imediatamente a seguir, a «área das variáveis», depende das dimensões do programa carregado e do número de variáveis que usa. Cada vez que introduzimos uma linha de BASIC, o conteúdo da memória acima do ponto onde vai ser armazenada essa linha é «empurrado» de maneira a deixar espaço para a nova linha, e os ponteiros apropriados, mantidos como variáveis de sistema, são alterados em conformidade. O espaço seguinte destina-se à alteração das linhas de um programa *(editing);* quando seleccionamos uma, ela move-se para este espaço, e uma sua cópia segue para a parte inferior do *écran*. Movendo o cursor, faremos as alterações que pretendermos, até premirmos a tecla ENTER; então, a linha com o número igual é procurada, reeditada e substituída pela nova linha corrigida, através do *buffer* de

reedição. É também neste último que se criam as linhas completamente novas, à medida que as vamos escrevendo e sempre antes de carregarmos na tecla ENTER.

Aparecem a seguir no mapa espaços que podem ser expandidos para guardar dados temporários, tais como números entrados em resposta a um pedido INPUT, ou cadeias *(strings)* que estão a ser manipuladas. Antes de chegarmos ao espaço livre, encontramos um tipo especial de pilha que serve na programação do sistema para processar números. Como antes mencionámos, o *Spectrum* serve-se de um método de representação de algarismos que precisa de cinco *bytes* para os armazenar; isto permite manipular tanto valores altos como baixos com um grau de precisão bastante aceitável. Ao processar esses valores, o computador serve-se da pilha de cálculo.

MEMÓRIA LIVRE

O ponto de partida da área de memória livre (espaço livre) sobe e desce consoante os blocos em baixo se expandem ou contraem, durante a entrada de programas, alterações e tempo de execução. No topo do espaço livre existem mais duas pilhas, a da máquina e a de GOSUB. A UCP serve-se do primeiro destes, com o registo SP apontando sempre para a localização que é, no momento, o «topo» da pilha. Na verdade, a pilha cresce de cima para baixo; os primeiros valores regista-os sempre na parte mais alta da memória e o SP apontará para endereços mais baixos à medida que novos dados forem introduzidos. A segunda pilha é usada pelo interpretador de BASIC no armazenamento da instrução RETURN, bem como linhas e valores significativos empregados como regressos de GOSUB. O último valor memorizado é 3E00h, que representa um número de linha ilícito, de modo que o BASIC acusa um erro logo que tentamos fazer um RETURN sem o prévio GOSUB.

Por fim, no limite superior da memória, encontramos um bloco de *bytes* FF58h ou FFFFh no *Spectrum* de 48 k e 7F58h ou 7FFFh na versão de 16 k. É o espaço de armazenamento dos padrões gráficos de «utilização pré-definida» *(user-definable),*

que a princípio são formados pelos contornos das suas letras correspondentes, mas cujo conteúdo pode ser alterado se definirmos outros padrões.

Devemos aqui realçar que, ao ligar o computador, nenhum dos endereços da RAM contém quaisquer dados relevantes, de modo que os valores necessários devem aí ser introduzidos durante a inicialização do sistema. Esta tarefa, bem como muitas outras, é executada no breve espaço de tempo que medeia entre a aparição de um *écran* branco, no momento da ligação à corrente, e a impressão da mensagem de direitos de autor *(copyright):* © muito rápido em termos humanos, mas mais que suficiente para que um computador execute enorme número de operações.

Concluímos assim o nosso passeio guiado ao mapa da memória. Muito do que aqui ficou expresso será aprofundado em breve, mas de momento vamos debruçar-nos sobre a forma como o computador consegue comunicar com o mundo exterior.

CAPÍTULO 9

O teclado

Um computador terá pouquíssima utilidade se não for capaz de receber informação. Temos duas maneiras de passar informação a um *Spectrum:* através da *cassette* ou do teclado. Poderemos carregar a memória com dados de um gravador de *cassettes,* mas os programas necessitarão com certeza de outros parâmetros, introduzidos via teclado. Mesmo quando o *écran* transmite a habitual mensagem «Prima qualquer tecla para começar», está à espera de dados, na sua forma mais simples, vindos de uma tecla.

A maneira como um teclado passa a informação para um microcomputador varia de máquina para máquina. Algumas interrompem o que a UCP esteja a fazer, e chamam a atenção a si próprias de cada vez que têm algo para passar: o método seguido depende do tipo de *software* que se incluiu durante o fabrico do computador.

O *Spectrum* utiliza uma aproximação ao problema ligeiramente diferente: o teclado em si é um dispositivo completamente passivo, mas o sistema operativo verifica-o a intervalos curtos e regulares, para se certificar se premimos alguma tecla. Se assim sucedeu, o sistema operativo armazena o valor numa certa localização da memória, que a programação do sistema lê, actuando de acordo com o indicado. Para percebermos como ocorrem estes intervalos regulares, precisamos de analisar algumas das funções do processador *Z80* que até aqui ainda não mencionámos.

O USO DO TECLADO A PARTIR DO CÓDIGO MÁQUINA

Os dois pinos a estudar são o NMI e o INT, com os quais o projectista de microcomputadores torna possível uma máquina que, sendo capaz de fazer uma coisa de cada vez, é também capaz de repartir o tempo gasto entre várias tarefas. Por exemplo, algumas impressoras, como as dos velhos terminais de telimpressão, que já todos vimos (e ouvimos), são muito lentas: quando se

lhes dá ordem de impressão de um carácter, observam um período de espera considerável, em termos de computadores, antes que nos permita imprimir nova letra. Em lugar de mantermos o computador parado, sem fazer nada, durante esse compasso de espera, podemos aproveitá-lo para outra coisa, até que a impressora diga que está pronta a imprimir de novo; este aviso de prontidão surge através do pino INT.

Porquê então preocuparmo-nos com velhas impressoras e *Z80s* impacientes? Enquanto o sistema de interrupção estiver disponível, podemos usá-lo para fazer com que o *Spectrum* pareça esquizofrénico. Cada quinquagésimo de segundo, o *chip* da ULA emite um sinal de vídeo, que começa com um «impulso de estrutura», e com o qual se sincroniza a televisão ou o monitor. Durante o processo de geração do sinal de vídeo, a ULA produz um impulso de zero volts, que envia para a UCP através do pino INT, isto uma vez em cada quinquagésimo de segundo. O programa base do *Spectrum* configurou o *Z80* para responder a este sinal da forma mais simples possível. Depois de terminar cada instrução, o *Z80* verifica a entrada de interrupção; se detectar um impulso negativo, guarda o conteúdo do contador do programa (situado na pilha), carregando-o com o valor 0038h (56 décimal).

Conseguimos assim fazer com que o processador se afaste do que esteja a fazer, apontando-o para a rotina de código máquina armazenada em 0038h da ROM. Esta rotina perfaz três coisas. Em primeiro lugar, incrementa a variável de sistema chamada *frames* («estrutura»); trata-se de um valor de três *bytes* mantido na RAM, nas localizações 5C78h a 5C7Ah, e que mede o tempo através do comando (BASIC) PAUSE.

Em segundo lugar, a rotina «varre» o teclado para detectar a activação de qualquer tecla. Se for esse o caso, descodifica a tecla, tomando em conta as teclas de *shift* e, se não houver mais teclas premidas, armazena o código na RAM, no endereço 5C58h. Por fim, o valor prévio do contador do programa é recuperado da pilha e recarregado: assim, o *Z80* regressa exactamente ao ponto onde estava antes da ocorrência da interrupção.

O programa assegura que os registos da UCP não sejam alterados. Todos os que são usados ficam com os respectivos valores armazenados, prontos a serem reinseridos depois de cessar a interrupção, de modo que um programa interrompido não é afectado de maneira nenhuma. O sistema operativo só toma uma acção ao ser premida uma tecla — quer esteja em modo de comando, aceitando a tecla premida com uma instrução, quer esteja a executar um programa, caso em que pura e simplesmente ignora a tecla.

Quando estamos à vontade com o código máquina, a rotina principal de varrimento tem um certo interesse. Podemos, por exemplo, lidar com o *rollover*, característica que é mais fácil de demonstrar do que de explicar: liga-se a máquina e apaga-se o logótipo da Sinclair premindo a tecla PRINT; mantém-se se então premida a tecla «p», que se auto-repetirá imprimindo uma série de «pp» no *écran*. Sem deixar de carregar no «p», faz-se o mesmo com outra tecla qualquer — a repetição pára imediatamente, porque a máquina deixa de compreender qual delas queremos mandar para o *écran*. Se levantarmos o dedo do «p», aparecerá logo a segunda letra. Note-se que a repetição automática também é processada pela rotina de varredura.

Vejamos agora como as teclas estão ligadas. Existem 40 teclas, cada uma com o seu próprio interruptor. O conjunto destes forma uma matriz que liga as oito linhas de endereço altas a cinco linhas numeradas de KBD 9 a KBD 13. Dentro da ULA, os circuitos detectam se o *Z80* está a executar uma operação IN (FEh), respondendo de modo a colocar as voltagens das linhas KBD nos cinco *bits* baixos da via de dados. O estudo da figura 9.1 mostra como isto se consegue: é bom lembrar que, apesar de poder construir o circuito com peças soltas, o *Spectrum* contém-nas como parte integrante da ULA. A figura 9.1 é um bom exemplo da descodificação de portos. Quando tanto IOQR como RD estão baixos, o *Z80* tenta ler o porto cujo endereço é mantido nos oito *bits* baixos da via de endereços. A função deste circuito é fazer com que, quando a UCP executa IN (FEh), as cinco linhas de dados baixas sejam alimentadas a partir das linhas do teclado.

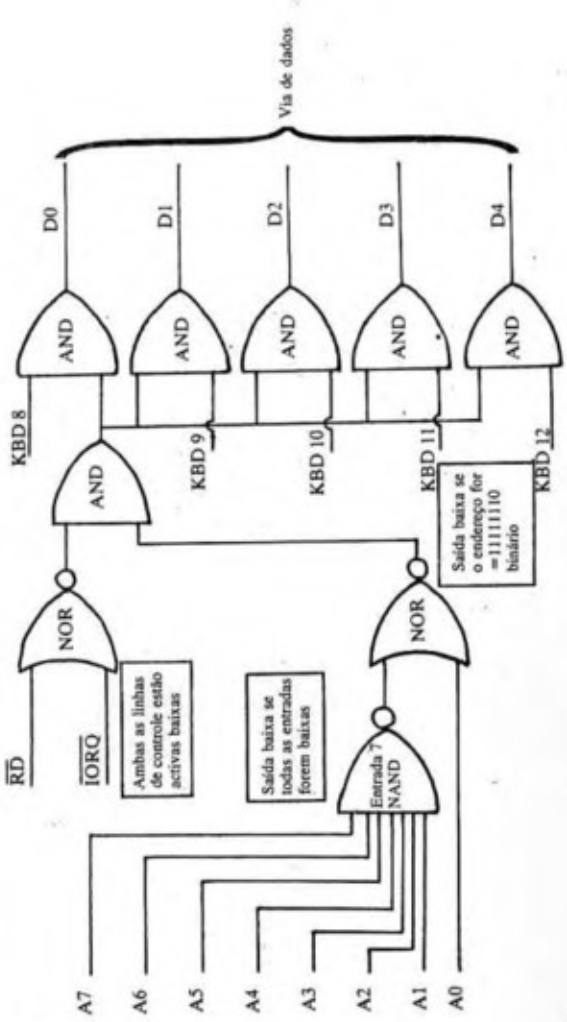


Figura 9.1

Na figura 9.2, seguindo a linha marcada por KBD 9, verifica-se como ela pode ser ligada a qualquer das oito linhas de endereço altas. É importante compreender como é que se põe alta a linha pela voltagem do outro lado da resistência; KBD 9 é alta até premirmos uma das teclas, fazendo-se então o contacto com a linha de endereço apropriada. Mesmo quando isto sucede, a linha de dados só ficará baixa se a linha de endereço com que esteve ligada for também baixa. O diagrama contém, nas linhas de endereço, símbolos que ainda não foram explicados e que estão marcados com o nome «díodos». Como já dissemos, um díodo é o mais simples dos dispositivos semicondutores, pois tem a propriedade de só deixar passar a corrente num sentido. Estes díodos asseguram que, se mais de uma tecla for premida, o seu efeito não vá aquém do díodo, através das linhas de endereço. Esta protecção evita que a pressão sobre as teclas interfira com outras operações da via de endereços.

O método de colocação de diferentes níveis nas linhas de endereço altas depende do comportamento do *Z80* durante a execução de instruções IN. Há duas maneiras de ler dados vindos de um porto. Uma delas assume a forma IN A, (N), em que N é o número do porto que deverá fornecer dados ao registo do acumulador. Este número, de oito *bits*, é colocado nas oito linhas de endereço baixas, e o conteúdo prévio do acumulador é transferido para as oito linhas altas. Ao ler os dados, se carregarmos o acumulador com, por exemplo, 7Fh (127 dec), que é, na forma binária, 01111111, e depois dermos a instrução IN A, (FEh), o bit 0 do resultado lido no acumulador será alto — a menos que a tecla SPACE seja premida, caso em que será baixo.

Podemos ver, assim, por este exemplo simples, um modo de testar teclas individuais.

A outra instrução IN do *Z80* reveste-se da forma IN A (C), e permite carregar o registo C com o número do porto desejado. Ainda mais útil é o facto de o valor do registo B ser colocado nas oitó linhas altas, de modo que o acumulador não precisa de estar sempre a ser recarregado.

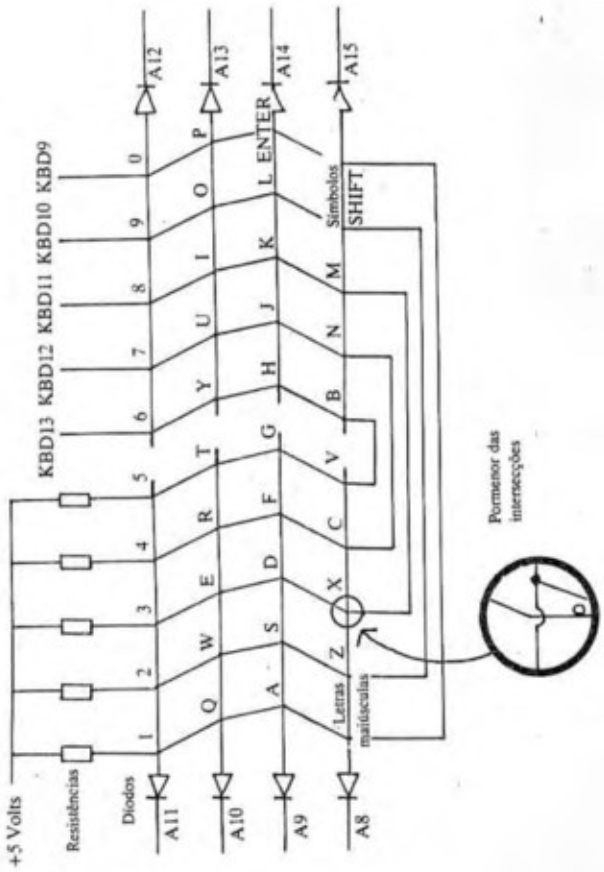


Figura 9.2

O VARRIMENTO DO TECLADO

Vejamos outra vez a figura 9.2, que mostra a disposição das teclas e o modo como estão ligadas as linhas de endereço e de porto. O programa 9.1 fornece uma ilustração simples do varrimento do teclado, que verifica quais as teclas que são premidas em cada um dos lados do computador. O programa está em código máquina porque assim manipulamos facilmente cada um dos *bits* que passam do teclado para a UCP. Se bem que o *Spectrum* possua as instruções OUT e IN dentro do seu BASIC, o teste individual de *bits* é lento e complicado, pois os seus argumentos estão em decimal. Podemos modificar o programa 9.1 de maneira a testar qualquer meia fiada de teclas, ou mesmo determinadas teclas numa coluna, se «mascararmos» mais do que os três *bits* altos lidos para o registo A; aplicando a A um AND com, digamos, 10h, testaremos unicamente a primeira tecla de cada coluna. O programa tem interesse potencial para os programadores de jogos; pode usar-se num programa um BASIC ou misturar o código com um outro programa escrito em código máquina.

Tabela 9.1. Código maquina para varrimento do teclado

Endereço	**Código hex**	**Assembler**
RAMTOP+1	010000	LD BC,0000h
	3EF0	LD A,F0h
	DBFE	IN A,(FEh)
	2F	CPL
	E61F	AND 1Fh
	2802	JR Z,+2
	CBC1	SET 0,C
	3E0F	LD A,0Fh
	DBFE	IN A,(FEh)
	2F	CPL
	E61F	AND 1Fh
	C8	RET Z
	CBC9	SET 1,C
	C9	RET

De resto, os três *bits* altos que são lidos ao mesmo tempo que os cinco *bits* do teclado devem ser sempre «mascarados». Quando apareceram, as versões dois e três do *Spectrum* causaram enormes confusões, pois estes *bits* assumiam um comportamento aleatório, fazendo com que naves espaciais, monstros e demais família pulassem no *écran* como que dotados de vontade própria!

Programa 9.1. Varrimento do teclado

```
  10 REM Funcao de varrimento
  11 REM     do teclado
  12 REM Programa 9.1
  20 REM Abaixamento do RAMTOP
  21 REM
  30 RESTORE : LET x=(PEEK 23730
+256*PEEK 23731)-26: CLEAR x
  40 REM
  41 REM Colocar Codigo Maquina
  42 REM       na Memoria
  43 REM
  50 LET x=(PEEK 23730+256*PEEK
23731)+1
  60 FOR y=x TO x+24: READ z: PO
KE y,z: NEXT y
  70 REM
  71 REM Print Endereco USR
  72 REM
  80 PRINT AT 1,4;"A Rotina e ch
amada pela";TAB 7;"funcao USR ";
x
  90 REM
  91 REM    Exemplo
  92 REM
 100 PRINT AT 8,5;"Prima quaisqu
er teclas"
 110 PRINT AT 12,13;: LET a=USR
x
 120 IF a=0 THEN PRINT "NADA
"
 130 IF a=1 THEN PRINT "ESQUERDA
"
 140 IF a=2 THEN PRINT "DIREITA
"
 150 IF a=3 THEN PRINT "AMBAS"
 160 PRINT AT 16,10;"USR ";x;" =
";a
 170 GO TO 110
 180 REM
 181 REM  Codigo Maquina
 182 REM
 190 DATA 1,0,0,62,240,219,254,4
7
 200 DATA 230,31,40,2,203,193,62
,15
 210 DATA 219,254,47,230,31,200,
203,201,201
```


E que tal se usássemos a rotina de varrimento incluída no *Spectrum* para nosso próprio proveito? Lembremos que ela actua por interrupções; a interrupção disfarçada pode ser desligada, em código máquina, com o auxílio da instrução DI, que faz com que a UCP ignore o seu pino MI. Há uma série de maneiras de extrair um valor do teclado, dependendo a escolha do método do tipo de programa que estivermos a escrever.

O facto de deixarmos as interrupções estabelecidas consegue interferir com a fita de tempo envolvida, mas, se as deixarmos assim de propósito, faremos como se segue.

Testa-se o *bit* 5 das bandeiras, que é a variável de sistema alojada em 5C3Bh (23611 dec): se estiver estabelecida, é porque foi premida uma tecla desde a última vez que se estabeleceu essa bandeira. Encontramos o código ASCII para essa tecla na localização variável do sistema LAST K, 5CO8h. Se só estivermos interessados no valor do *caps shift* (tecla de maiúsculas), sabemos que este se encontra guardado no endereço 5CO4h (23556 dec) e é actualizado em cada interrupção, de modo que não precisaremos de testar as bandeiras. Acerca de CAPS SHIFT: podemos estabelecer *caps lock* (fixação de capitais) quer a partir de código máquina quer de BASIC, estabelecendo o *bit* 3 de FLAGS 2 (5C69h). Para vermos esta possibilidade, façamos POKE 23658,8: INPUT A$.

É provável que apareça uma interrupção entre o teste das bandeiras e a obtenção do valor, que por sua vez poderá provocar um resultado falso. Numa rotina em código máquina, este contratempo pode ser evitado de duas maneiras. Fazendo com que a UCP marque o tempo com uma instrução HALT, leva-la-á a perfazer um NOP até receber a ordem de interrupção. Se inserirmos HALT no código, imediatamente antes da leitura das

variáveis de sistema, obteremos um intervalo para respirar de cerca de um quinquagésimo de segundo, antes da próxima interrupção.

Se quisermos deixar as interrupções estabelecidas, mas ao mesmo tempo proteger-nos contra os «piratas» de código máquina, não nos esqueçamos de estabelecer o modo de interrupção 1 no início do programa. Um método alternativo consiste em desligar as interrupções em conjunto, chamando na altura própria a rotina de varrimento. Como esta fica em 38h, o código *op* RST 38 executará a tarefa só com um *byte*. A variável de sistema será estabelecida em consonância, e o registo A passará a reter o valor de LAST K. Como a rotina de varrimento possui uma instrução EI, não há necessidade de se usar DI outra vez.

Vale a pena realçar que a ROM possui uma outra rotina de varrimento, que testa o teclado para verificar se a tecla BREAK foi premida. Quando executa um programa em BASIC, ou quando aplica determinados comandos, como SAVE e LOAD, o *Spectrum* chama frequentemente esta rotina, de modo a permitir-nos a paragem da máquina, com a consequente possibilidade de regresso ao modo de comando. Para utilizar a rotina de teste de BREAK, encontramo-la em 1F54h: ela limita-se a levar a bandeira de transporte (*carry*) a zero no caso de serem premidas simultaneamente as teclas BREAK e CAPS SHIFT. Esta sub-rotina é muito curta e podemos copiá-la para outros fins ou modificá-la para responder a outras teclas.

O teclado não se limita a produzir os códigos ASCII, gerando igualmente os valores BASIC que representam funções e comandos. Esta característica depende do «modo» em que o teclado está a operar e que é indicado pelo cursor; se este for um K a relampejar, a tecla premida a seguir será traduzida pelo sistema operativo como uma ordem ou comando. Actuando sobre a tecla «p» em modo de comando, gera-se o valor F5h: este aparece no *écran* como um PRINT, porque a rotina de saída reconhece-o como um caso especial, de modo que vai verificá-lo numa tabela localizada na ROM entre 95h e 204h. Neste caso, a rotina verifica quais os caracteres a enviar para o *écran* ou, melhor, qual

é o fluxo de saída seleccionado para esse momento. Deste modo, o *Spectrum* poupa tempo e espaço, pois não precisa de armazenar cinco *bytes* ao representar PRINT, bastando-lhe decifrar um *byte* para cada comando ou função. É por isso que não temos de nos preocupar com o soletrar de palavras tais como RANDOMIZE!

O teclado do *Spectrum* é um dispositivo relativamente simples de varrer, e o sistema operativo fá-lo com brevidade, mas se estivermos a trabalhar em código máquina pode acontecer que necessitemos de criar a nossa própria rotina de varrimento. A tabela da ROM situada entre 205h e 22Bh ajudar-nos-á a descodificar os valores enviados pelo varrimento, pois contém os códigos ASCII na mesma ordem da do teclado do computador. Quem for um bom dactilógrafo, se tiver adaptado o *Spectrum* a um teclado do tipo «profissional», conseguirá «ganhar a rotina de varrimento»: que tal tentar escrever uma rotina que memorize, digamos, os dezasseis últimos batimentos de tecla? Não, nós não fizemos nenhuma, e de resto somos muito maus dactilógrafos!

CAPÍTULO 10

Imagens no «écran»

Se um certo pioneiro escocês da electrónica tivesse conseguido levar a sua avante, a televisão no canto da nossa sala teria um disco com lentes à volta dos bordos, girando grande velocidade. Para felicidade da indústria do vídeo, a história legou-nos um sistema muito melhor. No entanto, quer os discos giratórios de Baird quer os equipamentos electrónicos dos nossos dias baseiam-se no mesmo princípio de varrimento; a única desvantagem do método actual é que é mais difícil compreendê-lo.

O MODO DE APRESENTAÇÃO DAS IMAGENS DE TELEVISÃO

A válvula que apresenta a imagem que vemos nas televisões é designada por «tubo de raios catódicos». A figura 10.1 representa-a em corte.

Faz-se o vácuo no interior do tubo, e a face frontal é coberta com partículas à base de fósforo. A parte posterior contém um «canhão de electrões», no qual uma peça de metal designada «cátodo» é aquecida, gerando-se ao mesmo tempo uma voltagem considerável entre este e a face fosfatada, que impele os electrões, já excitados pelo calor, a viajar para a área de atracção altamente positiva que existe na face do tubo. Como nada se opõe ao seu deslocamento, o fluxo de electrões percorrerá o vácuo até ao seu destino final.

Qualquer campo magnético, bem como outras voltagens, exerce influência notória no caminho seguido por esses electrões, de modo que conseguimos empregar enrolamentos electromagnéticos, bem como outros cátodos e «ânodos» (a versão positiva dos cátodos) para deflectir e focar o «raio» de electrões que se deslocam até ao fósforo. Este raio pode ser apontado para qualquer ponto do *écran*. Diga-se aqui que as propriedades especiais do fósforo são de grande significado: se for bombardeado com muitos electrões, o fósforo emitirá um brilho luminoso e, dentro de certos limites, quanto mais electrões maior será o brilho produzido — o que se torna muito útil.

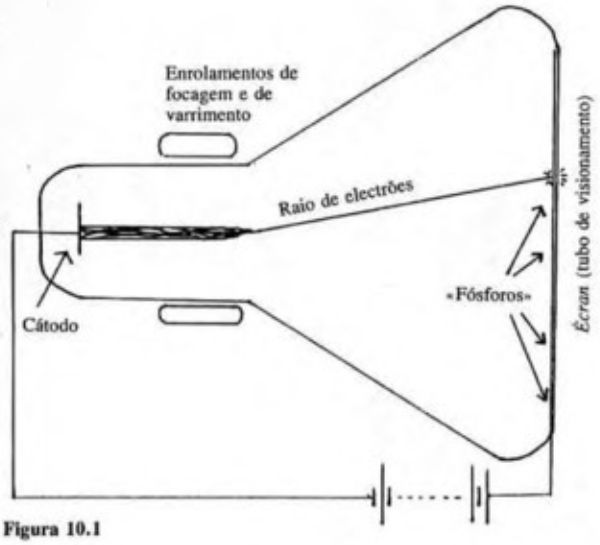


Figura 10.1

Temos assim um método de iluminar um determinado ponto do *écran* da TV, por aplicação de voltagens de comando adequadas aos vários componentes do tubo de raios catódicos. Daqui até percebermos como poderemos «pintar» uma imagem apontando o raio a várias partes do *écran*, fazendo-as brilhar, vai um passo. E, se movermos o raio para outros pontos, regressando ao inicial antes que esmoreça, temos o nosso problema basicamente resolvido. A figura 10.2 mostra como isso se consegue: divide-se a área do *écran* em linhas horizontais, que são varridas uma a uma. A intensidade do raio é variável, proporcionando assim as diferenças de brilho necessárias. No momento em que se tem de passar a uma nova linha, reduz-se a intensidade do raio e todo o processo se repete até se passar por todas as linhas do *écran*.

Neste ponto, dirige-se o raio para o início do percurso, mas desta vez vai encher os espaços situados entre o primeiro conjunto de linhas.

Todo este processo decorre com uma velocidade espantosa, gastando-se somente 25 décimos de segundo até ao regresso do raio à posição inicial de varrimento.

Para se conseguir uma imagem compreensível no *écran*, os circuitos associados têm de ser alimentados com um certo número de dados, construídos normalmente por impulsos que obrigam ao varrimento de nova linha ou de nova moldura. Estes sinais controlam também o brilho requerido.

Os dados estão contidos num sinal analógico designado por «sinal de vídeo», que usa impulsos negativos para dirigir os circuitos de linha e de moldura, enquanto a informação relativa ao brilho é gerida por uma voltagem positiva variável, originada a partir dos impulsos de sincronização de linha. A figura 10.3 representa um gráfico de uma linha de sinal de vídeo. Os sinais recebidos através da propagação electromagnética, ou vindos do terminal TV de um computador, são sinais modulados da mesma forma que os sinais de rádio, de modo que podem ser transmitidos e seguidamente descodificados pelos circuitos de «sintonização» do aparelho de televisão.

A cor, porém, torna o assunto muito mais complicado. Qualquer cor pode ser formada a partir da mistura de três fontes puras de cores primárias — vermelho, verde e azul.

As televisões a cores tem três espécies de camadas de fósforo impressas no *écran*, e o tubo possui não um, mas três canhões de electrões. Uma difusão judiciosa entre os cátodos e o *écran* faz com que o raio de um canhão só incida sobre uma determinada espécie de fósforo. O tamanho das diferentes áreas de cor é tão pequeno que à distância se confundem, para a vista humana, numa única fonte de luz. Com os três canhões ligados no máximo, o *écran* será de um branco puro e brilhante, mas se o relacionamento entre as saídas das três fontes for convenientemente regulado, veremos várias cores na TV. Os monitores coloridos de alto nível só requerem que os três sinais de cor sejam

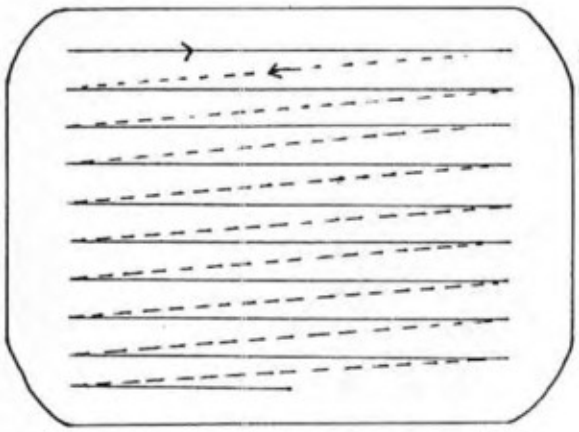


Figura 10.2

enviados em separado, conseguindo depois uma imagem a cores de muito boa qualidade.

As televisões normais, porém, precisam que a informação sobre a cor seja incluída no que usualmente designamos por «sinal de vídeo composto codificado», processo utilizado pelas emissões públicas coloridas. Nestes sistemas, um canal não ocupa muito espaço de difusão, de modo que os receptores monocromáticos (a preto e branco) podem compartilhar o mesmo sinal. O processo para codificar a informação sobre a cor é complexo: basta dizer que os sinais de «diferença de cor» são modulados e misturados com o vídeo de uma forma muito semelhante àquela com que se consegue que mais de um canal de TV possa estar «no ar» simultaneamente e de modo a que um circuito de sintonização os possa distinguir entre si.

A rápida descrição anterior deverá pelo menos familiarizar com alguns dos termos associados ao «vídeo», fornecendo uma ideia clara sobre a forma como um aparelho de televisão constrói

as suas imagens, através de uma varredura muito rápida do *écran*.

Voltemos agora ao *Spectrum*, para vermos como ele gera o sinal de vídeo, através do modulador, de modo a alimentar um aparelho de televisão.

Figura 10.3

O SINAL DE VÍDEO DO «SPECTRUM»

Já aqui referimos que parte significativa da ULA está dedicada à apresentação do *écran*; os dados destinados à geração da imagem estão armazenados em dois «arquivos» na área compreendida entre 4000h e 5AFFh. A saída do *chip* «de fábrica» inclui os impulsos de sincronização, bem como os níveis analógicos de cor e de vídeo. A temporização dos impulsos é proporcionada pela subdivisão da frequência do relógio.

A UCP pode enviar à ULA informação de controle de alguns dos aspectos da geração do sinal de vídeo. Enviando dados aos três *bits* baixos do porto FEh, podemos alterar a cor da moldura *(border)* — os valores correspondem aos números de cor usados pelo BASIC: 0 é o preto, 1 é o azul, e por aí adiante até ao 7, que é o equivalente do branco. Por aqui podemos ver como se

processa a mistura de cores — cada um dos três *bits* controla uma cor primária, de modo que se misturarmos vermelho com verde enviando o binário 110 para o porto, a cor apresentada na moldura será o amarelo, que é a cor resultante da sobreposição das tonalidades vermelha e verde.

O que se passará no interior da ULA quando o *Spectrum* está a gerar imagens? Sem estar dependente do processador, ela gera um impulso de estrutura que inícia a imagem no topo do *écran*, e depois emite impulsos de linha a intervalos regulares. Para a informação de vídeo analógica, a ULA serve-se dos *bits* da moldura como dados destinados a gerar os correspondentes níveis de brilho e de cor, de modo a serem inseridos entre os impulsos de linha do vídeo composto. Depois de enviar um certo número de linhas contendo a moldura, a ULA dá início à primeira linha propriamente dita. Um impulso de sincronização linear é seguido por um *bit* de moldura, começando depois a acção. Chamando a si o comando até aí exercido pelas linhas de endereço dos 16 k mais baixos da RAM, através do já mencionado circuito multiplex, a ULA lê o primeiro *byte* armazenado no «ficheiro de apresentação» *(display file)*, bem como o primeiro *byte* do ficheiro de atributos. Só depois disso passa o comando da RAM para a UCP.

Pode aqui surgir um problema, se a UCP tentar ler ou escrever do ou para o mesmo bloco de memória no momento em que não puder controlar as linhas de endereço: ser-lhe-á impossível obter quaisquer dados significativos se tentar ler o bloco ao mesmo tempo que a ULA. Para se evitar tal situação, a via de endereços é monitorizada pela ULA e, se esta detectar que está para acontecer um «engarrafamento», congela pura e simplesmente a UCP, parando o impulso de relógio que até aí estava a enviar, até ao momento em que deixar de necessitar de controlar as linhas de endereço.

A informação relacionada com os primeiros oito *pixels* do *écran* está agora no *chip* da ULA, e cada *bit* do *byte* do ficheiro de apresentação representa 1 para INK ou 0 para PAPER. O ficheiro de atributos controla a informação da cor da seguinte

forma — os *bits* 0, 1 e 2 determinam a cor dos *pixels* de INK, e os *bits* 3, 4 e 5 a do PAPER.

Se o *bit* 6 estiver estabelecido, significa que o BRIGHT e todo o conjunto dos oito *pixels* estão «prontos», isto é, aumenta a luminescência de ambas as cores. O *bit* 7 é o controle do *flash:* se for estabelecido em zero, os *pixels* não são afectados; se for 1, a saída de um circuito oscilador de baixa frequência da ULA determina se se alternarão os valores de fundo e tinta (*paper* e *ink*) antes da respectiva codificação.

Deste modo, enviam-se os primeiros oito *pixels* para o *écran* e o processo repete-se outras 31 vezes até toda a linha superior ter sido afixada. Só neste momento se reinstala para o restante da linha a cor da moldura. A ULA executa o mesmo trabalho para todas as linhas do *écran*. Contudo, uma linha de *pixels* não é armazenada imediatamente a seguir à anterior. Mostraremos como o mapa de memória do *écran* está disposto num determinado momento. Cada uma das linhas continua a ser obtida e transmitida até se alcançar o fundo do *écran,* momento em que se produzem mais linhas de moldura, seguindo-se novo impulso de estrutura; e o processo repete-se de novo. É deste modo que a ULA transfere para o mundo exterior o conteúdo dos ficheiros de atributos.

Existe um *chip (LM 899)* que ajuda a codificar a informação de cor antes de o sinal ser enviado para o modulador. A partir daqui, a emissão é semelhante à das estações de TV, dirigindo-se para o aparelho receptor através do cabo próprio. Se se verificar um acesso às zonas da memória compreendidas entre os endereços 4000h e 7FFFh, o que poderia desestabilizar o trabalho da UCP, esta pára os impulsos do relógio até o perigo ter passado.

Vejamos agora a disposição verdadeira dos ficheiros de atributos. Parecerá de início que o método seguido é muito confuso; no entanto, é importante que a ULA possa varrer esta área com um mínimo esforço de descodificação. Poderá pois compreender-se que o padrão usado faça mais sentido se estudarmos os endereços em binário.

O *écran* está dividido em três secções, cada uma usando 2 k de

memória para armazenar os dados dos *pixels*. Estas secções ocupam, respectivamente, os endereços 4000h e 47FFh para o topo do *écran,* 4800 a 4FFFh para o meio e 5000h a 57FFh para a parte inferior. Cada secção contém oito filas de caracteres. A linha superior da primeira destas filas de caracteres é mantida pelos primeiros 32 *bytes* de cada secção.

Os sete blocos seguintes, de 32 *bytes* cada, contêm dados semelhantes para as restantes sete filas de texto de cada uma das secções. Subsequentemente, as linhas de *pixels* das filas de caracteres são armazenadas da mesma maneira — isto é, todas as segundas linhas, depois as terceiras, e assim por diante, para um total de oito linhas de *pixels* que constituem uma fila de espaços da caracteres.

Este assunto é complicado para ser descrito por palavras, de modo que é melhor ilustrá-lo com um programa simples. Introduza-se o programa 10.1 e execute-se. Ele pede dois valores para usar na demonstração; sugerimos que, de início, se tente 255 para o primeiro valor e qualquer número de oito *bits* para o segundo. O programa encherá assim, de forma sequencial, todo o ficheiro de apresentação com o primeiro *byte,* e de seguida as três secções do *écran* serão preenchidas com pontos coloridos. Repare-se no efeito de «estore veneziano» que se desenrola à medida que as filas de caracteres se formam gradualmente.

Quando o *écran* estiver cheio, ocorrerá qualquer coisa muito mais dramática — o programa faz um POKE ao segundo valor, colocando-o no ficheiro de atributos, localizado imediatamente acima do ficheiro-D 1 (D-fila 1). O efeito conseguido é muito mais rápido, pois este ficheiro é bastante mais pequeno. Dissemos aqui que as coisas podem ser bem mais fáceis em binário, e podemos por este exemplo ver que assim é na realidade. Se escrevermos um número de 16 *bits* para representar as localizações do *écran* e usarmos símbolos para indicar quais os *bits* que se relacionam directamente com os parâmetros pretendidos, ficaremos com qualquer coisa como:

010SSLLLRRRBBBBB

Estas letras tem o seguinte significado: S é o número da secção com 0 no topo e 2 (10 binário) no fundo. L é o número da linha de *pixels* dentro da fila de caracteres (um número de três *bits* pode significar uma das oito linhas). R é o número da fila e B indica a posição do *bit* na fila, ou seja, o número da coluna. O endereço divide-se claramente em dois *bytes:* podemos incrementar o *byte* alto de maneira a colocar as sete linhas de *pixels* subsequentes em qualquer espaço de carácter.

Programa 10.1. Disposição do *écran*

```
  10 REM Exame da disposicao
  11 REM     do ecran
  12 REM Programa 10.1
  20 REM  Iniciacao
  21 REM
  30 INK 0: PAPER 7: BORDER 7: F
LASH 0: BRIGHT 0: CLS
  40 REM
  41 REM Obtencao de valores
  42 REM
  50 INPUT "Entre o byte (0>255)
 ";byte
  60 IF byte<0 OR byte>255 THEN
GO TO 50
  70 INPUT "Entre a cor (0>255)
";cor
  80 IF cor<0 OR cor>255 THEN GO
 TO 70
  90 REM
  91 REM Alteracao do Ecran
  92 REM
 100 FOR x=16384 TO 22527: POKE
x,byte: NEXT x
 110 REM
 111 REM Alteracao de cores
 112 REM
 120 FOR x=22528 TO 23295: POKE
x,cor: NEXT x
 130 REM
 131 REM Apresentacao de valores
 132 REM
 140 PRINT AT 2,4;"Byte= ";byte;
AT 6,4;"Cor= ";cor;AT 10,2;"Prim
a uma tecla para alterar"
 150 REM
 151 REM Aguarde pela accao da
 152 REM          tecla
 160 IF INKEY$="" THEN GO TO 160
 170 GO TO 10
```


O *Spectrum* serve-se de um método muito directo para imprimir caracteres no *écran*. Possui, na ROM, uma tabela de «silhuetas» que, quando submetidas a um POKE para o *écran*, formarão as letras. Vimos atrás que a rotina de impressão no *écran* pode ser chamada em 10h e dissemos também que esta rotina é um «pau para toda a obra», manuseando outros canais dentro do princípio de que foi escolhido o canal correcto. Possui ainda outras capacidades: pode mover a posição de impressão quando se selecciona o *écran;* faz a pergunta «SCROLL?» se se acabar o espaço de impressão; estabelece as cores; e pode mesmo expandir os parâmetros BASIC quando os enviamos para impressão.

Contudo, no âmago desta rotina tão útil encontramos um código que está sempre a verificar a posição actual de impressão. Este código utiliza a tabela de caracteres ASCII que lhe foi fornecida pelo registo A no momento em que a rotina foi chamada, de modo a localizar o correspondente padrão de pontos na ROM. Ela copia o primeiro *byte* desse padrão para o *écran*, incrementa o seu ponteiro para a tabela da ROM e também incrementa o *byte* alto do registo que mantém o endereço do *écran*, de modo a que este aponte para a linha seguinte de *pixels*, logo abaixo no *écran*. O *byte* seguinte do padrão de pontos é então copiado para o ficheiro-D, sendo o processo repetido até os oito *bytes* estarem nas partes correctas da memória do *écran*.

A partir do momento em que o padrão está no ficheiro de apresentação, é automaticamente enviado para o *écran* pela ULA. A rotina de saída estabelece também o *byte* de atributos do espaço de carácter de acordo com as cores que estiverem determinadas nesse momento e actualiza a posição de impressão. Esta é então armazenada de duas formas — como números de linha e de coluna (como os que são usados na instrução PRINT AT) e como um endereço absoluto para o primeiro *byte* do espaço do carácter. Deste modo, o endereço «real» do *écran* tem de ser calculado com menos frequência do que seria no caso de só as coordenadas do *écran* serem colocadas na memória.

Esta rotina usa outra variável de sistema, a CHARS, que

mantém um endereço 256 vezes menor que o do início da tabela de formas de caracteres. Para a maioria destes, só é necessário multiplicar os seus códigos ASCII por oito e adicionar o resultado a CHARS. Isto permite localizar o endereço do bloco de oito *bytes* que contém o formato de um determinado carácter. O valor normal de CHARS é 3C00h, pois a tabela situa-se entre 3D00h e 3FFFh da ROM principal, mas como a variável CHARS é também mantida na RAM, podemos fazer com que o sistema operativo se sirva de qualquer conjunto de caracteres desenhado de propósito, e armazenado numa outra área diferente da memória. Provoca-se uma certa confusão tentando fazer POKE 23606,8: para ser maior ainda, executa-se POKE 23607,0.

Utiliza-se um método muito parecido para armazenar uma outra categoria de formas, os «gráficos pré-definidos» *(user definable graphics)*, que são mantidos no topo da memória. Quando se lhe pede que imprima um caráter gráfico, a rotina de saída vai buscar o endereço da tabela de UDG, obtendo aí a «silhueta» pretendida. Antes de serem definidas novas formas, a RAM assegura a manutenção de uma cópia de parte da tabela da ROM, de modo que obtemos letras maiúsculas normais. Além disso, o *Spectrum* possui um conjunto de «gráficos pré-definidos», sob a forma de blocos de «um quarto de carácter», que são obtidos por cálculos internos. Se analisarmos as suas formas, relacionando-as com os respectivos códigos de carácter, e traduzirmos esses valores para o sistema binário, descobriremos facilmente como se consegue tudo isto (vejam-se os códigos no manual do computador). Os quatro quartos podem ser considerados como os *bits* de um número binário de quatro *bits*, com o prefixo binário 1000. Deste modo, a «silhueta» é construída por uma inteligente combinação de código máquina.

MANIPULAÇÃO DO «ÉCRAN» A PARTIR DO CÓDIGO MÁQUINA

Só existem duas possibilidades para manipular o *écran* a partir do código máquina: escrever as suas rotinas de impressão de propósito, o que não é tão difícil como parece, ou aproveitar a

programação incluída no *Spectrum*. As rotinas disponíveis na ROM são mais do que suficientes para escrever um programa que não necessite de gráficos rápidos e de alta qualidade. Podemos mandar todos os códigos de comando BASIC para a rotina de saída, e mesmo imprimir nas duas linhas inferiores do *écran*, bastando para tal alterar o valor de DF SZ, que mantém o número de linhas reservado para as mensagens de erro (não esquecer de o fazer voltar ao normal antes de regressar ao BASIC; caso contrário o computador não terá espaço para imprimir o OK!). A quem pretenda melhorar o jogo de acção *Buck Rogers e o Planeta Zoom* através da criação de novas rotinas, boa sorte! Podemos deixar aqui duas ajudinhas...

A primeira relaciona-se com a moldura. A maior parte dos programas comerciais para o *Spectrum* são prejudicados por relâmpagos negros na moldura de cada vez que mudam as cores, e no entanto, é bem simples evitar este inconveniente. Antes de se mudar os três *bits* baixos do porto FEh para a nova cor, basta usar a instrução HALT, de modo a que a UCP passe a aguardar uma interrupção antes de continuar. No momento do recomeço, a interrupção coincidirá com um impulso de estrutura, de modo que a mudança da cor da moldura não ocorrerá durante o tempo real de exposição da imagem.

O segundo aspecto é personificado pelo programa 10.2, uma listagem de código máquina destinada a calcular endereços de *écran*. Aqui só fornecemos os códigos de operação, e não um programa BASIC para os carregar, pois esta listagem tem de ser incorporada como sub-rotina dentro de um outro programa em código máquina.

A rotina pode ser colocada em qualquer parte da memória livre. Para a usar, carrega-se o registo E com o número da coluna horizontal, como na instrução AT, e fornece-se ao registo D o número do *pixel* do *écran*, que começa no topo com a linha zero e acaba em 192 dec. Depois disto, chama-se a rotina. Cumprida esta instrução, o registo HL conterá o endereço do *byte* pretendido do ficheiro de apresentação, e o registo BC manterá o *byte* correspondente do ficheiro de atributos. Se quisermos, encurta-

remos o programa, de maneira a lidar só com o primeiro *byte* de cada carácter; também não há necessidade de usarmos os mesmos registos, desde que estas alterações de programação bastem aos nossos propósitos.

Mantivemos este programa simples para demonstrar com maior acuidade o método utilizado; cada qual deverá encontrar alguma utilidade no princípio aqui enunciado.

Programa 10.2. Rotina em código máquina para cálculo de endereços de *écran*

Endereço	Código hex	Assembler
SCALC	F5	PUSH AF
	7A	LD A,D
	E6F8	AND F8h
	0F	RRCA
	0F	RRCA
	0F	RRCA
	6F	LD L,A
	E618	AND 18h
	F640	OR 40h
	67	LD H,A
	7A	LD A,D
	E607	AND 07h
	84	ADD A,H
	67	LD A,L
	E607	AND 07h
	0F	RRCA
	0F	RRCA
	0F	RRCA
	83	ADD A,E
	6F	LD L,A
	7A	LD A,D
	E6C0	AND C0h
	07	RLCA
	07	RLCA
	F658	OR 58h

	47	LD B,A
	4D	LD C,L
	F1	POP AF
	C9	RET

Ao introduzir	D mantém o número de linha de *pixel* E mantém o número da coluna
Regressos de rotina	HL mantém o endereço do ficheiro de apresentação BC mantém o endereço do ficheiro de atributos

Escrever rotinas para gráficos de alta qualidade não é tarefa fácil, mas dispomos já de uma boa base de partida. Alguns dos efeitos que se conseguem com uma pequena «dose» de código máquina, particularmente se esta for aplicada ao ficheiro de atributos, representam uma agradável surpresa.

CAPÍTULO 11

Som

«Se ao menos ele pudesse falar...», dirá quem fica a olhar, à procura de compreender, para uma mensagem de erro surgida no fundo do *écran*. Na verdade, o *Spectrum* pode ser persuadido a falar, e a alguns dos leitores talvez já se tenha deparado um desses exemplos. No entanto, a programação adequada para esse fim só dá resultados muito primários, por boa que seja. O *hardware* do *Spectrum* destinado à produção de som consiste numa única instrução, BEEP, que reproduz, por exemplo, os *clics* que ouvimos à medida que as teclas vão sendo premidas. Os sinais enviados para as ligações MIC provêm da mesma fonte.

COMO OPERA O PORTO «BEEP»

A maior parte do circuito simples do *beep* está contida na ULA. As portas responsáveis pela descodificação da tentativa da ULA em escrever ou ler para o porto FEh ou do mesmo porto estão ligadas a um dos pinos daquela, associado a vários componentes externos. Se uma operação de escrita para o porto FEh estabelecer o *bit* 4, gera-se no pino da ULA uma voltagem suficiente para activar o pequeno altifalante incluído no computador. Se se estabelecer o *bit* 3, a voltagem gerada é incapaz de activar o altifalante, mas é a bastante para estar presente na ligação *mic*, de modo a gravar em *cassette*. A onda quadrada muito pronunciada que se forma quando uma voltagem passa de alta a baixa é um tanto ou quanto modificada pelo emprego de um condensador que delimita os picos de tensão.

O já referido pino da ULA está também ligado ao terminal *ear*; a leitura do *bit* 6 do porto FEh provoca uma resposta 1 ou 0, dependendo da voltagem de entrada. Dentro da ULA, uma porta especial designada por *Schmitt trigger* (multivibrador) serve para aproximar o valor do sinal até 1 ou 0. Se elevarmos lentamente a voltagem na entrada desta porta, a saída manter-se-á baixa até um

valor de entrada da ordem dos 2,5 volts, limite a partir do qual o dispositivo «dispara», elevando a saída.

A programação de sistema do *Spectrum* cria um *beep* da seguinte forma: estabelece-se o *bit* 4 do porto FEh e uma rotina em ciclo faz com que a UCP aguarde por um determinado espaço de tempo, relacionado com a frequência do *beep* pretendido. Quando esse intervalo se esgotou, o *bit* 4 do porto é restabelecido, e o intervalo do ciclo volta a repetir-se. Consegue assim enviar um ciclo sonoro para o altifalante, e todo o processo é repetido até que se gerem os ciclos suficientes para a duração do som pretendido.

Há dois aspectos a realçar nesta rotina. Se duplicarmos a frequência, isto é, o número de ciclos por segundo, através da compressão do período de retardamento, a duração do *beep* será também comprimida, a menos que dupliquemos o número de ciclos gerados. A rotina *beep* calcula quantos ciclos são necessários para a produção de uma determinada frequência, de modo a fazer com que o som dure o tempo pretendido. Ao fazê-lo, serve-se abundantemente das rotinas de vírgula flutuante.

O outro aspecto relaciona-se com os três *bits* baixos do porto FEh, que comandam a cor da moldura; se escrevermos para esse porto indiscriminadamente, sem estabelecer os *bits* nos seus valores correctos, a moldura alterar-se-á. A rotina *beep* analisa a cor existente na moldura num determinado momento, mediante uma variável de sistema chamada BORDCR: os *bits* 3, 4 e 5 indicam qual a cor que esta activa. Estes *bits* são movidos para os três *bits* baixos do acumulador, de modo que a moldura se mantém constante durante a execução de uma instrução OUT (FE), A.

Tabela 11.1. Código máquina para efeitos sonoros

Endereço	Código hex	Assembler
RAMTOP+1	3A485C	LD A,(5C48h)
	1F	RRA
	1F	RRA

```
1F      RRA
E607    AND 07h
0EFF    LD C,FFh
2600    LD H,00h
44      LD B,H
CBE7    SET 4,A
D3FE    OUT (FEh),A
10FE    DJNZ -2
44      LD B,H
CBA7    RES 4,A
D3FE    OUT (FEh),A
10FE    DJNZ -2
CBE7    SET 4,A
D3FE    OUT (FEh),A
10FE    DJNZ -2
CBA7    RES 4,A
D3FE    OUT (FEh),A
10FE    DJNZ -2
24      INC H
0D      DEC C
20E2    JR NZ,-30
C9      RET
```

UTILIZAÇÃO DO PORTO «BEEP» A PARTIR DO CÓDIGO MÁQUINA

A utilização do porto *beep* a partir do código máquina sujeita-nos a inúmeras armadilhas, mas oferece a vantagem da velocidade com que os seus parâmetros podem ser alterados. Por exemplo, podem produzir-se ciclos alternantes de diferentes durações ou alterar-se a forma dos ciclos, com a parte baixa a durar menos tempo que a parte alta. Existe uma outra armadilha provocada pelos circuitos de geração do sinal de vídeo, que param a UCP quando tanto esta como a ULA tentam servir-se ao mesmo tempo dos primeiros 16 k da RAM. Qualquer código máquina aí localizado correrá de forma atabalhoada, sendo interrompido com frequência. Se bem que seja aconselhável desligar as interrupções, tal como o faz a rotina da ROM, estas causam

muito menos confusão que o parar constante do programa, se este se situar nos 16 k mais baixos da RAM. Se pretendermos criar tons puros com a porta *beep*, teremos de nos cingir ao código ROM, rapidamente disponível no endereço ROM 3B5h, ou utilizar a máquina expandida.

O programa 11.1 dá um exemplo do que se consegue com uns poucos *bytes*. O programa BASIC pode simplesmente ser escrito, gravado e executado. A listagem de código máquina explica o que se vai passando e permite incorporar o efeito obtido nos seus programas. Repare-se como os tons individuais se misturam uns com os outros, efeito de difícil obtenção se se empregar somente a linguagem BASIC.

Programa 11.1. Rotina do porto *beep* (de som)

```
  10 REM Rotina do Porto de Som
  11 REM Programa 11.1
  20 REM Abaixamento do RAMTOP
  21 REM
  30 RESTORE : LET x=(PEEK 23730
+256*PEEK 23731)-43: CLEAR x
  40 REM
  41 REM   Colocacao do Codigo
  42 REM   Maquina na Memoria
  43 REM
  50 LET x=(PEEK 23730+256*PEEK
23731)+1
  60 FOR y=x TO x+42: READ z: PO
KE y,z: NEXT y
  70 REM
  71 REM  Impressao do Endereco
  72 REM          USR
  73 REM
  80 PRINT AT 1,4;"A rotina e ch
amada pela";TAB 7;"funcao USR ";
x
  90 REM
  91 REM    EXEMPLO
  92 REM
 100 PRINT AT 12,8;"Prima uma te
cla"
 110 IF INKEY$="" THEN GO TO 110
 120 LET a=USR x
 130 GO TO 110
 200 REM
 201 REM   Codigo Maquina
```

```
 202 REM
 210 DATA 58,72,92,31,31,31,230,
7,14,255
 220 DATA 38,0,68,203,231,211,25
4,16,254,68
 230 DATA 203,167,211,254,16,254
,203,231,211,254
 240 DATA 16,254,203,167,211,254
,16,254,36,13
 250 DATA 32,226,201
```

ROTINAS DE GRAVAÇÃO

Vejamos agora, com brevidade, como se processam as funções de gravação e carregamento da *cassette*, executadas pela programação do sistema, mediante rotinas da ROM localizadas entre 4C2h e 9F3h. Se quizermos conceber as nossas próprias rotinas, porventura para conseguir uma forma mais rápida de arquivo de dados, lembremo-nos de que o facto de a ULA parar a UCP significa que a melhor localização para essas mesmas rotinas será na área de memória RAM de uma máquina expandida.

O método de gravação processa-se como se segue: o título do arquivo (e a informação sobre o tipo de arquivo de que se trata) entra na memória como um conjunto *(string)* de *bytes*, que mais tarde representará o cabeçalho. Depois, apresenta-se a mensagem «accione o gravador» *(start tape);* a máquina testa o teclado para verificar se alguma tecla foi premida e, quando isto sucede, o cabeçalho e depois os dados são formatados e dirigidos para a saída *mic*, a fim de serem gravados. Esta formatação consiste, em primeiro lugar, numa tonalidade mestra com uma frequência ligeiramente acima dos 800 ciclos, seguida por um ciclo de 2040 Hz. Os *bits* individuais são depois transmitidos como um de dois ciclos cada: 2040 Hz para representar o zero ou 1020 para o um.

Os primeiros oito *bits* vêm do registo A, que mantém a bandeira que é estabelecida se os dados a enviar forem os do cabeçalho. Depois, cada *byte* a armazenar segue, *bit* a *bit*, para o terminal *ear*, sob a forma de um som. Após cada *byte*, o teclado é testado para verificação de uma possível pressão sobre a tecla

BREAK, e nesse caso a rotina aborta imediatamente. No final dos dados, segue para a fita magnética um último *byte* (o comprimento do bloco era mantido no registo DE), a que se chama o «*byte* de paridade», e que é gerado por uma operação XOR sobre todos os *bytes* em conjunto, à maneira de soma de verificação.

O carregamento de dados a partir da *cassette* processa-se através de um esquema inverso ao acima descrito. A UCP monitoriza o *bit* 6 do porto FEh e, ao detectar o tom mestre, espera pelo primeiro ciclo de 2040 Hz, que assinala a chegada dos dados. Lê então os *bits* enviados pela fita, reagrupando-os em *bytes*. O primeiro destes informa a UCP se ela está a ler um cabeçalho ou os próprios dados, e a programação do sistema reage de acordo com este facto. Quando se descobre um cabeçalho apropriado, a informação sobre o local de armazenamento dos dados (PROG se se tratar de um programa BASIC) e sobre a sua extensão é imediatamente reconhecida. Imprime-se no *écran* o título do arquivo e os dados subsequentes são armazenados na memória do computador.

No fim da recolha dos dados, o *byte* de paridade da fita é comparado com um outro *byte* de paridade que a rotina de carregamento foi construindo a partir dos *bits* enviados. Se houver alguma discrepância a rotina pára com uma mensagem de erro. Durante o espaço de tempo de leitura bem sucedida dos *bits*, o comportamento da moldura reflecte o carregamento, alternando entre amarelo e *ciano* a acompanhar as alterações de voltagem. Pode dar-se o caso de querermos estabelecer duas áreas de RAM adjacentes, formando blocos de zeros e de FFhs (255), e guardá-los com um código de gravação. Observa-se então que a largura das riscas do *écran* duplica, enquanto correrem pelo televisor, à medida que se transmitem os FFhs.

Para os musicalmente dotados, deixamos aqui algumas observações. A instrução BEEP produz notas relacionadas com a escala musical; o uso de SOUND e de código máquina obrigá-los-á a calcular os seus próprios valores. O lá nomal é uma frequência de 440 Hz; dobrando a frequência da nota, o som

resultante será uma oitava mais alto. Calcularemos os semitons intermédios se considerarmos que o lá sustenido multiplicado pela raiz 12 de 2 produzirá a frequência de lá, e assim por diante. A raiz 12 de 2 é aproximadamente igual a 1,059463. Boas composições!

CAPÍTULO 12

Mais caracteres por linha

O número máximo de caracteres que cabem numa linha do *Spectrum*, a nível de apresentação de *écran*, constitui um factor limitativo. Se quisermos usar o computador para fins mais sérios, como o processamento de palavra, as 32 colunas habituais são pouco adequadas; mesmo uns poucos caracteres a mais em cada linha melhoram a utilidade do computador. Colocar 42 caracteres numa única linha não é especialmente difícil se recorrermos ao código máquina, mas neste caso como será possível efectuar a *interface* entre esse código e o BASIC? O programa seguinte destina-se a ser incorporado no nosso próprio utilitário em BASIC, e mostra algumas técnicas simples de manipulação do *écran*, fornecendo ainda um exemplo de como podemos explorar a área BASIC das variáveis para encontrar uma destas.

Em primeiro lugar, vejamos como se deve empregar o programa. O código máquina está armazenado no topo da memória e é protegido de qualquer sobreposição pela aplicação da ordem CLEAR, que abaixa o RAMTOP de forma a deixar o espaço suficiente. O código tem 953 *bytes* de comprimento, dos quais só 283 *bytes* constituem o programa em si. O resto é uma tabela de referência das formas dos caracteres, de modo que a princípio não devemos preocupar-nos com a sua introdução correcta. Para imprimir uma cadeia no *écran*, usaremos a instrução LET a$= «Aquilo que se quer imprimir», seguida por RAND USR 64421 no caso de se tratar de uma máquina de 48 k ou de RAND USR 31653 para os que operam com o *Spectrum* de 16 k. Não há necessidade de nos certificarmos de que a$ é a última variável declarada. Por exemplo, podemos usar a rotina como parte de um ciclo FOR... NEXT sob a forma:

```
LET a$= «Letra minúscula»: FOR X=0 TO 23: RAND USR
64421: NEXT X
```

A rotina reterá a memória da última posição de impressão, da mesma maneira que o faz em BASIC, e movemos essa posição com um comando do tipo AT. Para tal bastará embeber as instruções CHR$(22) seguidas de CHR$(número de linha) e de CHR$(número de coluna). Por exemplo,

LET a$=CHR$(22)+CHR$(5)+CHR$(10)+«Espere por favor»: RAND USR 64421

imprimirá a mensagem na linha 5, coluna 10, sem ter em conta a posição prévia de impressão. Qualquer outro CHR$ abaixo de 32(dec) forçará o texto a começar numa nova linha. Os CHR$ acima de 127 provocarão o fim prematuro da impressão. A rotina não faz tentativas para alterar o ficheiro de atributos; as letras sobrepõem-se aos limites dos atributos.

O *Spectrum* de 16 k não dispõe de espaço suficiente para o programa 12.1. Da mesma forma, os códigos *op* são ligeiramente diferentes, pois a posição da rotina não é independente — por outras palavras, contém endereços absolutos que têm de ser mudados se tivermos de alojar o programa noutro local qualquer. Por isso, devemos usar sempre o programa 12.2, que só carrega a primeira parte do código. Depois de o escrever e gravar sem problemas, fazemos um RUN e então carregamos o monitor, programa 6.1, ou qualquer outra rotina capaz de trabalhar com dados decimais da memória. Teremos então de introduzir os dados do formato dos caracteres, colocando-os do endereço 31928 para cima, servindo-nos dos elementos fornecidos pela versão para 48 k (programa 12.1) a partir do passo 500. Só depois disto testaremos o código máquina e, se ficarmos satisfeitos com o formato dos caracteres, gravaremos a rotina em *cassette*.

A instalação da rotina é muito mais fácil com um *Spectrum* de 48 k. Introduzimos o programa 12.1. O grande volume de instruções DATA parecerá assustador, mas, depois de a primeira secção estar correcta (a que vai até a linha 450), o programa correrá sem qualquer acidente desde que se apague a instrução STOP existente no fim da linha 30. Detectaremos então facilmente os erros na tabela dos caracteres imprimindo todas as letras no *écran*. Com um pouco mais de prática, até desenharemos os nossos caracteres, incluindo letras futuristas ou clássicas góticas.

Quando o programa estiver todo certo, gravamo-lo cuidadosamente em fita. A partir daí, podemos gravar o código em separado, de forma a carregá-lo nos nossos programas, sem nunca esquecermos a indispensável instrução CLEAR.

Programa 12.1. Versão 48 K.

```
    5 REM 42 CARACTERES/LINHA
    6 REM
    7 REM VERSAO para 48K
    8 REM
   10 CLEAR 64412: CLS : PRINT "
      VERIFICACAO DE DADOS"''"
      ESPERE POR FAVOR"
   20 RESTORE : LET total=0: FOR
x=64413 TO 64695: READ byte: LET
 total=total+byte: POKE x,byte:
NEXT x: REM IF total<>32570 THEN
 PRINT "Erro nos dados do progra
ma.": STOP
   30 LET total=0: FOR x=64696 TO
 65367: READ byte: LET total=tot
al+byte: POKE x,byte: NEXT x: RE
M IF total<>61796 THEN PRINT "Er
ro nos dados dos caracteres."
   40 CLS : LET a$="Rotina de 42
colunas instalada acima      do
RAMTOP.Para Gravar use:-"+CHR$ 2
2+CHR$ 3+CHR$ 6+"SAVE '42col'COD
E 64413,953"
   50 RANDOMIZE USR 64421
   60 LET a$=CHR$ 0+CHR$ 0+"Para
usar LET a$=string que quer impr
imir"+CHR$ 0+"e depois RAND USR
64421"
   70 RANDOMIZE USR 64421
   80 LET a$=CHR$ 0+CHR$ 0+"CHR$
22+CHR$ 1+CHR$ c colocara a posi
cao  de impressao na linha 1 e c
oluna c;CHR$ 0 provocara uma nov
a linha"
   90 RANDOMIZE USR 64421: STOP
   97 REM
   98 REM  CODIGO PRINCIPAL
   99 REM
  100 DATA 0,0,0,0,0,0,0,0
```

```
 110 DATA 42,75,92,126,71,254,12
8,200
 120 DATA 35,230,224,254,224,32,
6,17
 130 DATA 18,0,25,24,238,254,160
,32
 140 DATA 11,203,126,35,40,251,1
7,5
 150 DATA 0,25,24,223,254,96,40,
246
 160 DATA 94,35,86,35,120,254,65
,40
 170 DATA 3,25,24,207,34,159,251
,25
 180 DATA 34,161,251,42,159,251,
84,93
 190 DATA 237,75,161,251,167,237
,66,208
 200 DATA 235,126,35,34,159,251,
254,128
 210 DATA 208,254,22,32,24,126,2
54,23
 220 DATA 48,3,50,158,251,35,126
,35
 230 DATA 34,159,251,203,39,71,1
28,128
 240 DATA 50,157,251,24,206,254,
32,218
 250 DATA 165,252,111,38,0,203,3
7,203
 260 DATA 20,203,37,203,20,203,3
7,203
 270 DATA 20,22,0,95,167,237,82,
17
 280 DATA 216,251,25,229,221,225
,33,163
 290 DATA 251,58,158,251,95,230,
24,246
 300 DATA 64,87,123,230,7,15,15,
15
 310 DATA 95,58,157,251,203,63,2
03,63
 320 DATA 203,63,131,95,14,255,5
8,157
 330 DATA 251,230,7,71,62,3,40,1
0
 340 DATA 55,31,203,25,31,203,25
,5
 350 DATA 16,247,119,121,50,164,
251,58
 360 DATA 157,251,230,7,14,0,71,
221
```



```
 370 DATA 126,0,40,9,31,203,25,3
1
 380 DATA 203,25,5,16,247,71,26,
166
 390 DATA 176,18,19,35,26,166,17
7,18
 400 DATA 27,43,221,35,20,122,47
,230
 410 DATA 7,32,212,58,157,251,19
8,6
 420 DATA 50,157,251,254,250,218
,224,251
 430 DATA 175,50,157,251,58,158,
251,60
 440 DATA 254,24,202,224,251,50,
158,251
 450 DATA 195,224,251
 497 REM
 498 REM FORMATO DOS CARACTERES
 499 REM
 500 DATA 0,0,0,0,0,0,0,0
 510 DATA 64,64,64,64,0,64,0
 520 DATA 80,80,0,0,0,0,0,0
 530 DATA 0,80,248,80,248,80,0
 540 DATA 32,120,160,112,40,240,
32
 550 DATA 64,168,80,32,80,168,16
 560 DATA 64,160,72,176,144,104,
0
 570 DATA 64,64,0,0,0,0,0,0
 580 DATA 64,128,128,128,128,64,
0
 590 DATA 128,64,64,64,64,128,0
 600 DATA 0,168,112,248,112,248,
0
 610 DATA 0,32,32,248,32,32,0
 620 DATA 0,0,0,0,0,192,64
 630 DATA 0,0,0,240,0,0,0
 640 DATA 0,0,0,0,0,192,0
 650 DATA 32,32,64,64,128,128,0
 660 DATA 96,144,176,208,144,96,
0
 670 DATA 64,192,64,64,64,64,0
 680 DATA 96,144,16,32,64,240,0
 690 DATA 96,144,32,16,144,96,0
 700 DATA 32,96,160,240,32,32,0
 710 DATA 224,128,192,32,32,192,
0
 720 DATA 112,128,224,144,144,96
,0
 730 DATA 240,16,32,32,64,64,0
 740 DATA 96,144,96,144,144,96,0
```

```
 750 DATA 112,144,144,112,16,16,
0
 760 DATA 0,64,0,0,64,0,0
 770 DATA 0,64,0,0,64,128,0
 780 DATA 0,32,64,128,64,32,0
 790 DATA 0,0,224,0,224,0,0
 800 DATA 0,128,64,32,64,128,0
 810 DATA 64,160,32,64,0,64,0
 820 DATA 112,136,8,104,168,112,
0
 830 DATA 96,144,144,240,144,144
,0
 840 DATA 224,144,224,144,144,22
4,0
 850 DATA 96,144,128,128,144,96,
0
 860 DATA 224,144,144,144,144,22
4,0
 870 DATA 240,128,224,128,128,24
0,0
 880 DATA 240,128,224,128,128,12
8,0
 890 DATA 96,144,128,176,144,112
,0
 900 DATA 144,144,240,144,144,14
4,0
 910 DATA 224,64,64,64,64,224,0
 920 DATA 240,32,32,32,160,64,0
 930 DATA 144,160,192,192,160,14
4,0
 940 DATA 128,128,128,128,128,22
4,0
 950 DATA 136,216,184,136,136,13
6,0
 960 DATA 144,144,208,176,144,14
4,0
 970 DATA 96,144,144,144,144,96,
0
 980 DATA 224,144,144,224,128,12
8,0
 990 DATA 112,136,136,168,152,12
0,0
1000 DATA 224,144,144,224,160,14
4,0
1010 DATA 112,128,96,16,16,224,0
1020 DATA 240,64,64,64,64,64,0
1030 DATA 144,144,144,144,144,96
,0
1040 DATA 136,136,136,136,80,32,
0
1050 DATA 136,136,136,168,168,80
,0
```



```
1060 DATA 136,80,32,32,80,136,0
1070 DATA 136,136,80,32,32,32,0
1080 DATA 240,16,32,64,128,240,0
1090 DATA 224,128,128,128,128,22
4,0
1100 DATA 128,64,64,32,32,16,0
1110 DATA 224,32,32,32,32,224,0
1120 DATA 32,112,168,32,32,32,0
1130 DATA 0,0,0,0,0,0,252
1140 DATA 96,144,128,192,128,240
,0
1150 DATA 0,96,16,112,144,112,0
1160 DATA 128,128,224,144,144,22
4,0
1170 DATA 0,96,128,128,128,96,0
1180 DATA 16,16,112,144,144,96,0
1190 DATA 0,96,144,224,128,112,0
1200 DATA 96,128,192,128,128,128
,0
1210 DATA 0,96,144,144,112,16,22
4
1220 DATA 128,128,192,160,160,16
0,0
1230 DATA 128,0,128,128,128,192,
0
1240 DATA 32,0,32,32,32,160,64
1250 DATA 128,128,160,192,160,14
4,0
1260 DATA 128,128,128,128,128,64
,0
1270 DATA 0,80,168,168,136,136,0
1280 DATA 0,160,208,144,144,144,
0
1290 DATA 0,96,144,144,144,96,0
1300 DATA 0,224,144,144,224,128,
128
1310 DATA 0,96,160,160,96,32,48
1320 DATA 0,160,192,128,128,128,
0
1330 DATA 0,96,128,64,32,192,0
1340 DATA 128,192,128,128,128,96
,0
1360 DATA 0,144,144,144,144,240,
0
1370 DATA 0,136,136,136,80,32,0
1380 DATA 0,136,136,136,168,112,
0
1390 DATA 0,144,144,96,144,144,0
1400 DATA 0,144,144,144,112,16,2
24
1410 DATA 0,240,32,64,128,240,0
1420 DATA 32,64,64,128,64,64,32
```

```
1430 DATA 64,64,64,64,64,64,64
1440 DATA 128,64,64,32,64,64,128
1450 DATA 80,80,0,0,0,0,0
1460 DATA 112,136,168,200,168,13
6,112
```

Programa 12.2. Versão 16 K.

```
    3 REM
    4 REM 42 CARACTERES/LINHA
    5 REM VERSAO para 16K
    6 REM
    7 REM programa 12.2
    8 REM
   10 CLEAR 31644: CLS : PRINT "
      VERIFICACAO DE DADOS"'"
      ESPERE POR FAVOR"
   20 RESTORE : LET total=0: FOR
x=31645 TO 31927: READ byte: LET
 total=total+byte: POKE x,byte:
NEXT x: IF  total<>29824 THEN PR
INT "Erro nos dados do programa.
": STOP
   30 PRINT "O codigo esta' armaz
enado a"'"partir do endereco 316
45,tendo"'"um comprimento de 283
 bytes"'"apo's a introducao dos
dados"'"relativos aos caracteres
,"'"podera' chamar a rotina"'"co
m USR 31653": STOP
   97 REM
   98 REM  CODIGO PRINCIPAL
   99 REM
  100 DATA 0,0,0,0,0,0,0,0
  110 DATA 42,75,92,126,71,254,12
8,200
  120 DATA 35,230,224,254,224,32,
5,17
  130 DATA 18,0,25,24,238,254,160
,32
  140 DATA 11,203,126,35,40,251,1
7,5
  150 DATA 0,25,24,223,254,96,40,
246
  160 DATA 94,35,86,35,120,254,65
,40
  170 DATA 3,25,24,207,34,159,123
,25
  180 DATA 34,161,123,42,159,123,
84,93
  190 DATA 237,75,161,123,167,237
```



```
,66,208
  200 DATA 235,126,35,34,159,123,
254,128
  210 DATA 208,254,22,32,24,126,2
54,23
  220 DATA 48,3,50,158,123,35,126
,35
  230 DATA 34,159,123,203,39,71,1
28,128
  240 DATA 50,157,123,24,206,254,
32,218
  250 DATA 165,124,111,38,0,203,3
7,203
  260 DATA 20,203,37,203,20,203,3
7,203
  270 DATA 20,22,0,95,167,237,82,
17
  280 DATA 216,123,25,229,221,225
,33,163
  290 DATA 123,58,158,123,95,230,
24,246
  300 DATA 64,87,123,230,7,15,15,
15
  310 DATA 95,58,157,123,203,63,2
03,63
  320 DATA 203,63,131,95,14,255,5
8,157
  330 DATA 123,230,7,71,62,3,40,1
0
  340 DATA 55,31,203,25,31,203,25
,5
  350 DATA 16,247,119,121,50,164,
123,58
  360 DATA 157,123,230,7,14,0,71,
221
  370 DATA 126,0,40,9,31,203,25,3
1
  380 DATA 203,25,5,16,247,71,26,
166
  390 DATA 176,16,19,35,26,166,17
7,16
  400 DATA 27,43,221,35,20,122,47
,230
  410 DATA 7,32,212,58,157,123,19
8,6
  420 DATA 50,157,123,254,250,216
,224,123
  430 DATA 175,50,157,123,58,158,
123,60
  440 DATA 254,24,202,224,123,50,
158,123
  450 DATA 195,224,123
```

Vejamos agora como trabalha o programa. Acrescentámos notas explicativas à listagem assembler (no final deste capítulo) que serão de alguma utilidade.

Dividimos a rotina em três secções, sendo a última constituída pela tabela de caracteres acima mencionada. O código máquina, para estes propósitos, serve-se de cinco variáveis: as XPOS e YPOS são *bytes* simples que armazenam as posições actualizadas da coluna e da linha no *écran*. XPOS mantém a localização do *pixel*, de modo que pode ter um valor até 248 (os caracteres têm uma largura de seis *pixels* cada). YPOS guarda o número da linha, dentro dos limites 0 a 23. As outras variáveis, DATA, LENTH e BUFFR, são todas formadas por dois *bytes*. Perdoe-se o modo de escrever LENTH — o equivalente da palavra inglesa *lenght* (comprimento) —, mas a convenção para etiquetas só permite cinco letras em alguns dos programas assembler.

A primeira tarefa da rotina consiste em descobrir o tamanho e a localização na memória de a$. Determinaremos tanto o começo como o fim da área de variáveis BASIC através das variáveis de sistema VARS e ELINE. De cada vez que definimos uma variável em BASIC, o interpretador passa revista a todas as variáveis, começando em VARS, para descobrir se aquela que foi definida já existia antes. Se assim for, o interpretador apaga-a e fecha o espaço vazio deixado atrás antes de colocar a nova variável no fim da área VARS.

Mas — e aqui decerto protestarão —, o que sucede se usarmos instruções como, por exemplo, LET A=A+1 ? De modo a resolver situações como esta, o interpretador decifra sempre qual o significado que deverá ter uma variável, antes de a armazenar. Pelo exposto compreende-se que encontramos sempre as variáveis de definição mais recente no fim da área VARS, mas infelizmente não há processo de pesquisar esta área do fim para o príncipio, o que pouparia tempo.

Ao estudar o capítulo 24 do manual do *Spectrum* vemos que cada tipo de variável pode ser identificado pelo seu primeiro *byte*, que mantém a primeira letra do respectivo nome nos cinco *bits* baixos e um identificador nos três *bits* altos. É possível assim

descobrirmos qual o tipo que queremos localizar.

Em primeiro lugar, a rotina encontra o início da área VARS, servindo-se da variável de sistema, e recolhe o que lá estiver armazenado. Se aí encontrar 80 hex, regressa ao BASIC, pois este valor significa o fim das variáveis. Em toda a extensão desta secção do programa, a rotina emprega o registo HL para saber até que ponto chegou dentro da área; esta técnica é conhecida como «apontar HL para os dados». Se o *byte* recolhido tiver uma configuração 111 nos três *bits* altos, a variável é uma FOR... NEXT (de controle), que tem 18 *bytes* de comprimento, de modo que o ponteiro é adicionado em conformidade, passando a indicar a variável seguinte.

Se o padrão for 101, o número da variável tem um nome ou mais de uma letra. Neste caso, o programa vai identificar os *bytes* seguintes do nome; o interpretador BASIC estabelece o *bit* 7 da última letra do nome em 1; graças a este facto descobriremos o fim do nome, somando depois o comprimento de uma variável numérica (5 *bytes*) ao ponteiro. Uma variável numérica contendo um nome de uma só letra tem a configuração 011, de modo que, quando o programa encontra este tipo, soma cinco ao ponteiro.

Todos os restantes tipos de variáveis tem o seu tamanho armazenado em dois *bytes*, à maneira do *Z80* (isto é, o *bit* menos significativo em primeiro laugar); estes dois *bytes* são o 2 e o 3, de modo que o programa recolhe esta informação. A seguir, testa o nome do *byte* para verificar se se trata de 41 hex, que é o código para uma dimensão única de a$. Assim sendo, acabamos de localizar a variável pretendida. O seu endereço inicial encontra-se armazenado em DATA e o final é calculado adicionando o comprimento ao ponteiro; seguidamente, a variável é armazenada em LENTH. Se a$ não foi localizado, a variável seguinte é procurada de forma idêntica à acima descrita, sendo todo o procedimento repetido até se chegar a uma conclusão ou até se encontrar o valor 80h.

E é assim que se descobre a$ (a menos que nos tenhamos esquecido de o definir)! O processo que explicámos é muito semelhante ao que o BASIC percorre de cada vez que lida com

uma variável, mas o código máquina é muito mais rápido.

Vejamos agora o aspecto da impressão no *écran*. A segunda parte do programa adquire cada *byte* da variável passo a passo, servindo-se da sua própria variável DATA para acompanhar a execução da instrução. De cada vez, e antes de obter um carácter para impressão, compara o valor de DATA com o de LENTH, para ver se se alcançou o final da cadeia. Se encontrar qualquer dos códigos de controle, reage adequadamente, com 16h a mandá-la obter os dois *bytes* seguintes da cadeia e colocá-los em XPOS e YPOS. Para cada carácter, o programa tem de ir descobrir os dados de formato dessa letra à tabela de caracteres. Esta operação é muito simples, pois cada formato ocupa sete *bytes*, de modo que basta multiplicar o código do carácter por sete e adicionar-lhe um endereço de base (o do início da tabela menos 32*7 para os códigos ASCII não utilizados). Temos depois de calcular o endereço do *écran*, a partir de YPOS e dos cinco *bits* altos de XPOS, utilizando uma técnica muito semelhante à que mostrámos anteriormente neste livro, no programa SCALC (programa 10.2).

Por fim, a parte mais difícil. Dispomos já do *byte* da forma, da posição no *écran* e, a partir dos três *bits* baixos de XPOS, sabemos qual a compensação a aplicar à forma antes que esta esteja colocada no *écran*. Primeiro precisamos de construir uma «máscara» que nos assegure que não apagaremos nenhuns *pixels* que não devam ser perturbados. Conseguimo-lo construindo um registo de 16 *bites* contendo o binário 0000001111111111, e depois rodando-o circularmente para a direita em tantos *bits* quantos a compensação exigir. A forma do carácter é também colocada num outro registo de 16 *bits*, que rodamos da mesma maneira. Talvez seja melhor servirmo-nos de um exemplo para compreendermos o que se passa a seguir.

Suponhamos que queremos pôr os seis *bits*, 001110, obtidos dos seis *bits* altos do *byte* da forma da tabela, no *écran* mas com uma compensação de seis *pixels*. A «máscara» terá de ser

1111110000001111 binário

e a forma será rodada para

0000000011100000 binário

Obviamente, temos de lidar com dois *bytes* do *écran*. Digamos que estes contêm:

1111111111111111 binário.

Se fizermos um AND aos *bytes* do *écran* com o *byte* da «máscara», o resultado será um «buraco» de seis *bits:*

1111110000001111 binário.

Se a seguir aplicarmos um OR ao *byte* da forma, obteremos:

1111110011101111 binário

que é o resultado pretendido! O programa tem de processar um *byte* de cada vez, mas o resultado final é o mesmo. Por isso, cada linha da forma é submetida a um POKE para o *écran*, até que todas as sete linhas do carácter tenham sido impressas. Quando a letra está completa, o programa faz avançar as posições de impressão, e obtém o carácter seguinte da cadeia para o imprimir, a menos que se tenha chegado ao fim daquilo que queremos escrever no *écran*.

Esperamos que esta descrição, combinada com o estudo da listagem assembler, dê uma boa ideia de como o programa trabalha. Querendo desenvolver o sistema, é mesmo possível reduzir a largura das letras para cinco *bytes*, à custa da legibilidade. Uma ideia mais razoável (e que pode ser posta em execução!) é servir-nos do espaçamento proporcional: se a letra de que precisamos só ocupar três *pixels*, armazenamos a largura de cada carácter numa tabela, acompanhando as respectivas formas, e obrigamos a rotina a reagir de acordo com estes novos dados.

O programa também indica como se pode conseguir um movimento suave de um *pixel* de cada vez para os formatos ou caracteres gráficos.

Listagem assembler para o programa em código máquina dos 42 caracteres por linha.

Nota: os codigos *op* estão em hexadecimal e destinam-se ao *Spectrum* de 48 k.

Código	Rótulo	Instrução		Comentários
—		ORG	FB9D	Iniciar o programa a partir deste endereço.
00	XPOS	DEFB	00	Configurar oito *bytes* para serem usados pelas variáveis do sistema.
00	YPOS	DEFB	00	
0000	DATA	DEFW	0000	
0000	LENTH	DEFW	0000	
0000	BUFFR	DEFW	0000	
2A4B5C	FIND$	LD	HL,(5C4B)	Estabelecer HL no início da área de variáveis BASIC.
7E	GETBY	LD	A,(HL)	Carregar A com o primeiro *byte* de variável BASIC.
47		LD	B,A	Guardá-lo em B.
FE80		CP	80	Se for o marcador que estiver
C8		RET	Z	no fim da área VARS, regressar ao BASIC.
23		INC	HL	Apontar Hl ao segundo *byte*
E6E0		AND	E0	Mascarar dados das letras de A.
FEE0		CP	E0	Se não for uma variável de
2006		JR	NZ,NUM?	controle, saltar para NUM?
111200		LD	DE,0012	Somar o compr. da var. de
19		ADD	HL,DE	controle ao ponteiro, em HL.
18EE		JR	GETBY	Regressar em ciclo a GETBY.
FEA0	NUM?	CP	A0	Se não for uma var. com
200B		JR	NZ,SNUM?	nome comprido, ir p. SNUM?
CB7E	TEST	BIT	7,(HL)	Procurar última letra do nome;
23		INC	HL	pôr HL a apontar p. o *byte* seg.
28FB		JR	Z,TEST	
110500	ADD5	LD	DE,0005	Adicionar o comprimento de
19		ADD	HL,DE	uma variável numérica a HL.
18DF		JR	GETBY	Saltar para GETBY.

FE60	SNUM?	CP	60	Se for uma variável de número
28F6		JR	Z,ADD5	simples, saltar para ADD5.
5E		LD	E,(HL)	Carregar DE com o compr. da
23		INC	HL	var. BASIC e apontar
56		LD	D,(HL)	HL para o compr. dos dados.
23		INC	HL	
78		LD	A,B	Restaurar o primeiro *byte* da variável para A.
FE41		CP	41	Se a var. BASIC for um a$
2803		JR	Z,STORE	unidimensional, ir p. STORE.
19		ADD	HL,DE	Adicionar o comprimento da variável a HL.
18CF		JR	GETBY	Regressar em ciclo p. GETBY.
229FFB	STORE	LD	(DATA),HL	Armazenar a localização de a$ em DATA.
19		ADD	HL,DE	Carregar HL c. o end. depois
22A1FB		LD	(LENTH),HL	de a$ e armazená-lo em LENTH.
2A9FFB	PRNTA	LD	HL,(DATA)	Carregar HL com o endereço do carácter a imprimir.
54		LD	D,H	Armazenar HL em DE, para acesso rápido.
5D		LD	E,L	
ED4BA1FB		LD	BC,(LENTH)	Carregar BC c. o end. a seguir a a$.
A7		AND	A	Limpar a bandeira de transporte.
ED42		SBC	HL,BC	Comparar o endereço actual c.
D0		RET	NC	BC e regressar ao BASIC se for maior ou igual a ele.
EB		EX	DE,HL	Restaurar o end. actual em HL.
7E		LD	A,(HL)	Carregar A com o carácter.
23		INC	HL	Apontar HL para o carácter
229FFB		LD	(DATA),HL	seguinte e armazená-lo.
FE80		CP	80	Se o carácter for maior ou igual
D0		RET	NC	a 80, regressar ao BASIC.
FE16		CP	16	Se o carácter não estiver «AT»,
2018		JR	NZ,NL?	saltar para NL?
7E		LD	A,(HL)	Obter a posição da linha.
FE17		CP	17	Se estiver fora dos limites
3003		JR	NC,TOOHI	(acima de 23) ir para TOOHI.
329EFB		LD	(YPOS),A	Estabelecer nova pos. da linha.
23	TOOHI	INC	HL	Apontar HL p. a pos. da col.
7E		LD	A,(HL)	Carregar A com a nova posição da coluna.

```
23              INC   HL          Apontar HL para o carácter
229FFB          LD    (DATA),HL   seguinte e armazená-lo.
CB27            SLA   A           Multiplicar a posição da
47              LD    B,A         linha em A por 6.
80              ADD   A,B
80              ADD   A,B
329DFB          LD    (XPOS),A    Estabelecer a posição da co-
                                  luna num novo valor.
18CE            JR    PRNTA       Regressar em ciclo p. PRNTA.
FE20      NL?   CP    20          Se o carácter for menor que
DAA5FC          JR    NEWLN       20, saltar para NEWLN.
6F              LD    L,A         Colocar A em HL.
2600            LD    H,0
CB25            SLA   L           Multiplicar HL por 7, de
CB14            RL    H           modo a que possa apontar a
                                  tabela de caracteres.
CB25            SLA   L
CB14            RL    H
CB25            SLA   L
CB14            RL    H
1600            LD    D,0
5F              LD    E,A
A7              AND   A
ED52            SBC   HL,DE
11D8FB          LD    DE,FBD8     Carregar DE com um ende-
19              ADD   HL,DE       reço base para a tabela de ca-
                                  racteres e adicioná-lo a HL.
E5              PUSH  HL          Transferir HL p. IX, que agora
DDE1            POP   IX          aponta p. a tabela de formatos.
21A3FB          LD    HL,(BUFFER) Apontar HL p. a área de buffer.
3A9EFB          LD    A,(YPOS)    Carregar A c. a linha do écran.
5F              LD    E,A         Guardar A em E.
E618            AND   18          Mascarar o número da secção.
F640            OR    40          Adicionar 0100000 binário.
57              LD    D,A         Armazenar A em D.
7B              LD    A,E         Restaurar núm. da linha em A.
E607            AND   7           Isolar os últimos três bits.
0F              RRCA              Multiplicar A por 8.
0F              RRCA
0F              RRCA
5F              LD    E,A         Colocar A em E.
```



```
3A9DFB          LD    A,(XPOS)    Carregar A c. o núm. da col.
CB3F            SRL   A           Dividir A por 8.
CB3F            SRL   A
CB3F            SRL   A
83              ADD   A,E         Somar E a A e substituir em E.
5F              LD    E,A         DE aponta agora para a pri-
                                  meira localização do écran.
0EFF            LD    C,FF        Colocar 11111111 bin. em C.
3A9DFB          LD    A,(XPOS)    Carregar A c. o núm. de deslo-
E607            AND   7           camento de pixels requerido.
47              LD    B,A         Armazenar o desloc. em B.
3E03            LD    A,03        Carregar A c. 00000011 bin.
280A            JR    Z,TRNFR     Se o deslocamento for zero,
                                  saltar para TRNFR.
37              SCF               Tornar o transporte igual a 1.
1F        LOOP1 RRA               Rodar o valor binário
CB19            RR    C           0000001111111111 circular-
1F              RRA               mente p. a direita aplicando o
CB19            RR    C           valor do deslocamento.
05              DEC   B
10F7            DJNZ  LOOP1
77        TRNFR LD    (HL),A      Colocar o padrão da máscara
79              LD    A,C         em BUFFR e em BUFFR+1.
32A4FB          LD    (BUFFR+),A
3A9DFB    LOOP2 LD    A,(XPOS)    Carregar A com o deslocamento
                                  dos pixels.
E607            AND   7
0E00            LD    C,0         Carregar C c. 00000000 bin.
47              LD    B,A         Colocar o deslocamento em B.
DD7E00          LD    A,(IX)      Carregar A c. o byte da forma.
2809            JR    Z,JMP2      Se o deslocamento for zero
                                  saltar para JMP2.
1F        LOOP3 RRA               Rodar a forma circularmente
CB19            RR    C           para a direita aplicando o valor
1F              RRA               do deslocamento.
CB19            RR    C
05              DEC   B
10F7            DJNZ  LOOP3
47        JMP2  LD    B,A         BC contém agora a forma.
1A              LD    A,(DE)      Carregar A com o que já está
                                  no écran.
A6              AND   (HL)        Mascarar um espaço e adicio-
```

```
B0              OR      B           nar a forma.
12              LD      (DE),A      Pôr (poke) nova forma no écran.
13              INC     DE          Apontar DE p. próx. byte écran.
23              INC     HL          Apontar HL para BUFFR+1.
1A              LD      A,(DE)      Repetir a impressão para o
                                    segundo byte.
A6              AND     (HL)
B1              OR      C
12              LD      (DE),A
1B              DEC     DE          Restaurar os ponteiros em HL
                                    e em DE.
2B              DEC     HL
DD23            INC     IX          Apontar IX p. a próxima forma.
14              INC     D           Apontar DE p. linha pixel seg.
7A              LD      A,D         Testar D p. verif. se tem estabe-
2F              CPL                 lecidos os três últimos bits.
E607            AND     7
20D4            JR      NZ,LOOP2    Se não estiverem, saltar para
                                    LOOP2.
3A9DFB          LD      A,(XPOS)    Carregar A com a posição da
C606            ADD     A,06        coluna e somar 6.
329DFB          LD      (XPOS),A    Armazenar a nova posição.
FEFA            CP      250 dec     Testar para verificar se A atin-
DAE0FB          JR      C,PRNTA     giu o fim da linha; se não, sal-
                                    tar para PRNTA.
AF      NEWLN   XOR     A           Estabelecer A em zero.
329DFB          LD      (XPOS),A    Colocar a pos. da col. em zero.
3A9EFB          LD      A,(YPOS)    Carregar A com a posição da
3C              INC     A           linha e somar 1.
FE18            CP      18          Se a posição de impressão
CAE0FB          JP      Z,PRNTA     atingiu o fundo, saltar p. PRNTA.
32A0FB          LD      (YPOS),A    Armazenar a nova pos. da linha.
C3E0FB          JP      PRNTA       Saltar regressando a PRNTA.

00---   TABLE   DEFB    ---         Tabela de consulta contendo as
                                    formas dos caracteres.
```


CAPÍTULO 13

O «Spectrum» fala

No capítulo dedicado ao som referimos a possibilidade de um «*Spectrum* falante». Hoje encontram-se nas lojas da especialidade acessórios especiais para a geração de palavra e que a reproduzem quer pelo altifalante do computador quer por dispositivos amplificadores complementares. O problema da incorporação destes efeitos nos programas é que só serão executados em *Spectrums* dotados com dispositivos acessórios idênticos.

O programa seguinte, sacrificando à qualidade, permite incluir palavras e mensagens, que não serão de alta fidelidade mas que no entanto serão perfeitamente compreendidas. O programa correrá em qualquer *Spectrum* de 48 k, e os resultados obtidos passam mesmo num de 16 k, se nos limitarmos a usar algumas pequenas palavras.

Primeiro explicaremos o modo de colocação do programa na máquina. Só se torna necessária a listagem, programa 13.1. Como se observará, contém código máquina sob a forma de instruções DATA, que são colocadas na RAM acima de um RAMTOP previamente alterado. Depois de introduzir o programa (e de ter feito uma cópia de segurança em cassette), executamo-lo para verificar se existem erros nas declarações DATA. Se o programa detectar alguma anomalia, informa-nos do facto com uma mensagem no *écran*. Contudo, como o método de verificação é o «de soma», é possível que dois erros se cancelem um ao outro. Se o programa mesmo assim não trabalhar, faz-se nova verificação de todos os elementos entrados em DATA, ou contraverifica-se o conteúdo da RAM com o programa «Monitor», tanto em relação aos dados decimais da listagem assembler como aos dos códigos *op* hexadecimais.

Programa 13.1. O *Spectrum* fala

```
1 REM O Spectrum fala
2 REM
3 REM Instalacao do Codigo
4 REM
```

```
  10 CLEAR 32767: RESTORE : LET
sum=0: FOR x=32768 TO 32867: REA
D y: LET sum=sum+y: POKE x,y: NE
XT x: GO SUB 50: IF sum<>11375 T
HEN PRINT "Ha um erro nos DATA,"
: STOP
  17 REM
  18 REM       MENU
  19 REM
  20 CLS : PRINT AT 2,6;"O SPECT
RUM FALA";AT 5,7;"MENU:-"'"1-",
"Carregar frase"'"2-","Alterar
frase"'"3-","Gravar frase";AT 1
8,2;"Prima o numero desejado"
  30 LET a$=INKEY$: IF a$<"1" OR
 a$>"3" THEN GO TO 30
  40 GO SUB 100*VAL a$: GO TO 20
  47 REM
  48 REM Recomeco, comprimento
  49 REM
  50 LET com=0: LET compr=28572
  60 LET h=INT (com/256): LET l=
com-(256*h): POKE 32808,l: POKE
32809,h: LET h=INT (compr/256):
LET l=compr-(256*h): POKE 32819,
l: POKE 32820,h: RETURN
  67 REM
  68 REM Accao nas Teclas
  69 REM
  70 IF INKEY$<>"" THEN GO TO 70
  80 IF INKEY$="" THEN GO TO 80
  90 RETURN
  97 REM
  98 REM Carregar a mensagem
  99 REM
 100 CLS : PRINT AT 3,1;"Toque a
 fita e prima uma tecla"''
 110 GO SUB 70: PRINT "A carrega
r...": RANDOMIZE USR 32768
 120 PRINT "Mensagem gravada:-"'
'"Prima uma tecla para a reprodu
zir"''
 130 GO SUB 50: GO SUB 70: PRINT
 "A Reproduzir...": RANDOMIZE US
R 32813
 140 RETURN
 197 REM
 198 REM Alterar a mensagem
 199 REM
 200 CLS : PRINT AT 3,3;"As opco
es de alteracao sao:-"'"1-";TAB
 4;"Alterar o inicio"'"2-";TAB
```



```
4;"Alterar o comprimento"'"3-";
TAB 4;"Recuperar os valores ante
riores"'"4-";TAB 4;"Reproduzir
a frase"'"5-";TAB 10;"Voltar ao
 Menu"
 210 PRINT AT 17,4;"Prima o nume
ro desejado";AT 20,0;"Comeco=";c
om;TAB 5;AT 20,15;"Comprimento="
;compr;TAB 5
 220 LET a$=INKEY$: IF a$<"1" OR
 a$>"5" THEN GO TO 220
 230 PRINT AT 17,4;TAB 25: GO TO
 10*VAL a$+230
 240 INPUT "Novo inicio?";com: L
ET com=ABS com: GO TO 210
 250 INPUT "Novo comprimento?";c
ompr: LET compr=ABS compr: GO TO
 210
 260 GO SUB 50: GO TO 210
 270 LET com=com+8: GO SUB 60: R
ANDOMIZE USR 32807: LET com=com-
8: GO TO 210
 280 GO SUB 60: RETURN
 297 REM
 298 REM Gravacao do Codigo da
 299 REM       Mensagem
 300 CLS : FOR x=32867 TO 32813
STEP -1: POKE x+com,PEEK x: NEXT
 x
 310 INPUT "Nome?";a$: SAVE a$CO
DE 32813+com,compr: CLS : PRINT
AT 10,5;"Rebobine a fita e verif
ique": VERIFY a$CODE
 320 PRINT AT 10,0;"Gravacao O.K
."'"Comprimentoda Rotina=";compr
'''"Prima uma tecla.....": GO S
UB 70: IF com<55 THEN GO TO 10
 330 GO TO 20
 397 REM
 398 REM  Codigo Maquina
 399 REM
 400 DATA 62,15,211,254,243,33,1
00,128
 410 DATA 17,0,240,14,0,6,0,4
 420 DATA 40,7,219,254,230,64,18
5,40
 430 DATA 246,112,79,35,229,167,
237,82
 440 DATA 124,181,225,32,232,251
,201,33
 450 DATA 0,0,9,68,77,33,55,0
 460 DATA 9,229,17,156,111,25,23
```

```
5,225
 470 DATA 58,72,92,203,63,203,63
,203
 480 DATA 63,230,7,79,243,70,4,5
 490 DATA 40,4,47,230,16,177,211
,254
 500 DATA 190,0,0,5,32,248,35,22
9
 510 DATA 167,237,82,71,124,181,
225,120
 520 DATA 32,227,251,201
```

Se o programa foi correctamente introduzido, vamos usá-lo.

O *menu* oferece três opções: a primeira destina-se a fazer entrar a mensagem que se pretende que o computador reproduza. Gravamos a palavra ou a frase em *cassette*, servindo-nos do melhor equipamento de que possamos dispor para o efeito. O melhor será o emprego de um gravador de *cassettes* sem controle automático de nível de gravação, pois quando os aparelhos deste tipo executam gravações com um microfone tendem a acentuar o ruído de fundo existente entre as palavras.

Colocamos depois a *cassette*, enrolada até ao princípio da mensagem, no gravador que usamos habitualmente para trabalhar com o computador, regulando-o para o modo de carregamento *(loading)*. Seleccionamos a opção 1 do *menu*, e o programa pedirá para pôr a fita a correr e para carregar numa tecla. Depois de o fazermos e passado certo tempo, uma nova mensagem no *écran* informará que o carregamento está terminado. Premimos novamente uma tecla qualquer e, se tudo correr bem, o altifalante do *Spectrum* reproduzirá a frase ou palavra em questão.

Se o tempo gasto a carregar a frase for superior a cerca de dez segundos e se a reprodução se resumir a um silêncio (ou a silêncio mais as palvras muito distorcidas) teremos de aumentar o volume do gravador. Se a gravação demorou menos de três segundos, mas mesmo assim a reprodução surgir distorcida, reduzimos o volume. Vamos experimentando até descobrir o nível óptimo, fixando então essa posição do botão do volume. Podemos deste modo colocar correctamente dentro da máquina a nossa mensagem.

A segunda opção do *menu* permite «alterar» a frase, de modo a reproduzir somente a palavra ou palavras e não o ruído ou o silêncio antes e depois delas. Ao princípio, limitar-nos-emos a aumentar o valor do parâmetro «início», até que a palavra que queremos reproduzir seja dita logo após premir uma tecla; depois, reduzimos o valor do «comprimento», para não aparecerem mais ruídos no fim da reprodução. Se estivermos satisfeitos com o resultado, passamos à opção 3, que grava e verifica o código máquina em *cassette*.

Até aqui, apenas gravámos em fita magnética um programa em código máquina, que reproduzirá uma palavra ou frase. Para a incorporar noutro programa, tomamos nota do «tamanho», dado pela opção 3. Não é mais do que o espaço que deve ser reservado para o código, e que podemos colocar em qualquer localização RAM disponível, desde que haja espaço livre suficiente para ele acima desse endereço. Suponhamos, por exemplo, que o «tamanho» dado é de 4000, isto é, a rotina de código máquina requer 4000 *bytes* de espaço. No computador de 48 k, o RAMTOP é normalmente 65367, de modo que a instrução CLEAR 61367 (que é 65367−4000) permite carregar o código, fazendo LOAD "" CODE 61368. Para fazer o *Spectrum* reproduzir a frase, emprega-se RAND USR e o endereço a partir do qual o código foi carregado, que neste exemplo é 61368.

O princípio em que se baseia a secção de código máquina do programa de reprodução de frases é bastante simples, devendo ver-se a listagem assembler para uma análise mais minuciosa. A rotina de gravação monitoriza o *bit* 6 do porto FEh e mede o tempo com o registo B. Se o sinal de entrada da fita altera o seu estado (isto é, se passa de alto a baixo, ou vice-versa), ou se o registo B fica «cheio» e retorna a zero, o conteúdo de B é armazenado numa vasta tabela da RAM, o ponteiro é incrementado e o processo vai-se repetindo até a tabela estar completamente preenchida. É por isso que gravar silêncio em memória é muito mais eficiente do que gravar som.

A rotina de reprodução é mais ou menos o contrário da rotina de gravação. Vai buscar os dados à tabela e acciona o altifalante

através do *bit* 4 do porto FEh, ao ritmo imposto pelo fluxo de dados. A segunda rotina está escrita de modo a ser posicionada independentemente — ao gravar-se o código «carcereiro» volta a situar-se no início da secção de dados requerida.

Este programa é divertido apesar da má qualidade do som. No entanto, é possível identificar a voz da pessoa que gravou a frase, de modo que até se pode pedir a algum locutor célebre que grave uma frase para nós!

Listagem Assembler para o programa de reprodução sonora em código máquina.

Nota: os códigos *op* estão em hexadecimal.

Código	Rótulo	Instrução		Comentários
3E0F	LOAD	LD	A,0F	Limpar o porto.
D3FE		OUT	(FE),A	
F3		DI		Desligar as interrupções.
216480		LD	HL,8064	Apontar HL para o início da tabela de armazenamento.
1100F0		LD	DE,F000	Apontar DE p. o fim da tabela.
0E00		LD	C,00	Limpar C.
0600	NBYTE	LD	B,00	Estabelecer o contador em zero.
04	LISTN	INC	B	Aumentar a contagem em um.
2807		JR	Z,STORE	Se a contagem regressou a zero, saltar para STORE.
DBFE		IN	A,(FE)	Ler o porto FE para A.
E640		AND	40	Isolar o *bit* 4.
B9		CP	C	Se A coincidir c. C (mantendo
28F6		JR	Z,LISTN	o últ. valor ali lido) ir p. LISTN.
70	STORE	LD	(HL),B	Armazenar contagem na tabela.
4F		LD	C,A	Carregar C c. novo estado porto.
23		INC	HL	Apontar p. espaço seg. da tabela.
E5		PUSH	HL	Gravar HL na pilha.
A7		AND	A	Limpar a bandeira de transporte.
ED52		SBC	HL,DE	Comparar HL com DE e, se
7C		LD	A,H	forem iguais (se o ponteiro ti-
B5		OR	L	ver alcançado o fim da tabela), estabelecer a bandeira Z.
E1		POP	HL	Recuperar antigo valor de HL.

20E8		JR	NZ,NBYTE	Se a bandeira Z não estiver estabelecida, ir p. NBYTE.
FB		EI		Religar as interrupções.
C9		RET		Regressar ao BASIC.
210000	TEST	LD	HL,0000	Carregar HL c. compens. inicial.
09		ADD	HL,BC	Adicionar HL ao end. «RAND»
44		LD	B,H	mantido em BC e transferir o
4D		LD	C,L	resultado de volta a BC.
213700	REPLY	LD	HL,0037	Carregar HL com o comprimento de REPLY.
09		ADD	HL,BC	Apontar HL p. o início da tabela.
E5		PUSH	HL	Armazenar HL na pilha.
119C6F		LD	DE,6F9C	Carregar DE c. tamanho da tabela.
19		ADD	HL,DE	Apontar HL para o fim da tabela
EB		EX	DE,HL	e transferi-lo para DE.
E1		POP	HL	Restaurar HL.
3A485C		LD	A,(5C48)	Carregar A com o conteúdo de BORDCR.
CB3F		SRL	A	Dividir A por 8.
CB3F		SRL	A	
CB3F		SRL	A	
E607		AND	07	Desmascarar cinco *bits* altos.
4F		LD	C,A	Colocar a cor da moldura em C.
F3		DI		Desligar as interrupções.
46	OUTBY	LD	B,(HL)	Carregar B c. o *byte* da tabela.
04		INC	B	Testar B para ver se é zero.
05		DEC	B	
2804		JR	Z,LOOP	Se for zero, saltar para LOOP.
2F		CPL		Complementar o *bit* 4 de A.
E610		AND	10	
B1		OR	C	Adicionar a cor da moldura.
D3FE	LOOP	OUT	(FE),A	Injectar *bit* 4 de A no altifalante.
00		CP	(HL)	Retardar de modo a que a
00		NOP		velocidade de reprodução
00		NOP		acompanhe a rotina de gravação.
05		DEC	B	Decrementar o contador e, se
20F8		JR	NZ,LOOP	este não for zero, ir p. LOOP.
23		INC	HL	Apontar HL p. o *byte* seguinte.
E5		PUSH	HL	Armazenar HL.
A7		AND	A	Restabelecer bandeira de transp.
ED52		SBC	HL,DE	
47		LD	B,A	Armazenar A.

```
7C          LD      A,H         Se HL tiver alcançado o fim da
B5          OR      L           tabela, estabelecer bandeira zero.
E1          POP     HL          Restaurar HL.
78          LD      A,B         Restaurar A.
20E3        JR      NZ,OUTBY    Se a bandeira Z não estiver
                                estabelecida, ir para OUTBY.
FB          EI                  Ligar as interrupções
                                e regressar
C9          RET                 ao BASIC.
00---  TABLE                    Área de memória livre.
```

POSFÁCIO

Neste ponto, descrevemos já os aspectos de *hardware* e de características de marca do *Spectrum* da Sinclair. O serviço que toda esta informação nos pode oferecer depende das aplicações a dar ao computador.

Há quem conduza o seu automóvel sem se preocupar em abrir sequer o *capot*, enquanto outros preferem ter uma ideia de como o carro funciona, mesmo que não tencionem sujar as mãos a mexer no motor. Quem se considerar um potencial «mecânico» de computadores será obrigado a adquirir muita experiência de código máquina, para além de ter de se armar com imensa paciência. Há inúmeras obras dedicadas a todos os aspectos de *hardware* e de código máquina do *Z80*. Quanto a livros sobre paciência...

A seguir, nesta colecção:

Programas de Inteligência Artificial em Basic
de Monteil e Schomberg