Este livro põe à disposição do leitor
os elementos essenciais
da programação em código máquina
e desfaz a falsa ideia
de que esta linguagem não é para todos.

O código máquina é uma linguagem de computador,
exactamente como é o BASIC.
Quem conhece instruções de BASIC
compreenderá facilmente as instruções
do código máquina
e poderá vir a construir os seus programas
de forma muito avançada
até atingir a incrível velocidade
desta linguagem de alta resolução.

Não estamos perante uma obra para ler
e arrumar na estante.
A sua leitura será sempre compensadora
e demonstrará que o Spectrum
não serve só para vencer invasores espaciais.
Vale a pena saber dominá-lo a fundo!

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

BIBLIOTECA VERBO DE INFORMÁTICA

3

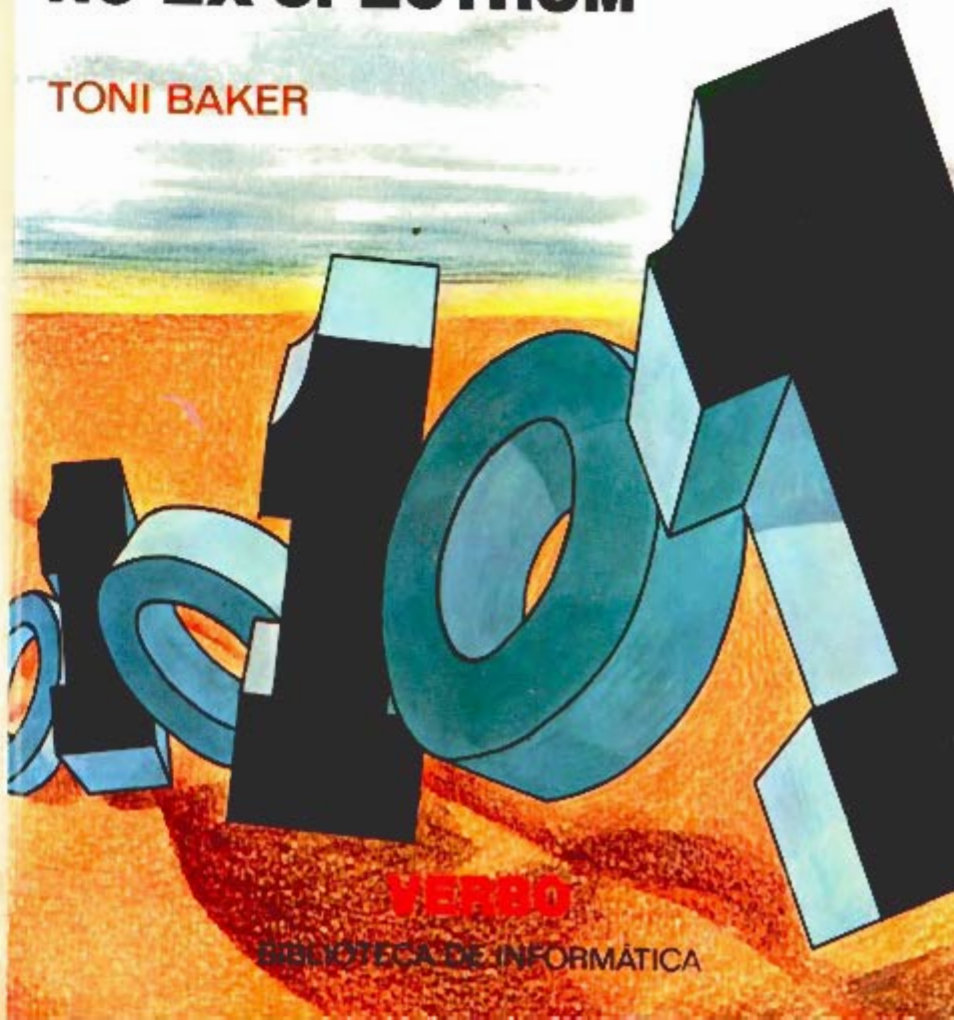
O DOMÍNIO DO CÓDIGO MÁQUINA T. BAKER

verbo

O DOMÍNIO DO CÓDIGO MÁQUINA

NO ZX SPECTRUM

TONI BAKER



VERBO

BIBLIOTECA DE INFORMÁTICA

TONI BAKER

O Domínio do Código Máquina no ZX Spectrum

Verbo

Título original: *Mastering Machine Code On your ZX Spectrum*

Tradução de Maria da Luz Martins
Revisão técnica de Miguel Martins
Capa de Luís Anglin

EDITORIAL VERBO. Lisboa/São Paulo
N.º Ed. 1595
Composto por Fotocompográfica
Impresso por Empresa Litográfica do Sul
em Abril de 1985
Depósito Legal n.º 7932/85

ÍNDICE

GLOSSÁRIO DE ABREVIATURAS

capit.	— capítulo	p.	— para
cm.	— comprimento	pr.	— próximo
cód.	— código	prim.	— primeiro
comp.	— computador	quad.	— quadrado
cont.	— contador	rest.	— restaura
coord.	— coordenada	segn.	— segundo
cr.	— carácter	seq.	— sequência
desl.	— deslocamento	sist.	— sistema
dig.	— dígitos	sup.	— superior
dir.	— direcção	tab.	— tabuleiro
endr.	— endereço	tds.	— todos
estr.	— estrutura	últ.	— último
gráfs.	— gráficos	unds.	— unidades
ints.	— interrupções	vars.	— variáveis
máq.	— máquina	vel.	— velocidade
marc.	— marcador	vf.	— verifica
mov.	— movimento	vfr.	— verificar
nec.	— necessário		

CAPÍTULO UM

Introdução

Este livro é dirigido a todos aqueles que têm conhecimentos razoáveis de BASIC, mas cujo conhecimento de código máquina é nulo. Começando pelas primeiras noções em programação BASIC, introduziremos gradualmente o conceito de uma sub-rotina em código máquina e desenvolvê-lo-emos no decurso do livro, até atingirmos programas completos unicamente em código máquina. Em pouco tempo o leitor verá aumentada a compreensão do código máquina, e dentro em breve começará a escrever os seus próprios programas e rotinas.

A linguagem máquina não é mais que uma segunda linguagem de computador, muito parecida até com o BASIC. Começaremos por aprender as instruções mais simples, familiarizando-nos e chegando a elas por meio de programas BASIC.

Em breve trataremos da escrita de cadeias *(strings)*, o que leva ao uso da sub-rotina PRINT na ROM. Mostraremos a rotina pela «impressão» de um tabuleiro de damas, que usaremos mais adiante.

Explicaremos o código máquina equivalente à função INKEY$ e usaremos a técnica de «leitura» *(scanning)* do teclado para escrever um programa que simula uma máquina de escrever, imprimindo caracteres que são versões aumentadas dos caracteres normais.

Usaremos a mesma técnica de *scanning* do teclado para produzir notas musicais de uma maneira surpreendente. A máquina produzirá duas oitavas, com que se poderá tocar, a partir do teclado, uma grande variedade de músicas simples.

O computador gerará muitas imagens variadas e fascinantes no programa LIFE, desafiando a habilidade para os jogos gráficos, tais como *spirals* e *breakout*. Também se inclui um programa de jogo de damas com várias características interessantes. Cada programa é explicado minuciosamente para se compreender o porquê de cada instrução.

O estudo pormenorizado da listagem destes programas ensina muito sobre código máquina, mas é claro que o essencial da aprendizagem virá da experimentação. Escreva o leitor os seus

próprios programas — ou adapte os nossos (que foram feitos com essa intenção); alguns, na realidade, foram deliberadamente escritos de uma maneira susceptível de melhoramento com esse mesmo propósito (no entanto, todos eles funcionarão satisfatoriamente tal como estão).

Para tirar o melhor partido deste livro é aconselhável estudá-lo do princípio ao fim e deve-se tentar alterar ou melhorar um programa quando isso for pedido. Não é difícil, uma vez que o livro progride muito lentamente, mas exige algum esforço mental. Os dois últimos capítulos são bastante ambiciosos. Apresentamos um programa através do qual a ROM pode ser «desassemblada» e dá-se a explicação das sub-rotinas aritméticas — e explicamos mesmo funções complexas como SIN e COS.

Os apêndices do final, muito condensados, são usados como fonte de referência através do livro. Em geral, encontrar-se-á qualquer informação necessária nestes apêndices ou no capítulo 8, que é uma espécie de dicionário de código máquina.

O verdadeiro primeiro capítulo começa na página seguinte, com uma introdução ao uso de «hexadecimal».

CAPÍTULO DOIS

Introdução ao hexadecimal e ao código máquina

O *ZX Spectrum* está ligado e pronto a funcionar e aquela pequena mensagem de *copyright* ou aquele K a piscar está à nossa frente à espera que introduzamos qualquer coisa. Que fazer em seguida?

O primeiro trabalho é preparar a máquina de maneira que aceite código máquina em vez de BASIC. Isto não é difícil mas, infelizmente, quando Sinclair idealizou a sua máquina, esqueceu-se de incluir um botão que dissesse «Mude para código máquina»; portanto, a maneira de o fazermos terá de ser através de um programa BASIC.

Antes de mais, temos de arranjar lugar na memória do computador onde possamos guardar o código máquina. Para isso, vamos escrever:

```
16K: CLEAR 28671
48K: CLEAR 61439
```

O efeito é bastante claro. A figura 2.1 é uma espécie de diagrama «antes e depois», mostrando o que se passa na parte superior da memória do *Spectrum*. (Talvez se deva comparar esta figura com o diagrama da página 165 do manual de instrução do *Spectrum*.) A área a partir do endereço 32600 (16 K) ou 65368 (48 K) até ao endereço 32767 (16 K) ou 65535 (48 K) é toda ela usada pelo computador, mas a área abaixo, quer dizer, à esquerda, no diagrama, está toda livre até à zona da máquina usada pelo BASIC. Podemos mudar este endereço usando CLEAR, que move toda a informação usada pelo BASIC de maneira que o último endereço susceptível de ser usado pelo BASIC é o número a seguir à palavra CLEAR. Nos exemplos acima dados, CLEAR 28671 faria de 28671 o último endereço utilizável por BASIC, por isso 28672 (o endereço seguinte) seria o primeiro endereço não usado pelo BASIC — e portanto livre para ser utilizado por nós para guardar código máquina. Se tivéssemos introduzido CLEAR 29999, poderíamos usar os endereços 30000 e seguintes.

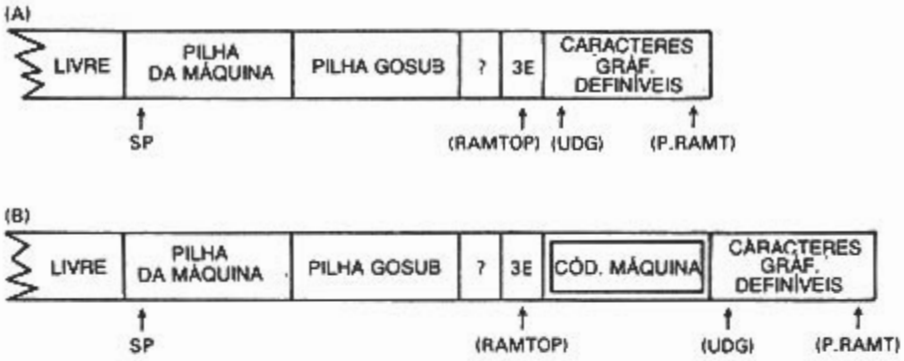


Figura 2.1

Podemos começar. Introduzamos o seguinte programa BASIC:

```
10 LET X = 28672 ou 61440 conforme se trate de 16 K ou de 48 K
20 LET A$ = ""
30 IF A$ = "" THEN INPUT A$
40 LET Y = CODE A$ - 48:IF Y> 9 THEN LET Y = Y - 7
50 LET Z = CODE A$(2) - 48: IF Z> 9 THEN LET Z = Z - 7
60 POKE X,16*Y + Z
70 LET X = X + 1
80 LET A$ = A$(3 TO )
90 GO TO 30
```

Compreende-se como o programa funciona? Ou pelo menos o que é que ele faz? Em resumo, ele aceita um programa em código máquina e guarda-o no endereço 28672/61440 e seguintes. Para sair do programa é preciso introduzir EDIT (*shift* 1) para apagar as aspas e então STOP (*shift* A) (sem aspas), antes de introduzir ENTER. Isto pára o programa, com a indicação H STOP em INPUT. Repare-se que, apesar de permitir introduzir código máquina, este programa não tentará corrê-lo.

Explicaremos agora o que é código máquina. O código máquina, ou linguagem máquina, como é também conhecido, é uma outra linguagem de computador muito semelhante ao BASIC mas a um

nível muito baixo, o que quer dizer que certas instruções ligeiramente complicadas (tais como FOR-NEXT, por exemplo) nem sequer existem. No entanto este motivo leva também a que seja uma língua muito fácil de aprender. Tal como o BASIC, consiste num conjunto de instruções, cada qual ordenando ao computador a realização de uma tarefa diferente e bastante específica. Uma dessas instruções, a RET, é mais ou menos equivalente ao RETURN do BASIC.

No entanto, ao contrário do BASIC, o computador não sabe ler as instruções se elas estiverem em «inglês». Não se pode correr o programa BASIC anterior e introduzir RET porque o *Spectrum* não o compreende. Para facilitar o trabalho à sobrecarregada máquina, cada instrução em linguagem máquina tem o seu próprio código numérico e estes códigos são directamente compreendidos. Por exemplo, o código para RET é 201. Todos os códigos se encontram entre zero e 255 e é muito mais conveniente escrever estes códigos num sistema chamado hexadecimal.

NUMERAÇÃO HEXADECIMAL

Em resumo, hexadecimal, ou hex, para encurtar, é um sistema de numeração que usa dezasseis símbolos em vez de dez. Os primeiros dez são os mesmos a que estamos habituados. São eles:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Mas há mais seis símbolos que representam os números dez a quinze e que são:

A, B, C, D, E, F.

As coisas começam a complicar-se quando queremos representar números maiores que quinze, porque, acredite-se ou não, dezasseis escreve-se 10! Pior ainda, dezassete é 11 e dezoito, 12. Assim se continua até vinte e cinco, que se escreve 19, e quando chegamos a vinte e seis começamos a utilizar os novos símbolos mais uma vez: vinte e seis escreve-se 1A e assim por diante.

No final do livro encontra-se uma tabela, o Apêndice 1. O primeiro conjunto de números é uma tabela completa de todos os

números de zero a duzentos e cinquenta e cinco. Destina-se a ajudar a compreender o sistema hexadecimal de numeração. Deve-se tentar usar a tabela o menos possível, mas não nos preocupemos se a princípio for necessário recorrer a ela a todo o momento, pois a habituação ao sistema não demorará.

Os símbolos do lado esquerdo no sentido descendente são o primeiro dígito hexadecimal; os símbolos na horizontal na parte de cima são o segundo. Claro que os zeros situados à esquerda podem ser omitidos, se aparecerem, mas é em geral mais conveniente representar os códigos hex com dois dígitos em vez de um só (por exemplo, escrever Ø7 em vez de 7).

Se houver confusão sobre se um número está ou não escrito em hex, deve-se torná-lo claro, escrevendo um pequeno "h" (para hex) ou um "d" (para decimal) à direita do número. Assim 19d quer dizer dezanove, enquanto 19h é na realidade vinte e cinco. Na maior parte dos casos, no entanto, não haverá confusão. Números como CD, por exemplo, não podem ser decimais; por isso não há necessidade de escrever CDh (CDd não faz sentido!). Contrariamente, por convenção, os números hex são geralmente escritos com um número par de dígitos, de maneira que 118 (que tem um número ímpar de dígitos) quer dizer 118d. 118h seria escrito Ø118.

Conhecer os fundamentos da numeração em hex é essencial no que respeita ao código máquina, e por isso será natural voltar a consultar esta secção ou recorrer à tabela do Apêndice 1 — é para isso que ela serve.

Outra nota: 19h diz-se «um nove» ou «um nove hex». Nunca se deve dizer «dezanove» ou «dezanove hex».

Existem algumas diferenças fundamentais entre código máquina e BASIC. Uma delas é a dos números de linha.

Como se sabe, cada linha de programa BASIC deve ser precedida por um número de linha, de maneira que o computador possa saber em que ordem o executará. Se não tiver sido digitado qualquer número de linha, o computador interpretará a instrução como um comando e executá-la-á imediatamente.

Em código máquina não existem números de linha. O *Spectrum* não permitirá usar instruções em código máquina como comandos; têm de fazer parte de um programa. As instruções serão executadas pela ordem em que foram guardadas na memória. Por exemplo, se o

computador tiver acabado de executar a instrução guardada no endereço 3ØØØØ, ele irá seguidamente executar a que estiver guardada no endereço 3ØØØ1. E continuará assim até receber instruções que lhe indiquem para proceder de outra maneira.

Ao contrário do BASIC, ele não fará STOP quando chegar ao fim do programa. Não lhe reconhecendo o fim, de cada vez que encontrar um número diferente de zero tratá-lo-á como um código para alguma instrução e tentará executá-la. Daqui resultará em geral aquilo a que chamamos *crash*.

Note-se que mesmo o zero é código de uma instrução: a instrução NOP.

CRASH

Crash, que em português se pode traduzir por «falha» ou «quebra», é o que acontece quando o *Sinclair* insiste em executar qualquer coisa que não devia ou qualquer coisa que não podia fazer. Mesmo um pequeno erro no código pode confundir a máquina. Quando a máquina entra em quebra, o teclado geralmente deixa de funcionar. Umas vezes executará NEW, outras o *écran* «congelará», por vezes ficará branco, mas, mais ainda: poderá aparecer no *écran* «um quadro qualquer em estilo moderno». Se isto acontecer logo saberá tudo a esse respeito.

É vulgar tentar sair desse estado de coisas usando a tecla de BREAK; mas ver-se-á então que a tecla não funciona. De facto, em geral, nenhuma das teclas funcionará. A principal razão por que não gostamos das quebras é que a única maneira de voltar ao normal é desligar a alimentação do computador. Ao voltar-se a ligar, perdeu-se todo o programa e será necessário introduzi-lo outra vez. É esta a dificuldade de usar código máquina; se um programa BASIC contiver um erro, ele simplesmente não funcionará, mas se um programa em código máquina contiver um erro, muito possivelmente entrará em quebra.

COMO EVITAR AS FALHAS

A única maneira de evitar a falha (ou quebra) é ter a certeza de que os programas não contêm erros: Já dissemos que a execução

não parará automaticamente no fim do programa; assim, ele precisará que lho digam com uma instrução específica; essa instrução é RET (RETorne ao BASIC).

Este capítulo tratou da maneira de reservar espaço para programas em código máquina e ofereceu também um programa para carregá-lo. Mas não ensinou como utilizar este programa, nem como fazer correr programas em código máquina, uma vez carregados. Foram apresentados os fundamentos da numeração hex assim como a noção de quebra *(crash)*.

Uma vez compreendido este capítulo, pode-se passar ao capítulo três para a primeira lição de programação em código máquina.

CAPÍTULO TRÊS

Aritmética simples

É agora altura de voltar ao programa que no último capítulo fizemos introduzir. A não ser que se tenha feito algum disparate, como desligar a máquina (nesse caso será necessário escrevê-lo outra vez), o programa ainda está na memória. Deve-se guardá-lo em *cassette* com o nome «HEXLD» (SAVE «HEXLD»). Digite-se CLEAR 28671 (16 K) ou CLEAR 61439 (48 K), se ainda não foi feito, e corra-se o programa (RUN).

O programa fica agora à espera de receber uma cadeia *(string)* — um tipo especial de cadeia. O que ele espera receber é um número hexadecimal. Sempre que se queira introduzir uma instrução em linguagem máquina, terá de se conhecer o respectivo código numérico e em hexadecimal.

O código para RET é, como já dissemos, 2Ø1. O que é isto em hexadecimal? Dividindo-o por dezasseis, obtém-se doze, resto nove. O código hexadecimal de doze é C e o de nove é 9. No Apêndice 1, ver-se-á que 2Ø1 se escreve C9 em hexadecimal. Será coincidência?

Ao introduzir «C9», indica-se ao computador que a primeira instrução do programa em linguagem máquina é RET. Interrompe-se o programa, apagando as aspas e introduzindo STOP. O programa em linguagem máquina está completo. Contém apenas uma instrução: RET. Isto escreve-se usualmente:

```
C9          RET
```

Para nos lembrarmos que o código hex para RET é C9. Por vezes, às instruções em linguagem máquina chama-se *opcodes* (códigos de operação) para as distinguir dos correspondentes códigos hexadecimais, *hexcodes*. A máquina usa códigos hexadecimais, pois não reconhece os *opcodes*.

Pelo contrário, nós, os homens, usamos os *opcodes*, uma vez que nos seria difícil pensar utilizando *hexcodes*.

Olhando para o *écran*, vê-se a mensagem H STOP em INPUT. A

máquina volta ao modo de comando e está à espera de uma instrução. Suponhamos que queremos correr o programa em código máquina que acabamos de introduzir. Podemos fazê-lo, quer a partir de um programa, quer a partir de um comando directo, como vamos fazer. Uma vez que o programa foi carregado no endereço 28672 (16 K) ou 61440 (48 K) o comando será:

```
16K: PRINT USR 28672
48K: PRINT USR 61440
```

O computador escreverá no *écran* 28672 ou 61440. Na realidade bem poderia ter escrito apenas PRINT 28672 ou PRINT 61440. Mas pense-se no que aconteceu. Deu-se um número ao computador, o número a seguir à palavra USR. O computador, a partir daí, executou um programa em código máquina com a única instrução RET, isto é, não fez nada. Como não fez nada, não surpreende que o número que se lhe deu chegue ao fim sem ter sido modificado e tenha assim sido escrito inalterado (pelo BASIC, uma vez que lhe mandamos escrever qualquer coisa). Mas não se desanime com este exercício. Já se usou a instrução USR e decerto se compreendeu por que motivo ela fez aquilo que fez. Correu-se um programa em código máquina, embora simples, sem ter perdido o controle *(crash)* e sabendo como ele funciona. Experimente-se outras maneiras de usar USR mas com o mesmo número a seguir. Por exemplo: LET X=USR 61440 ou PRINT PEEK USR 61440, etc.

Antes de podermos continuar a aprender mais instruções, vamos parar um pouco e explorar o conceito de registo. Um registo é semelhante a uma variável, já que tem um nome — geralmente uma letra do alfabeto — e pode guardar números da mesma forma que as variáveis BASIC. A grande diferença é que os registos só podem guardar números entre 00 e FF (ou, em decimal, entre 0 e 255).

Há sete registos facilmente utilizáveis. Os seus nomes são: A, B, C, D, E, H e L. Note-se que como o número maior que qualquer deles pode guardar é FF, na realidade um registo só pode guardar números de dois dígitos hexadecimais. Isto é bastante mais fácil de lembrar do que «um número entre 0 e 255».

Para permitir maior flexibilidade, usam-se alguns dos registos aos pares. B e C, por exemplo, podem ser usados como um par. Neste caso referimos o par como BC e tratamo-lo como um todo

único. Uma vez que B guarda dois dígitos hexadecimais e C também guarda dois dígitos, BC como par guardará quatro dígitos hexadecimais. Se B contém 4A e C contém 23h, então diremos que BC contém 4A23.

Podemos também fazer o mesmo em decimal, mas isto é muito mais difícil, e por esse motivo preferimos representar tudo em hexadecimal. Vamos usar exactamente os mesmos números. Se B contém 74d e C contém 35d, então diremos que BC contém 18979. Repare-se que em decimal não podemos fazer facilmente os cálculos de cabeça; não se trata apenas de justapor os dígitos como fizemos em hex; não podemos combinar 74d e 35d e obter 7435d; infelizmente isso não funciona, já que o resultado está errado. Em decimal precisamos de uma calculadora (um *Spectrum* geralmente serve) para calcular 256 vezes o primeiro número mais o segundo número; isto é, 256*74+35=18979. Não é muito difícil de compreender. Quando se começa a usar código máquina, contar em hexadecimal simplifica muito as coisas.

Há três pares de registos possíveis: são eles BC, DE e HL.

A INSTRUÇÃO LD

Considere-se a instrução BASIC LET A=42. Em linguagem máquina atribuímos valores às nossas variáveis (registos) usando a instrução LD. Podemos, por exemplo, escrever LD A,2A. Repare-se que não existe sinal de igual, ao contrário do BASIC. Em vez disso, utilizamos uma vírgula para separar o A do número. O resultado desta expressão é precisamente aquele que se poderia esperar, o valor anterior de A é substituído por um novo valor (neste caso 2A — decimal 42).

Cada instrução LD diferente tem um código diferente. Por exemplo, o código para LD A é 3E. Isto quer dizer que a instrução LD A,2A pode ser escrita em hexadecimal como 3E2A. Note-se que esta instrução tem o comprimento de dois *bytes* (um *byte* são dois dígitos hex) enquanto que RET ocupava um *byte* (o *byte* 69). A instrução BASIC LET A=42 ocupa onze *bytes*, mais quatro para o número de linha ou mais um para os dois pontos, se não for a primeira instrução de linha. Portanto o código máquina representa uma redução de espaço significativa.

Os restantes códigos são os seguintes:

```
LD A, 3E
LD B, Ø6      LD BC, Ø1
LD C, ØE
LD D, 16      LD DE, 11
LD E, 1E
LD H, 26      LD HL, 21
LD L, 2E
```

Usando o programa «HEXLD», faça o leitor entrar o seguinte programa introduzindo os símbolos da coluna da esquerda. (Os símbolos da coluna da direita são apenas para uso dos humanos — o computador não os reconhece.) Introduza "Ø6ØØ", a seguir "ØE2A", depois "C9" e então faça *break*.

```
ØóØØ      LD B,ØØ
ØE2A      LD C,2A
C9        RET
```

O programa irá carregar B com zero e C com 2A, pelo que BC ficará com o valor ØØ2A. Em decimal, BC conterá 256*Ø+42. Agora, digitando PRINT USR 28672 (16 K) ou PRINT USR 6144Ø (48 K), que valor se obterá? A resposta é −42. Esperar-se-ia por isto? Passemos agora a outra coisa: introduzamos este programa:

```
16ØØ      LD D,ØØ
1E2A      LD E,2A
C9        RET
```

Agora, ao escrever PRINT USR 28672 ou 6144Ø, acontece algo diferente.

Eis o que se passa quando o computador encontra a frase «USR qualquer coisa» em qualquer linha BASIC: primeiro, o número «qualquer coisa» é guardado no par de registos BC. (O número na linha BASIC estará em decimal, mas pode-se passá-lo para hex, querendo.) Então o controle passará para o endereço «qualquer coisa» e será executado qualquer código máquina que aí esteja. Ao encontrar a instrução RET o controle voltará ao BASIC, que se lembrará do número que retomou em BC. É perfeitamente possível, então, ter uma linha como LET A=USR 2ØØØØ + USR 3ØØØØ que correrá dois programas em código máquina, primeiro o que se

encontra em 2ØØØØ e depois o que se encontra em 3ØØØØ. À variável A será atribuído um valor igual à soma de duas quantidades diferentes: o valor retomado em BC pelo primeiro programa mais o valor retomado em BC pelo segundo programa. Note-se que o valor final de DE e HL (e mesmo de A) não será guardado. Por falta de melhor palavra diremos que eles foram «esquecidos».

Diremos ainda que HL simboliza HIGH/LOW (alto/baixo). Quando HL é tratado como um par, chama-se o H «parte alta» e o L «parte baixa». BC e DE também têm partes altas e baixas: a primeira letra é a parte alta e a segunda letra é a parte baixa.

Já sabemos que quarenta e dois em hex com quatro dígitos é ØØ2A. Que representará o seguinte programa?

```
Ø1ØØ2A      Lembre-se que Ø1 significa LD BC,....
C9          RET
```

Talvez seja surpreendente descobrir que ao fazer PRINT USR 6144Ø se obtém na realidade 1Ø752 e não 42. Ao correr agora este programa:

```
Ø12AØØ      Veja-se que ØØ e 2A foram trocados.
C9          RET
```

obtém-se quarenta e dois. O ØØ e o 2A precisam de ser trocados de maneira a estarem na «ordem errada». Apesar de isto ser bastante estranho, é na realidade usual em código máquina. Em todas as instruções que trabalham com números de quatro dígitos, trocam-se os dois *bytes* de maneira que a parte baixa apareça em primeiro lugar e a parte alta em segundo. Esta é, de facto, a maneira pela qual os números são guardados pelo BASIC nas variáveis de sistema que têm o comprimento de dois *bytes*. Na instrução LD HL, o primeiro *byte* é sempre 21h, o segundo é o novo valor de L e o terceiro *byte* é o novo valor de H. Note-se que esta instrução tem sempre o comprimento de três *bytes*.

TRANSFERÊNCIA DE UMA VARIÁVEL A OUTRA

Se estivéssemos constrangidos em BASIC a usar apenas instruções LET da forma LET A= a um número, estaríamos um pouco

limitados. Precisamos de mais flexibilidade. Por exemplo, qualquer coisa como LET A=B seria muito útil. Bem, nós podemos certamente fazer isso em código máquina. Os códigos são apresentados na figura 3.1

LD	A	B	C	D	E	H	L
A	7F	78	79	7A	7B	7C	7D
B	47	40	41	42	43	44	45
C	4F	48	49	4A	4B	4C	4D
D	57	50	51	52	53	54	55
E	5F	58	59	5A	5B	5C	5D
H	67	60	61	62	63	64	65
L	6F	68	69	6A	6B	6D	6E

Figura 3.1

Nesta tabela lemos os registos da coluna da esquerda em primeiro lugar e os registos da coluna horizontal superior em segundo, de maneira que o código para LD A,D é 7A. Repare-se que cada um destes registos tem o comprimento de apenas um *byte*. Compare-se isto com a instrução equivalente em BASIC LET A=D, que ocupa quatro *bytes* e mais quaisquer necessários para o número de linhas, dois pontos e/ou fim de linha (ENTER com *carriage return*).

E agora alguma aritmética simples. Se algum dos leitores pensou já com um certo avanço, possivelmente perguntará como se somarão e subtrairão registos em código máquina. Afinal, se LD A,B ocupa apenas um *byte*, como poderemos encontrar equivalentes para LET A=B+C?

De facto, utilizamos uma instrução totalmente diferente. A instrução é ADD. Um exemplo útil seria ADD HL,DE, que tem como efeito LET HL=HL+DE. O conteúdo do par de registos HL é somado ao conteúdo do par de registos DE e o resultado desta operação é guardado em HL, substituindo o seu valor anterior.

Na verdade podemos fazer aritmética em hex no papel. Vejamos como estas somas funcionam. Tal como em decimal, somamos

primeiro as unidades, mas somente quando o resultado ultrapassar quinze é que se diz «e vai um».

```
 1234      1234      5678      89AB
+1234     +5678     +9ABC     +963Ø
------    ------    ------    ------
 2468      68AC      F134      11FDB
```

No primeiro exemplo não existem surpresas. No segundo temos 4+8=C e 3+7=A. No terceiro exemplo precisamos ter em conta o que vai *(carry)*: 8+C=14h; isto quer dizer 4 e vai 1. 7+B+1 é 13h, ou 3 e vai 1. 6+A+1 é 11h e vai 1. 5+9+1 é F e não vai nada. Se se compreendeu isto, o quarto exemplo não trará preocupações, mas repare-se num problema: 11FDB é demasiado grande para ser guardado em dois registos, porque tem demasiados dígitos. Supondo que tentamos executar ADD HL,DE, sendo HL inicialmente 89AB e DE 963Ø, que irá acontecer?

Antes de nos debruçarmos sobre isso, vamos demonstrá-lo com um exemplo mais fácil. Introduza o leitor, e faça correr, este programa e tente descobrir o resultado antes de o ver.

```
3EFF      LD A,FF (em decimal 255)
Ø6Ø1      LD B,Ø1
8Ø        ADD A,B (note-se que FF+Ø1=1ØØ em hex)
Ø6ØØ      LD B,ØØ
4F        LD C,A
C9        RET
```

Recordemos que só o valor final de BC volta ao BASIC, de maneira que as últimas instruções apenas transferem a resposta que calculámos de A para BC, pois não fazem parte do cálculo. Vejamos o que o programa faz: tenta somar um a duzentos e cinquenta e cinco, mas tenta fazê-lo num registo que só pode guardar números até 255 e não mais. O que irá acontecer? Corre-se o programa e logo se vê (basta escrever PRINT USR 28672 ou PRINT USR 6144Ø).

Obter-se-á zero como resposta. FF+Ø1 é de facto 1ØØh mas somente os dois últimos dígitos são guardados em A.

Para responder à nossa pergunta anterior, 89AB+963Øh podem

somar realmente 11FDB, mas se tentarmos calcular esta soma em HL apenas restarão os últimos quatro dígitos: 1FDB.

Há somente dois registos aos quais podemos adicionar valores: são eles A e HL. Não se pode, por exemplo, fazer ADD B,C! Repare-se também que a A podem ser adicionados registos simples e que apenas registos duplos podem ser adicionados a HL.

As possíveis instruções são, pois, as que se encontram na lista abaixo. São também dados os seus códigos hex.

ADD HL,BC 09	ADD A,A 87
ADD HL,DE 19	ADD A,B 80
ADD HL,HL 29	ADD A,C 81
	ADD A,D 82
	ADD A,E 83
	ADD A,H 84
	ADD A,L 85

Uma vez que temos estado preocupados com o que acontece quando a resposta é demasiado grande, gostaríamos de apresentar um novo tipo de «variável» em código máquina — um *flag*. Enquanto que um registo pode guardar números entre ØØ e FF, um *flag* é ainda mais restrito, uma vez pode apenas guardar um dos dois valores possíveis, ou zero ou um.

Um desses *flags* é chamado *carry*. Tem uma função muito especial. Sempre que se executa uma instrução ADD atribui-se um valor a *carry*. O seu novo valor é sempre «o terceiro dígito» de uma soma de dois dígitos ou o «quinto dígito» de uma soma de quatro dígitos. Se retrocedermos até aos quatro exemplos de somas em hex veremos que acontecerá o seguinte:

1234 + 1234 = 2468	CARRY = Ø, HL = 2468
1234 + 5678 = 68AC	CARRY = Ø, HL = 68AC
5678 + 9ABC = F134	CARRY = Ø, HL = F134
89AB + 9630 = 11FDB	CARRY = 1, HL = 1FDB

Vê-se, pois, que *carry* se torna Ø sempre que a soma não tem complicações, mas torna-se 1 sempre que a resposta seja demasiado grande para caber no registo indicado.

Duas palavras especiais acompanham os *flags*. *Set* quer dizer «vale um» e *reset* «vale zero». Assim um *flag* pode ser *set* ou *set* a

um, ou *reset* ou ainda *reset* a zero. É comum ouvir-se dizer «*set* a zero» ou «*reset* a um», etc. Estas expressões não devem ser usadas porque na realidade não fazem sentido.

Há uma outra instrução que podemos usar em vez de ADD. É a instrução ADC, que significa «ADD» com *carry*.

Funciona assim: se a máquina encontra a instrução ADC A,B, tomará o conteúdo do registo B e somá-lo-á ao conteúdo do registo A e então somará ao resultado o valor anterior de *carry*. O resultado desta soma ficará em A e o *carry* será *set* ou *reset* conforme for necessário (dependendo, é claro, de o resultado ser maior que FF).

Assim, ADD A,B realmente significa
LET A = os últimos dois dígitos de A+B
LET CARRY = Ø se A + B ⩽ FF
1 se A + B > FF

Enquanto ADC A, B significa
LET A = os dois últimos dígitos A + B + CARRY
LET CARRY = Ø se A + B + CARRY ⩽ FF
1 se A + B + CARRY > FF

Estudem-se os programas a seguir. Se o valor final do registo A não for importante, estes programas serão equivalentes (quer dizer que ambos fazem a mesma coisa). Porquê?

O primeiro programa é:

118533	LD DE,3385h
21C77B	LD HL,7BC7
19	ADD HL,DE
44	LD B,H
4D	LD C,L
C9	RET

O segundo é:

118533	LD DE,3385h
21C77B	LD HL,7BC7
7D	LD A,L
83	ADD A,E
6F	LD L,A

```
7C      LD A,H
8A      ADC A,D
67      LD H,A
44      LD B,H
4D      LD C,L
C9      RET
```

O efeito de ambos os programas é o mesmo. Daqui se podem depreender duas coisas. Primeiro, a instrução LD não afecta ou altera de qualquer maneira o valor de *carry*, porque se o fizesse as suas instruções entre ADD A,E e ADC A,D iriam estragar tudo. Em segundo, a instrução ADD HL,DE é muito mais curta (e também muito mais clara) do que se se andar às voltas somando os *bytes* separadamente. E nunca devemos esquecer de alterar a ordem dos dois *bytes* nas instruções LD pares de registos.

Se correrrmos ambos os programas verificaremos que fazem o mesmo. Que aconteceria se ADC A,D, no segundo programa, fosse substituído por ADD A,D?

Compreendida a diferença entre ADD e ADC, vamos continuar a estudar outras maneiras de somar. Primeiro, no entanto, aqui estão os códigos para ADC:

```
ADC HL,BC  ED4A        ADC A,A  8F
ADC HL,DE  ED5A        ADC A,B  88
ADC HL,HL  ED6A        ADC A,C  89
                       ADC A,D  8A
                       ADC A,E  8B
                       ADC A,H  8C
                       ADC A,L  8D
```

Repare-se que os códigos para ADC HL têm todos o comprimento de dois *bytes* em vez de um só. O primeiro *byte* é sempre ED e o segundo *byte* depende daquilo que se estiver a somar. Não se pense no entanto que ED tem o significado de ADC HL, uma vez que ED pode significar muitas outras coisas, dependendo sempre daquilo que o seguir.

SOMA DE CONSTANTES

Podemos também empregar ADD e ADC para somar constantes numéricas ao registo A (mas não a HL). Um exemplo pode ser

ADD A,3 que, como é natural, soma três ao valor actual de A. Atribui também a *carry* o valor um ou zero, dependendo do resultado ultrapassar ou não FF. (Lembremo-nos que FF é o número 255 em decimal.)

O código para ADD A é C6 e o código para ADC A é CE. ADD A,3 escreve-se pois em hex como C6Ø3. Quando se somam constantes são sempre necessários dois *bytes*. Se adicionássemos ØØ3A a HL, poderia ser assim:

```
113AØØ    LD DE,ØØ3A
19        ADD HL,DE
```

Mas este método tem a desvantagem de utilizar o registo DE, apagando o seu valor anterior. Outra maneira de se conseguir a mesma coisa, mas desta vez destruindo apenas o valor do registo A:

```
7D        LD A,L
C63A      ADD A,3A
6F        LD L,A
7C        LD A,H
CEØØ      ADC A,ØØ
67        LD H,A
```

Aqui usou-se a instrução ADC A,Ø para somar qualquer *carry* proveniente da adição de 3A a L.

E POR FIM (NO QUE RESPEITA A SOMAS)...

Há ainda uma última maneira pela qual podemos somar constantes a um registo: utilizando a instrução INC. INC A significa somar um ao conteúdo do registo A. INC D significa somar um ao registo D. INC HL significa somar um ao valor de HL e assim por diante.

No entanto, ao contrário de qualquer das instruções para somas que aprendemos até agora, INC é única pelo facto de não alterar o valor de *carry*. Se *carry* era zero antes de uma instrução INC, continuará zero. Se *carry* era um antes de uma instrução INC, continuará um. Esta regra aplica-se mesmo se INC «incrementa» por assim dizer A de FF a ØØ. Isto é pois uma diferença fundamental entre ADD A, Ø1 e INC A. Os códigos INC são:

INC BC Ø3	INC A 3C
INC DE 13	INC B Ø4
INC HL 23	INC C ØC
	INC D 14
	INC E 1C
	INC H 24
	INC L 2C

Lembremo-nos de que a diferença entre ADD A,Ø1 e INC A é que ADD A,Ø1 atribuirá um valor novo a *carry*, mas INC A deixá-lo-á sem alterações. INC é abreviatura de «incrementar».

O valor de *carry* pode ser alterado directamente sem que nenhum dos outros registos seja alterado. Há uma instrução chamada SCF (que quer dizer *Set Carry Flag*) que tem por função atribuir a *carry* o valor um. Não muda o valor de nenhum dos registos. Pode também fazer *carry* igual a zero, com um pouco de engenho, utilizando a instrução ADD A,Ø (é um ponto a meditar). Por fim há uma outra maneira de mudar o valor de *carry*. Uma instrução chamada CCF (ou *Complement Carry Flag*) é utilizada para mudar o valor de *carry*, seja de zero para um ou de um para zero, sem mudar os registos. Os códigos para estas instruções são:

37	SCF	"LET CARRY = 1"
C6ØØ	ADD A,Ø	"LET CARRY = Ø"
3F	CCF	"LET CARRY = 1 – CARRY"

SUBTRACÇÃO

Em linguagem máquina existem códigos para subtracção tal como para a adição. A instrução tem o nome de SUB, que é abreviatura de SUBmarino. (Isto é brincadeira!) No entanto, o único registo do qual se pode subtrair qualquer coisa é A — não se pode subtrair de HL. Por isso a instrução SUB A,B escreve-se geralmente, apenas, SUB B. O A é subentendido, presume-se que já lá se encontre. Exactamente como com ADD, existe também uma instrução chamada SBC, que significa SuBtrair com *Carry*. SBC, no entanto, pode ser utilizada em HL e assim o A tem de ser escrito. (Confuso, não?)

Isto funciona assim: SUB B tomará o valor do registo B e subtraí-lo-á ao valor de A. Se o valor inicial de A for maior ou igual ao de B, a subtracção será efectuada como se poderia esperar e o *flag carry* será *reset*. Isto é, tornar-se-á igual a zero. Se A, no entanto, contém um valor inicial menor que o de B, pressupõe-se que A representa um número que é Ø1ØØ (hex) maior que o valor actual de A — neste caso o *carry* será *set*. Isto é, tornar-se-á um. Assim, se A contém Ø3 e B contém Ø5, a operação que será executada não é Ø3−Ø5 (que seria negativa) mas 103−05, ou FE com *flag carry set*.

As instruções permitidas são:

SUB A	97
SUB B	9Ø
SUB C	91
SUB D	92
SUB E	93
SUB H	94
SUB L	95
SUB número	D6 seguido desse número

Note-se que não existem instruções para subtrair pares de registos.

SuBtrair com *Carry* (SBC), por outro lado, pode trabalhar com pares de registos, mas apenas com o par de registos HL.

SBC A,C por exemplo, subtrairá primeiro o valor de C do valor de A e em seguida o valor inicial de *carry* deste resultado. *Carry* será então redefinido da mesma forma que para SUB. Os códigos para SBC são:

SBC HL,BC ED42	SBC A,A 9F
SBC HL,DE ED52	SBC A,B 98
SBC HL,HL ED62	SBC A,C 99
	SBC A,D 9A
	SBC A,E 9B
	SBC A,H 9C
	SBC A,L 9D
	SBC A, número DE seguido desse número

DEC é a abreviatura de *Decrement* (decremento). É, como já se deve ter percebido, o oposto de INC. Tem por função reduzir qualquer registo de uma unidade sem alterar o *flag carry*, de maneira que DEC DE tem o efeito LET DE=DE−1.

E agora um pequeno exercício: uma das duas seguintes rotinas irá subtrair DE de HL e obterá uma resposta sempre correcta — a outra não será tão exacta. Qual delas será a correcta, qual a errada e porquê?

A primeira rotina é:

```
C600        ADD A,0
D602        SUB 2
ED52        SBC HL,DE
```

A segunda rotina é:

```
C600        ADD A,0
3D          DEC A
3D          DEC A
ED52        SBC HL,DE
```

De facto, o primeiro exemplo está errado. A instrução SBC HL,DE subtrairá tanto DE como o *flag carry* e por isso temos de nos certificar que este *flag* é zero. É para isto que a instrução ADD A,0 serve. No entanto, depois disto feito, o programa do primeiro exemplo irá alterar o *carry* outra vez através da instrução SUB 2. É descurada a hipótese de que se se subtrair dois de A podem surgir problemas de excesso; assim, se A iguala 01 ou 00, SUB não só mudará A para FF ou FE como também mudará o *carry* de zero para um, de maneira que o efeito de SBC HL,DE seria deixar em HL o valor de HL−DE−1 e não HL−DE. No segundo exemplo, a instrução DEC A é usada duas vezes. DEC não irá mudar o *carry*, de modo que ele terá ainda zero quando chegar à instrução SBC HL,DE e a subtracção decorrerá então correctamente.

Temos, pois, que INC e DEC não alteram o valor do *flag carry* — mas as outras instruções aritméticas fazem-no. As outras instruções já estudadas, RET e LD, não alteram o valor de *carry*.


```
DEC BC 0B        DEC A 3D
DEC DE 1B        DEC B 05
DEC HL 2B        DEC C 0D
                 DEC D 15
                 DEC E 1D
                 DEC H 25
                 DEC L 2D
```

Neste capítulo estudámos a maneira de carregar programas em código máquina e de como corrê-los. Explicámos o uso das instruções RET e LD e apresentámos as instruções aritméticas ADD, ADC, SUB, SBC, juntamente com INC e DEC. Explicámos a função do *flag carry* e mencionámos as instruções SCF *(Set Carry Flag)* e CCF *(Complement Carry Flag)*.

Não é preciso lembrar os códigos hex que o computador usa — nem mesmo os peritos se preocupam com isso! Todos os códigos se encontram num apêndice no final do livro. Só é necessário saber os nomes que lhes estão atribuídos — os *opcodes* e aquilo que eles fazem.

Antes de passar ao capítulo quatro, o interessado deve tentar fazer alguns dos exercícios seguintes. Se parecer difícil apenas será necessário fazê-los calmamente e pensar com clareza.

Introduza o leitor o seguinte programa em linguagem máquina usando HEXLD (terá de procurar os diversos códigos necessários):

```
LD BC,0
LD HL,0
ADD HL,BC
LD B,H
LD C,L
RET
```

Correndo agora o programa, que resposta se obtém? Se for zero, está bem! Mas se tal não acontecer é porque se cometeu algum erro fundamental. LD BC e LD HL são instruções de três *bytes* — isto quer dizer, por exemplo, que LD HL,0 é 210000 e não apenas 2100. Que instruções recebeu realmente o computador para que desse a resposta obtida? Tente-se de novo, até obter zero.

Se se apagar HEXLD fazendo NEW o programa em código

máquina *permanecerá na memória*. Isto resulta de o código máquina ser guardado numa parte da memória à qual o BASIC não tem acesso. Vejamos o seguinte programa:

```
10  INPUT A
20  INPUT B
30  POKE 61441,A-256*INT (A/256)   (usa-se 28673 com o 16 K)
40  POKE 61442,INT (A/256)         (usa-se 28674 com o 16 K)
50  POKE 61444,B-256*INT(B/256)    (ou 28676)
60  POKE 61445,INT(B/256)          (ou 28677)
70  PRINT A,B
80  PRINT USR 61440                (ou 28672)
90  PRINT
100 GOTO 10
```

Este programa BASIC substituirá o segundo, terceiro, quinto e sexto *bytes* do programa de código máquina. Isto quer dizer que o valor que introduzir para a variável A será depositado na memória a seguir a LD BC e o valor que introduzir para B será colocado na memória a seguir à instrução LD HL. Deve-se correr o programa e usar outros valores para ver o que acontece, experimentando sair dos limites Ø a 65535.

Seria vantajoso escrever agora um programa semelhante em BASIC que, uma vez combinado com uma versão ligeiramente modificada do nosso programa em código máquina, escrevesse no *écran* uma tabela de valores de A e B e o resultado da subtracção A menos B em cada um dos casos. Deveria atribuir-se a A e B todos os valores de Ø a 1Ø, inclusive.

Escreva-se também uma rotina em código máquina que retorna um se BC for maior ou igual a DE e zero se for menor. Como se poderá testar isto? (Damos uma pista — o exercício anterior desta página.)

CAPÍTULO QUATRO

PEEK e POKE e mais algumas coisas sobre LOAD

Para aqueles que pensaram que sete registos talvez não fossem suficientes, é bom saber que dispomos em linguagem máquina de instruções equivalentes a PEEK e POKE e podemos assim utilizar todos os endereços da RAM. (RAM significa *Random Access Memory* — memória de acesso aleatório, e é a parte da memória que podemos alterar — os endereços a partir de 16384.) Quando temos algum número que precisamos de guardar, permanentemente ou não, podemos guardá-lo em qualquer endereço da RAM (qualquer sítio serve). E quando precisamos dele bastará ir buscá-lo àquele endereço.

UMA LIÇÃO SOBRE «PEEK»

Quem alguma vez já viu linguagem máquina escrita, talvez se tenha perguntado porque é que aparecem parêntesis com frequência. Qual é, por exemplo, a diferença entre LD HL, 5C65 e LD HL, (5C65)? Os parêntesis não estão lá apenas para efeitos decorativos; têm um significado! Os parêntesis, quando cercam um número ou um par de registos, referem-se ao conteúdo do endereço entre parêntesis; assim:

	LD HL,5C65	significa	LET HL=23653
e	LD HL,(5C65)	significa	LET HL=PEEK 23653 +256*PEEK 23654

Para maior clareza, 23653 é o equivalente decimal de 5C65.

O segundo exemplo pode ter causado confusão. O único endereço entre parêntesis é 23652; por isso, como é que aí vemos 23654? O que aconteceu foi uma espécie de efeito secundário. H e L contêm, cada um, um *byte;* assim HL juntos contêm dois *bytes*. O endereço 23653 apenas tem um *byte;* logo um outro terá de vir de qualquer lado. Na prática, este outro *byte* vem-nos do endereço

seguinte, neste caso, 23654. O efeito real da instrução LD HL,(5C65) é LET L=PEEK 23653, seguida por LET H=PEEK 23654.

Há uma instrução inversa, que é:

```
LD (5C65),HL
```

Isto é o mesmo que fazer POKE. O resultado da instrução é:

```
POKE 23653,HL-256*INT (HL/256)
POKE 23654,INT (HL/256)
```

Ou então se se pensa em H e L separadamente:

```
POKE 23653,L
POKE 23654,H
```

Em BASIC este par de instruções usa-se com muita frequência. Vamos dar um exemplo: admitamos que se escreveu um programa BASIC e se quer saber qual o seu tamanho. Pode-se saber qual o número de *bytes* que o programa ocupa, escrevendo a expressão PRINT (PEEK 23627+256*PEEK 23628) – (PEEK 23635+256*23636). Um programa muito simples em código máquina para calcular este valor será:

ED5B535C	LD DE,(23635d)
2A4B5C	LD HL,(23627d)
C600	ADD A,0
ED52	SBC HL,DE
44	LD B,H
4D	LD C,L
C9	RET

A instrução ADD A,0 é utilizada para pôr o *carry* a zero, de maneira que a instrução SBC imediata dará sempre uma resposta correcta. Como a instrução SUB HL DL,DE não existe, se quisermos subtrair DE de HL teremos de usar SBC. No entanto, esta instrução não fará a subtracção correctamente a não ser que *carry* seja igual a zero.

Repare-se como o código para LD HL,(23627d) é composto. O primeiro *byte* é 2A. Agora, apesar de ser natural isso não nos ocorrer, a última vez que usámos uma instrução LD HL o código foi 21h. A diferença está nos parêntesis! As instruções LD que usam parêntesis têm um código hex completamente diferente. Os próximos dois *bytes* são 4B e 5C, isto é, o número 23627 em hexadecimal. Se dividirmos 23627 por 255, obtém-se noventa e dois (hex 5C) resto setenta e cinco (hex 4B). No código hex estes *bytes* foram trocados para obtermos 4B5C em vez de 5C4B. É preciso lembrar sempre de fazer isto em código máquina.

Se se guarda este programa em código máquina acima de RAMTOP (usando CLEAR, como foi descrito anteriormente), poderá fazer-se LOAD de qualquer programa BASIC e por fim saber qual o seu comprimento escrevendo o comando, já familiar, PRINT USR 61440 (ou 28672). 23627 terá sempre o endereço do primeiro *byte* a seguir ao fim do programa BASIC — é essa a sua função. É uma das variáveis de sistema que a ROM usa para ajudar a saber o que está a fazer. Da mesma maneira 23635 guardará sempre o endereço no qual o programa começa. Deve-se verificar isto, lendo o que o manual do *Spectrum*, na página 174, diz acerca das variáveis de sistema VARS e PROG, e comparar com o diagrama da página 165. Se se alterar qualquer destes valores, seja por meio de POKE ou de LD, a máquina ficará baralhada, apesar de tal por vezes ser vantajoso, como veremos mais tarde.

É necessário compreender como o programa acima mencionado funciona e também porque precisamos de cada uma das linhas. A instrução mais importante ainda é a que aprendemos primeiro — RET. Se qualquer das outras instruções faltar, obteremos uma resposta errada, mas pelo menos obteremos uma resposta. Sem RET, o programa entrará em quebra. Nem todos os registos podem ser carregados directamente a partir da memória. As instruções que podemos utilizar, juntamente com os seus códigos e um resumo daquilo que fazem, são aqui indicados:

3A	LD A,(pq)	LET A = PEEK pq
ED4B	LD BC,(pq)	LET C = PEEK pq LET B = PEEK (pq + 1)
ED5B	LD DE,(pq)	LET E = PEEK pq LET D = PEEK (pq + 1)

```
2A          LD HL,(pq)      LET L = PEEK pq
                            LET H = PEEK (pq + 1)
```

E fazendo POKE:

```
32          LD (pq),A       POKE pq,A
ED43        LD (pq),BC      POKE pq,C
                            POKE pq + 1,B
ED53        LD (pq),DE      POKE pq,E
                            POKE pq + 1,D
22          LD (pq),HL      POKE pq,L
                            POKE pq + 1,H
```

Como se vê, só o registo A permite a leitura ou escrita directa de ou para qualquer posição na memória, todos os outros registos têm de ser usados aos pares. Geralmente, esta é uma característica muito útil mas por vezes deseja-se ler um só registo (normalmente a escolha será L) sem alterar o valor de A. Na verdade, não há uma maneira de se resolver isto, mas podem ler-se ambas as partes do par de registos como descrevemos acima e depois repor o necessário registo simples a zero.

Suponhamos que queremos saber em que posição do *écran* se encontra PRINT. O manual de instruções informa que PEEK 23689 dará exactamente o que se quer saber (claro, menos 24d). A dificuldade é fazer LD em BC, porque o número que queremos tem só um *byte*, não está guardado nem em 23688 ou 23690. Uma maneira de o conseguir é:

```
ED4B895C    LD BC,(23689d)
0600        LD B,00
C9          RET
```

Como se vê, a primeira instrução carrega o conteúdo de 23689 no registo C, como pretendemos, mas também (e isto como efeito secundário) carrega o conteúdo de 23690 no registo B. Assim, o registo B deve ser tornado igual a zero de maneira que BC contenha o mesmo valor de C só por si antes de voltarmos ao BASIC; de outro modo o número escrito no fim praticamente não terá sentido.

Também se obtém PEEK 23689 em BC por meio do registo A, uma vez que esse registo pode ser carregado directamente. Mas isto não oferece vantagens, uma vez que teremos de fazer B igual a zero da mesma maneira:

```
3A895C      LD A,(23689d)
0600        LD B,00
4F          LD C,A
C9          RET
```

Para se verificar que a segunda instrução é necessária, é conveniente omiti-la. O que estará mal? Se o valor de B não for mudado, o valor de BC estará errado (na quantidade 256d*B).

Ambos os programas acima mencionados, da maneira como estão escritos, terão o mesmo efeito, dão a primeira coordenada da posição PRINT actual menos vinte e quatro. Isto é, dizem o número de quadrados abaixo da posição PRINT, incluindo todas as linhas da parte inferior do *écran*.

Para experiência, introduza-se um dos programas acima mencionados e a seguir este programa BASIC:

```
10 FOR I = 0 TO 20
30 PRINT USR 61440          (ou 28672 para quem usa 16 K)
50 NEXT I
```

Ao correr o programa, deve-se tentar compreender porque é que ele faz aquilo que faz. Agora insiram-se mais estas linhas:

```
20 FOR J = 0 TO 3
30 PRINT TAB (8*J);USR 61440  (isto substitui a anterior linha 30)
40 NEXT J
```

e corra-se outra vez para ver o que acontece. É interessante, não?

POKE EM CÓDIGO MÁQUINA

Fazer POKE é igualmente fácil. Para colocar a linha 50 de qualquer programa BASIC no cimo do *écran*, na próxima listagem automática, nós podemos fazer POKE 23660,50 e depois POKE 23661,0

(o que só funcionará se a linha corrente, isto quer dizer, a linha com o cursor «maior que», estiver no *écran* quando a linha 5Ø estiver no cimo do *écran*).

Em código máquina:

```
213200      LD HL,0032      (0032h = 50d)
226C5C      LD (5C6C),HL    (5C6B = 23660d)
C9          RET
```

Mudando o cursor para qualquer linha maior que cinquenta, nota-se que, pela primeira vez, não importa qual o número que é retornado ao BASIC pela rotina de código máquina no regito BC porque o objectivo deste exercício é fazer um POKE duplo. Uma vez que não nos interessa como é que BC acaba, não faz sentido fazer PRINT; assim PRINT USR não é necessário. Mais vale usar RANDOMIZE USR. Repare-se que, ao fazê-lo, a linha 5Ø se desloca da mesma forma para o topo do *écran* (é essa a função do código máquina) mas nenhum número aparecerá escrito, obviamente, uma vez que não se usou PRINT. Assim, o que é que acontece ao número «lembrado» pela rotina, isto é, o número que ficou em BC quando se alcança RET? Encontra-se a resposta lendo as páginas 73 e 74 do manual do *Spectrum*, que se referem a RANDOMIZE — o número torna-se o novo valor da variável do sistema SEED. Isto não deve causar preocupação. Só precisamos de saber que PRINT USR e RANDOMIZE USR farão ambos correr um programa em código máquina, mas que PRINT escreverá o valor final de BC e RANDOMIZE não.

Agora veja-se o código hex de LD (5C6C),HL. O primeiro *byte* é 22h. Este é o código hex para LD (pq),HL, onde pq representa um endereço arbitrário. O resto do código é 6C5,C que é o número 23660d em hexadecimal (com, é claro, os primeiros e último *bytes* trocados). Assim, apesar de nós, humanos, escrevermos o nosso *opcode* com o (5C6C) primeiro e o HL depois, o código máquina exige que a instrução venha primeiro, mesmo se HL estiver no fim do *opcode*.

Existem outras instruções PEEK e POKE, mas estas referem nomes de outros registos. São eles:

```
0A      LD A,(BC)       LET A = PEEK BC
1A      LD A,(DE)       LET A = PEEK DE
```



```
7E      LD A,(HL)       LET A = PEEK HL
46      LD B,(HL)       LET B = PEEK HL
4E      LD C,(HL)       LET C = PEEK HL
56      LD D,(HL)       LET D = PEEK HL
5E      LD E,(HL)       LET E = PEEK HL
66      LD H,(HL)       LET H = PEEK HL
6E      LD L,(HL)       LET L = PEEK HL

02      LD (BC).A       POKE BC,A
12      LD (DE),A       POKE DE,A
77      LD (HL),A       POKE HL,A
70      LD (HL),B       POKE HL,B
71      LD (HL),C       POKE HL,C
72      LD (HL),D       POKE HL,D
73      LD (HL),E       POKE HL,E
74      LD (HL),H       POKE HL,H
75      LD (HL),L       POKE HL,L
```

Se se estudam os códigos das instruções que têm «(HL)», vê-se que formam um padrão regular. Agora, colocando todos estes códigos numa tabela de todas as instruções LD do tipo LD A,B, vê-se uma regularidade ainda mais espantosa. Observe-se a figura 4.1

LD	B	C	D	E	H	L	(HL)	A
B	40	41	42	43	44	45	46	47
C	48	49	4A	4B	4C	4D	4E	4F
D	50	51	52	53	54	55	56	57
E	58	59	5A	5B	5C	5D	5E	5F
H	60	61	62	63	64	65	66	67
L	68	69	6A	6B	6C	6D	6E	6F
(HL)	70	71	72	73	74	75	—	77
A	78	79	7A	7B	7C	7D	7E	7F

Figura 4.1

Esta tabela leva a pensar que deveria existir uma instrução LD (HL),(HL) com código 76. Na verdade essa instrução não existe! O código 76 representa uma instrução completamente diferente, que se chama HALT, mas trataremos disso depois.

Porque é que qualquer registo entre parêntesis é um par de registos em vez de um só? Porque é que qualquer registo fora de parêntesis é um registo simples e não um par? Se HL contivesse 5C8F, qual seria a diferença entre LD B,(HL) e LD BC,(5C8F)? Qual é o verdadeiro efeito de cada um deles? Tente o leitor escrever uma rotina em código máquina para atribuir a HL o valor de PEEK 23693 apenas, utilizando só uma das instruções LD,(HL).

Já vimos todas as instruções LD básicas que operam nos registos A, B, C, D, E, H e L. Vamos agora ver algumas das outras maneiras de carregar estes registos.

COMO CARREGAR BLOCOS

Carregar blocos significa carregar grandes zonas da memória de uma só vez. Por exemplo, se houvesse uma rotina em código máquina no endereço 30000 e quiséssemos mudá-la para o endereço 20000, desde que tivéssemos paciência poderíamos fazer qualquer coisa como isto:

```
11204E      LD DE,20000d
213075      LD HL,30000d
7E          LD A,(HL)
12          LD (DE),A
23          INC HL
13          INC DE
7E          LD A,(HL)
12          LD (DE),A
23          INC HL
13          INC DE
....        ....
            E assim por diante.
```

Encurtaria um pouco as coisas conhecer a instrução LDI, que significa *LoaD* com Incremento. Esta é uma instrução muito especial, que executa quatro coisas de uma só vez. Primeiro,

transfere o conteúdo do endereço contido em HL para o endereço contido em DE; depois incrementa tanto HL como DE; e por fim decrementa BC. O valor do registo A não é alterado. Em resumo:

```
EDA0        LDI         POKE DE,PEEK HL
                        LET HL = HL + 1
                        LET DE = DE + 1
                        LET BC = BC - 1
```

O programa acima dado poderia ser totalmente reescrito:

```
11204E      LD DE,20000d
213075      LD HL,30000d
EDA0        LDI
EDA0        LDI
EDA0        LDI
...         ....
            E assim por diante.
```

Não existe uma lista de operandos depois do *opcode* LDI porque a instrução fará sempre *load* de (HL) a (DE). Não se deve escrever LDI (DE),(HL) porque é totalmente inútil: não se pode usar LDI para carregar qualquer outra combinação. Fazer *load* a partir de (HL) para (BC), por exemplo, não pode ser realizado numa só operação.

Existe ainda uma instrução chamada LDD (*LoaD* com Decremento) que tem o mesmo efeito que LDI excepto que DE e HL são decrementados em vez de incrementados. BC é decrementado como antes. Nenhuma destas instruções altera o valor do *flag carry* (como acontece com todas as instruções do tipo LD). O código para LDD é EDA8.

REPETIÇÕES

Mesmo dispondo de LDI e LDD, seria um trabalho maçador mudar qualquer coisa de, digamos, 30000 para 20000 se essa coisa tivesse cerca de cinquenta *bytes* de comprimento. E se tivéssemos 100 *bytes* certamente desistiríamos, desesperados. Felizmente tanto LDI

como LDD têm variantes com a facilidade de permitir repetições. Se escrevêssemos LDIR em vez de só LDI (com o R representando repetição), a instrução LDI seria executada continuamente e só pararia quando BC fosse zero. Assim, se a rotina que quiséssemos transferir tivesse 100 *bytes* de comprimento, poderíamos mudá-la utilizando a rotina:

```
016400      LD BC,100d
11204E      LD DE,20000d
213075      LD HL,30000d
EDB0        LDIR
```

Quando a máquina alcança a instrução LDIR, BC terá o valor de 100. Após a execução de LDI uma primeira vez, o primeiro *byte* terá sido transferido, DE incrementado para 20001d, HL será aumentado para 30001d e BC diminuído para noventa e nove. Depois da repetição de LDI o segundo *byte* será transferido e BC conterá noventa e oito. Após a centésima execução de LDI, toda a rotina terá sido transferida e BC conterá o valor de zero, o que quer dizer que o programa continua com a próxima instrução. Se esta rotina fosse o programa completo, então a próxima instrução seria, é claro, RET.

As quatro instruções LDI, LDD, LDIR e LDDR fazem cada uma delas coisas ligeiramente diferentes. É conveniente conhecer bem a diferença entre cada uma delas. Têm também códigos diferentes, começando por ED.

São eles:

```
EDA0        LDI
EDA8        LDD
EDB0        LDIR
EDB8        LDDR
```


CAPÍTULO CINCO

Empilhando e saltando

A PILHA (STACK)

Existe uma zona de RAM reservada a diversas informações que servem para ajudar a máquina a saber o que está a fazer. Funciona da seguinte maneira:

A palavra «*stack*» (pilha) foi tirada directamente do dicionário. O seu sentido é esse mesmo. Imagine-se uma pilha de caixas de cartão. Cada caixa é, na realidade uma posição de memória; assim cada uma delas tem um endereço, mas se quisermos saber o que se encontra dentro de cada caixa de cartão, a única em que se pode espreitar facilmente é a que se encontra por cima. Se quiséssemos tirar alguma das caixas do meio da pilha, todas as que estivessem acima dessa cairiam. Do mesmo modo, para colocar uma nova caixa, o único sítio onde é possível fazê-lo facilmente é o topo da pilha.

As posições de memória da pilha funcionam da mesma maneira. Podem-se colocar coisas por cima, mas só por cima; e podem-se também tirar coisas de cima. Existem duas palavras próprias para associar ao trabalho com a pilha, uma que quer dizer «empilhar um novo número no topo» e outra que significa «retirar um número do topo»; a primeira palavra é PUSH 9ABC e PUSH 8000h, por exemplo, o primeiro número que se pode retirar da pilha com POP é 8000h, uma vez que é o número que se encontra, nesse momento, no cimo da pilha (porque foi o último número que foi aí «empilhado»); o segundo número que se pode retirar com POP é 9ABC e o terceiro é cinco.

No *Spectrum*, a pilha está guardada numa parte muito elevada da memória de endereços, de maneira que haja poucas hipóteses do programa BASIC «colidir» com ela à medida que tanto uma quanto o outro são aumentados. A pilha é na verdade muito peculiar, uma vez que está invertida. A parte inferior da pilha está perto do topo da memória disponível e o topo da pilha está abaixo desse limite. Acontece que assim é mais eficiente. Não se trata de uma conspiração para deliberadamente confundir toda a raça humana de modo que o mundo seja dominado pelos computadores ZX, mesmo

que por vezes o pareça. Por isso, lembre-se, a pilha ou *pilha da máquina*, como por vezes é chamada, é como uma pilha de caixas de cartão empilhadas no chão de uma loja, excepto que, num audacioso desafio às leis de Newton, esta pilha se encontrar «presa no tecto» e aumenta para baixo! O topo, a única parte a que temos acesso, está mais abaixo que a base.

A pilha é tão importante para o computador que há à parte um registo especial para guardar a posição do topo da pilha (a parte com o endereço mais baixo — aquele a que temos acesso). Esse registo chama-se SP, que significa *stack pointer* (ponteiro da pilha). Na realidade, trata-de de um par de registos, uma vez que guarda dois *bytes*, mas, ao contrário de outros pares de registos (BC, DE e HL), não podemos separar as duas metades: elas formam um todo único.

Eis como as instruções PUSH e POP funcionam. Faremos a exemplificação em hex porque nos parece ser a maneira mais fácil. Suponhamos que HL contém o valor ABCD. Isto quer dizer que H contém AB e L CD. A instrução PUSH HL irá guardar o número ABCD em duas posições no topo da pilha, começando por «empilhar» a parte alta (AB) e depois a parte baixa (CD). O valor de SP será diminuído em duas unidades por serem acrescentados mais dois *bytes* à pilha e a posição do topo será deslocada (para baixo) dois endereços.

Infelizmente não é possível fazer PUSH de endereços simples. Só se pode utilizar PUSH com pares de registos, de forma que BC pode ser «empilhado» mas B só por si não. Repare-se que PUSH BC não altera, de maneira alguma, o valor de BC, apenas o copia na pilha. Isto é, claro está, válido para todas as instruções PUSH.

A sequência de instruções BASIC apresentadas a seguir irá realizar as mesmas acções que PUSH.

```
PUSH HL        LET SP = SP - 2
               POKE SP + 1,H
               POKE SP,L
```

POP, é óbvio, funciona de maneira inversa. POP HL irá antes de mais tirar L da pilha e depois H. SP será aumentado em duas unidades por terem sido retirados dois *bytes* do topo da pilha.


```
POP HL         LET L = PEEK SP
               LET H = PEEK (SP + 1)
               LET SP = SP + 2
```

Verifique-se, utilizando as equivalentes BASIC dadas, que PUSH HL seguido de POP DE é a mesma coisa que LD D,H seguido de LD E,L.

PUSH

Estes são os códigos para a instrução PUSH. Um deles exigirá alguma explicação.

```
F5             PUSH AF
C5             PUSH BC
D5             PUSH DE
E5             PUSH HL
```

O par de registos AF, que normalmente não pode ser utilizado como tal, é constituído pelos registos simples A e F, do mesmo modo que BC é composto por B e C. A é o registo que já conhecemos através do livro mas F é algo completamente diferente. O F significa *flags*. Para compreender o significado de F, temos de o observar não em hex, mas em binário. F pode, por exemplo, ter um valor de 41h, que em binário é 01000001. Cada um dos dígitos é ou zero ou um e cada um dos diferentes dígitos tem um significado diferente, pois todos eles são *flags* diferentes. Já vimos um desses *flags*, o *flag carry*, que é guardado no dígito binário mais à direita de F. Não vamos preocupar-nos mais neste momento com F, uma vez que o veremos mais tarde em pormenor. Entretanto é bom perceber porque é que esta curta rotina fará o *flag carry set* (sem utilizar SCF). Não importa se não se conseguir agora, isto é apenas para nos obrigar a raciocinar.

```
0E01           LD C,01
C5             PUSH BC
F1             POP AF
```

POP

Os códigos para as instruções POP são muito semelhantes aos usados para PUSH. São eles:

```
F1          POP AF
C1          POP BC
D1          POP DE
E1          POP HL
```

Um dos usos mais frequentes de PUSH AF e POP AF é apenas fazer PUSH e POP do valor de A. O facto de F ser também «empilhado» não interessa. PUSH AF irá guardar na pilha o valor de A até precisarmos dele de novo e nessa altura podemos recuperá-lo usando POP AF. Isto será útil se tivermos de usar o registo A para fazer cálculos de algum tipo que não possam ser feitos noutro registo e precisarmos do valor de A mais tarde no programa.

Por exemplo, para somar vinte e cinco ao valor de B sem alterar o valor de A (ou de qualquer outro registo):

```
F5          PUSH AF
78          LD A,B
C619        ADD A,19h (=25d)
47          LD B,A
F1          POP AF
```

Por que motivo somente B será alterado e mais nenhum outro registo? (Nem mesmo o *flag carry!*) Será bom compreender o que a rotina acima mencionada faz exactamente, antes de prosseguir a leitura.

ALTERAR SP

SP é utilizável de maneira muito semelhante a BC e DE. Podemos somá-lo e subtraí-lo e também carregá-lo. Os códigos hex são:

```
F9          LD SP,HL
31????      LD SP,mn
```



```
ED7B????    LD SP,(pq)
ED73????    LD (pq),SP
39          ADD HL, SP

ED7A        ADC HL,SP
ED72        SBC HL,SP
33          INC SP
3B          DEC SP
```

Isto é muito poderoso e muito útil. Se queremos trocar os valores de D e E sem alterar mais nada, recorremos à rotina seguinte:

```
D5          PUSH DE
D5          PUSH DE
33          INC SP
D1          POP DE
33          INC SP
```

O INC SP final é necessário para restaurar o *stack pointer* no seu valor original. Se não o fizermos, quase de certeza entrará em quebra.

SP não é o único registo especializado de que dispomos. Existe um outro registo de dois *bytes*, chamado PC, *Program Counter* (contador de programa ou apontador de instrução). A sua função é lembrar-nos em que posição do programa é que nos encontramos. De cada vez que o *Spectrum* executa uma instrução começa por ver o registo PC. Se PC contiver FØØ4, por exemplo, o computador executará a instrução da posição FØØ4 e irá incrementar o valor de PC com o número de *bytes* dessa instrução, de maneira a apontar para a próxima instrução da sequência. Por exemplo, se FØØ4 contivesse a instrução LD A,B esta instrução seria executada e PC seria incrementado para FØØ5. Se a instrução em FØØ5 fosse LD B,2, uma vez executada, PC aumentaria no valor de dois, já que LD B,2 tem o comprimento de dois *bytes*. PC seria então FØØ7 no começo da próxima instrução.

A alteração do valor de PC tem um efeito semelhante ao de GO TO em BASIC. A única diferença está no facto de que o código de máquina não usa números de linha. A instrução em linguagem máquina para alterar o valor de PC é JP, que quer dizer, obviamente, *JumP* (salto). JP FØØØ significa GO TO endereço FØØØ e continua a executar o código máquina a partir daí. Claro que o

efeito desta instrução realmente é carregar o número FØØØ em PC (mas sem o incrementar no fim da instrução) de maneira que a máquina considere FØØØ como o próximo endereço do programa. No entanto é mais fácil para nós pensarmos em termos de uma instrução semelhante a GO TO porque já nos habituámos a ela.

É preciso cuidado com JP. Se se fizer um *loop* (ciclo) infinito em código máquina fica-se perdido. Não se poderá sair dele a não ser desligando a máquina. Um exemplo de um ciclo infinito será:

```
77          F000      LD (HL),A
23          F001      INC HL
C300F0      F002      JP F000
```

Escrevemos os endereços na coluna do meio, o que não é costume fazer-se. As linhas importantes são marcadas com *labels* (etiquetas), palavras que nos indicam o que fazem essas linhas. Estas etiquetas não aparecem em hex; de facto, só as escrevemos para uso próprio. Se, por exemplo, chamássemos START à primeira linha, escreveríamos assim o nosso programa:

```
77          START     LD (HL),A
23                    INC HL
C300F0                JP START
```

Existe uma outra instrução (semelhante a JP): JR, ou seja *Jump Relative* (salto relativo), que significa saltar um número dado de *bytes*. É, de certa maneira, melhor que JP porque tem só dois *bytes* de comprimento em vez de três e também porque uma rotina completa pode ser carregada noutra posição de memória e executada sem precisarmos de mudar os destinos de todos os JP. JRØ não tem efeito algum e a próxima instrução na sequência será executada. No entanto JR1 fará com que a próxima instrução (partindo do princípio de que é uma instrução de um só *byte*) não seja executada. Para saltar sobre instruções de dois *bytes* ou duas instruções de um só *byte*, será necessário usar JR2.

Também é possível «saltar para trás» utilizando JR, uma vez que existe uma convenção que diz que qualquer número hex entre 8Ø e FF será tomado como negativo (na verdade 256d menos o número que representaria normalmente). Para facilitar as coisas inclui-se

uma tabela de números hexadecimais entre 8Ø e FF e os números decimais negativos que os representam. Esta tabela é a segunda do Apêndice 1, do fim deste livro. Compare-se com a tabela anterior. Observe-se que o número, digamos, menos cinco, está representado por FB e, assim, é possível utilizar a instrução JR−5, mas repare-se que por causa desta convenção não podemos fazer JR 129d, por exemplo, porque 129d em hex é 81, ou seja −127d, que corresponde nesse caso a um salto para trás. Portanto, a região a que estamos limitados vai de −128d a 127d em torno do valor de PC.

JRØ, como já foi dito, não faz absolutamente nada. A execução do programa continua na próxima instrução. É importante lembrar que os saltos negativos são contados a partir do valor de PC, isto é, da instrução seguinte. JRØ significa: execute a partir do endereço da próxima instrução mais zero. JR1 significa: execute a partir do endereço da próxima instrução mais um. Em consequência, se tivéssemos JR−2 teríamos de contar dois *bytes* para trás partindo do endereço da próxima instrução. Vemos que dois *bytes* atrás nos levam exactamente à instrução que acabamos de executar, a instrução JR−2. JR−2 é portanto um ciclo infinito e não recomendado para ser usado num programa.

O programa de *loop* infinito acima mencionado pode ser agora reescrito com menos um *byte*, utilizando JR em vez de JP.

```
77          START     LD (HL),A
23                    INC HL
18FC                  JR START
```

Repare-se que escrevemos «JR START» em vez de «JR−4»: o programa torna-se muito mais fácil de seguir, já que só precisamos de olhar para a etiqueta START para saber onde é que JR nos leva, em vez de termos de contar os *bytes*.

JR e JP são praticamente inúteis quando não forem utilizados com condições, da mesma maneira que GO TO em BASIC seria inútil se não existissem as instruções IF-THEN GO TO. Necessitamos de uma espécie de *jump* condicional de modo que possamos dizer que, se (IF) certa condição se cumpre, então (THEN) salte para um novo endereço. Sem esta hipótese, JP e JR só podem ser utilizados para criar ciclos infinitos. Apesar de o código máquina não ser flexível como o BASIC, permite-nos o uso de quatro condições para JR e oito para JP. São estas (para JR):

18ee	JR e	Salto relativo de e *bytes*.
20ee	JR NZ,e	IF o último resultado calculado for diferente de zero THEN salto relativo de e *bytes*.
28ee	JR Z,e	IF o último resultado é zero THEN salto relativo de e *bytes*.
30ee	JR NC,e	IF CARRY=Ø THEN salto relativo de e *bytes*.
38ee	JR C,e	IF CARRY=1 THEN salto relativo de e *bytes*.

E para JP:

C3qqpp	JP pq	Salto p. o endereço pq.
C2qqpp	JP NZ,pq	IF o último resultado calculado for diferente de zero THEN saltar para o endereço pq.
CAqqpp	JP Z,pq	IF o último resultado for zero THEN saltar p. o endereço pq.
D2qqpp	JP NC,pq	IF CARRY=Ø THEN saltar p. o endereço pq.
DAqqpp	JP C,pq	IF CARRY=1 THEN saltar para o endereço pq.
E2qqpp	JP PO,pq	Ver abaixo
EAqqpp	JP PE,pq	Ver abaixo
F2qqpp	JP P,pq	IF o último resultado calculado for positivo (Plus) THEN saltar para o endereço pq.
FAqqpp	JP M,pq	IF o último resultado calculado for negativo (Minus) THEN saltar para o endereço pq.

Agora, apesar de estarmos muito longe, por exemplo, do IF A$="OLÁ" THEN PRINT "ADEUS" a que estamos habituados, em breve se verá que até mesmo esta tarefa «difícil» pode ser executada em código máquina. Primeiro, no entanto, explicaremos duas das instruções na lista anterior — JP PO e JP PE. Falando claramente, JP PO significa IF PV=Ø THEN salta *(jump)* para o endereço pq. Mas o que é exactamente PV?

Resposta: PV é um outro *flag*, exactamente como *carry*. Como este, também só pode guardar um dos valores, um ou zero. P significa paridade, mas explicaremos isso mais tarde. O V significa *oVerflow* (excesso). Este uso é fácil de explicar.

JP PO pode ser entendido como JP NV (salte se não *oVerflow*).
JP PE pode ser entendido como JP V (salte se *oVerflow*).

Em rigor, não se deve escrever JP NV ou JP V porque não está convencionado, mas a verdade é que é uma ajuda para a nossa memória. A tendência é para escrever JP NV ou JP V nos programas — não importa, realmente, o respeitar ou não as convenções desde que se saiba o sentido do que se está a fazer (NV=PO e V=PE).

Voltemos agora ao significado de *oVerflow*. Se considerarmos os números de 8Ø a FF como negativos e os números de ØØ a 7F como positivos (como vimos anteriormente neste capítulo), poderão suceder coisas muito estranhas, em aritmética, se quisermos ultrapassar esses limites.

Por exemplo, 41h (positivo) mais 41h (positivo) é igual a 82h (negativo)?

Este género de erros é chamado *overflow* (excesso) e assim JP V salta se ocorrem erros destes e JP NV salta se tal não acontecer. Em resumo, um *overflow* acontece quando o resultado de uma operação aritmética, operando sobre números positivos ou negativos segundo a convenção indicada (também chamada de 2.º complemento) der uma resposta com sinal errado.

Repetindo mais uma vez; JP NV é uma maneira (não *standard*) de escrever JP PO e JP V um modo de escrever JP PE. Assim NV=PO e V=PE. É sempre indispensável lembrarmo-nos disto.

Todos estes testes variados, se se combinam devidamente com outras instruções, são suficientes para cobrir qualquer situação concebível. Na verdade, só é precisa mais uma instrução para tornar

JP e JR tão fortes quanto IF-THEN GO TO — essa instrução é CP ou ComPare.

CP compara o registo A com qualquer outro registo ou constante numérica. Por exemplo, podemos ter CP B (compare A com B) ou CP L (compare A com L) ou CP 3E (compare A com o número 3E). O efeito da instrução será entendido como LET DUMMY=A−B em BASIC.

Por outras palavras, executa-se uma subtracção, mas o resultado dessa subtracção é «esquecido» e o valor de A nunca muda. Os *flags*, no entanto, mudam e dizem-nos o necessário sobre o resultado. Se A contém Ø5 e B Ø6, então depois de uma instrução CP B, a instrução JP NZ executará um salto (uma vez que Ø5−Ø6 é não zero), JP C também (já que Ø5 é menor que Ø6), JP NV também (não há *overflow*, uma vez que o resultado de Ø5−Ø6=FF ou seja, −1, que é negativo e portanto com o sinal correcto) e JP M funcionará (uma vez que FF é negativo). Embora a subtracção seja executada, devemos salientar que o resultado é posto de parte e o valor de A não se altera.

CP permite-nos fazer algumas coisas úteis:

IF A = B THEN GO TO ...	CP B / JR Z ...
IF A< B THEN GO TO...	CP B / JR C ...
(Só funciona se supusermos que todos os números são positivos.)	
IF A< B THEN GO TO ...	CP B / JP M ...

CALL

Mesmo em código máquina podemos ter sub-rotinas. O equivalente em código máquina para GO SUB é CALL. Escrevemos CALL pq para significar GO SUB, a sub-rotina que começa no endereço pq. A instrução equivalente a RETURN é RET. Parece familiar... RET tem um duplo propósito — no fim de uma sub-rotina significa «retorne dessa sub-rotina»; se não houver sub-rotina para fazer retorno, significará «retorne ao BASIC». CALL e RET podem ser testados como vemos abaixo:

CDqqpp	CALL pq	C9	RET
C4qqpp	CALL NZ,pq	CØ	RET NZ
CCqqpp	CALL Z,pq	C8	RET Z
D4qqpp	CALL NC,pq	DØ	RET NC
DCqqpp	CALL C,pq	D8	RET C
E4qqpp	CALL PO,pq "CALL NV,pq"	EØ	RET PØ "RET NV"
ECqqpp	CALL PE,pq "CALL V,pq"	E8	RET PE "RET V"
F4qqpp	CALL P,pq	FØ	RET P
FCqqpp	CALL M,pq	F8	RET M

Como se adivinha facilmente, instruções como RET Z, etc., também se utilizam para condicionar o retorno ao BASIC, isto é, RET Z é igual a IF zero THEN RETURN ao BASIC.

Em BASIC não há pilha, por isso não temos de nos preocupar com ela. Em código máquina, no entanto, existe pilha e temos de nos preocupar. Existem duas instruções que, além de PUSH e POP, alteram a pilha! São elas CALL e RET!

CALL pq é equivalente a PUSH PC, o endereço da instrução seguinte, seguido de JP pq.

RET é equivalente a POP PC, provocando um salto para o endereço que estava no topo da pilha.

É importante compreender por que motivo isto leva CALL e RET a agirem como o fazem. A respeito destas instruções devemos ter cuidado com alguns problemas.

O valor de SP não deve ser alterado durante uma sub-rotina, uma vez que tanto CALL como RET dependem da pilha. Pode-se fazer PUSH de tantos itens quantos se quiser durante uma sub-rotina, desde que se faça um número igual de POPs antes de retornar. É útil conhecer um truque hábil para alterar o endereço de retorno de uma sub-rotina. Digamos que queremos mudar o endereço de retorno para EØØØ:

```
E1          POP HL
2100E0      LD HL,E000
E5          PUSH HL
```

A primeira instrução retira o endereço de retorno original da pilha. A segunda e terceira instruções substituem-no por um

endereço de retorno alternativo. Quando, mais tarde, se alcança uma instrução RET, o controle «retornará» ao endereço EØØØ. Outro truque útil a saber é como ter a certeza de que as sub-rotinas terminam sempre com a pilha «equilibrada». Uma maneira de o fazer é guardar o valor do *stack pointer* no princípio da rotina e restaurá-lo no fim. Um bom lugar para guardar este valor é a variável de sistema SPARE no endereço 5CBO. Vejamos como fazer.

No princípio de uma sub-rotina:

```
ED73B05C        LD (5CBØ),SP
```

A sub-rotina pode ter então mais PUSH que POP ou vice-versa e deverá acabar por:

```
ED7BB05C        LD SP,(5CBØ)
C9              RET
```

Este método não impede no entanto a possibilidade de alteração do endereço de retorno, como o leitor poderá facilmente verificar.

EXERCÍCIOS

Para o leitor se certificar de que compreendeu o uso da pilha e das instruções condicionais, escreva um programa que coloque na pilha cada um dos números entre um e cinquenta (decimal) usando PUSH AF e então tente retornar a BASIC sem suscitar quebra. Sugestão: CP 32h (compare com cinquenta decimal) é uma instrução muito útil neste caso.

É preciso conhecer os diversos códigos para CP. São eles:

```
BF      CP A
B8      CP B
B9      CP C
BA      CP D
BB      CP E
BC      CP H
BD      CP L
BE      CP (HL)
FEnn    CP n
```


CAPÍTULO SEIS

«Bytes» e «bits»

A PILHA

Uma das coisas boas do mundo dos computadores é que utiliza um óptimo vocabulário, PEEK, POKE, PUSH, POP, etc. Outra dessas palavras é *bit*. *Bit* quer dizer *Binary Digit* (dígito binário). Um *bit* só se compreende em binário, de modo que vamos trabalhar um pouco em binário.

Vamos partir do princípio que o leitor já compreende o hexadecimal, ainda que não esteja familiarizado com ele. (Na realidade é muito fácil, números de dois dígitos são cerca de $1^1/_2$ vez aquilo que parecem, números de três dígitos cerca de $2^1/_2$ e números de quatro dígitos são mais ou menos quatro vezes aquilo que parecem.) Assim todos os números que aparecerem neste capítulo serão ou binários ou hexadecimais. Binários e hexadecimais estão intimamente relacionados. De facto, binário é o mesmo que hex, mas de maneira diferente!

Eis uma tabela de conversão. As colunas da esquerda contêm dígitos hex e a coluna da direita mostra grupos de quatro dígitos binários.

```
0    0000
1    0001
2    0010
3    0011
4    0100
5    0101
6    0110
7    0111
8    1000
9    1001
A    1010
B    1011
C    1100
D    1101
E    1110
F    1111
```

Todos os dígitos hex simples estão incluídos nesta lista e também, mas isto é menos óbvio, todos os grupos possíveis de quatro *bits* (um dígito binário é ou zero ou um). É impossível imaginar algum conjunto de quatro zeros ou uns que não se encontre na lista. Para transformar um número hex em binário só temos de converter um dígito de cada vez utilizando esta tabela. Por exemplo, o número 2A torna-se em binário ØØ1Ø1Ø1Ø uma vez que 2 corresponde a ØØ1Ø e A corresponde a 1Ø1Ø −ØØ1Ø e 1Ø1Ø colocam-se lado a lado e na ordem correcta (o 2 vem em primeiro lugar, e em seguida o A — assim, 2A).

Do mesmo modo, números de quatro dígitos hexadecimais podem ser convertidos em binário por meio desta simples regra: E6ØA torna-se 1111Ø 0110 ØØØØ 1Ø1Ø. (Escrevemos com espaços para facilitar a leitura, mas também poderia ter sido escrito 111ØØ11ØØØØØ1Ø1Ø ou 1111Ø11Ø ØØØØ1Ø1Ø — os espaços destinam-se aos nossos olhos, não têm outro significado.)

Para reconverter um número de binário a hex por meio de uma regra que funcione é essencial que o número binário em questão tenha um número de dígitos múltiplo de quatro. Se não for este o caso, a primeira coisa a fazer é dar-lhe um número de dígitos múltiplo de quatro. Isto faz-se precedendo-o de zeros. Assim, por exemplo, 1Ø1 passa a Ø1Ø1 e 1Ø1Ø1Ø passa a ØØ1Ø1Ø1Ø e ainda 111ØØØ11ØØØØ1 passa a ØØØ1 11ØØ Ø11Ø ØØØ1. (Colocamos espaços no último exemplo para torná-lo mais claro.) Tendo dividido o número em grupos de quatro dígitos, iremos procurar os dígitos na tabela acima dada ou na tabela do Apêndice 1, no fim do livro, um grupo de cada vez, do primeiro ao último; ØØØ1 11ØØ Ø11Ø ØØØ1 torna-se 1C61. Se convertesse 1C61 outra vez em binário, obteria ØØØ1 11ØØ Ø11Ø ØØØ1.

Não é necessário estar sempre a olhar para a tabela. Acaba por se aprender de cor, com o tempo, pois é bastante curta. Mas uma maneira melhor é compreender o que tudo isto significa. Em decimal estávamos habituados a trabalhar em colunas (pelo menos antes do aparecimento das calculadoras) — unidades, dezenas, centenas, milhares, etc. Em hex também temos colunas: de unidades, de dezasseis, de duzentos e cinquenta e seis, de quatro mil e noventa e seis e assim por diante. Assim, quando escrevemos 12BC na realidade queremos dizer o número 4Ø96*1+256*2+16*11+1*12 (decimal).

Veja-se como a coluna progride: cada número é dezasseis vezes maior que o anterior. Em binário também temos colunas, mas aqui as colunas são unidades, dois, quatro, oito, dezasseis, trinta e dois, sessenta e quatro e cento e vinte e oito, cada coluna é o dobro da anterior: assim ØØ1Ø1Ø1Ø significa na verdade (em decimal) 128*Ø + 64 * Ø + 32 * 1 + 16 * Ø + 8 * 1 + 4 * Ø + 2 * 1 + 1 * Ø, ou quarenta e dois, ou em hex 8Ø * Ø + 4Ø * Ø + 2Ø * 1 + 1Ø * Ø + Ø8 * 1 + Ø4 * Ø + Ø2 * 1 + Ø1 * Ø ou 2A. Repare-se como a soma se torna muito mais fácil se for feita em hex em vez de em decimal.

Agora, os *bits* — qual o interesse deles? Há evidentemente um motivo para nos ocuparmos deles, pois certamente que têm algum uso. Podemos manipular registos em *bits*, em código máquina, utilizando diversas instruções.

INSTRUÇÕES SOBRE «BITS»

A primeira dessas instruções sobre as quais nos debruçaremos é CPL, a abreviatura de ComPLemento. Escreve-se apenas CPL, sem referências a quaisquer variáveis. Tem por efeito mudar o valor de A mudando cada um dos *bits* de A. Se algum *bit* for inicialmente zero, então CPL mudá-lo-á para um, e se um *bit* for um, CPL mudá-lo-á para zero. Note-se que esta instrução opera com o registo A. Para o tornar mais claro, e supondo que A contém um valor de ØØ111ØØ1, CPL muda-o para 11ØØØ11Ø — todos os zeros se tornam uns e todos os uns se tornam zeros. Em hex, A mudou de 39 para C6. Veja o leitor se consegue demonstrar que o efeito de CPL pode ser obtido em hex subtraindo cada dígito de F (F−3=C,F−9=6, assim CPL muda 39 para C6).

A instrução seguinte a nível de *bit* que iremos ver é AND. AND escreve-se sempre com um nome de registo ou constante numérica a seguir — exemplo, AND B ou AND 3F. Como CPL, esta instrução actua apenas sobre o valor do registo A. O seu efeito é o de alterar A, mas por comparação *bit* por *bit* com o valor de um outro registo ou constante numérica. A regra é: se (IF) um dado *bit* de A é um e (AND) o *bit* correspondente do número ao qual o está a comparar é um, então (THEN) esse *bit* de A permanece um, senão esse *bit* de A tornar-se-á zero. Para tornar isto mais claro, calculemos o valor de

AND B, supondo que A contém o valor 11ØØØ1Ø1 e B Ø1Ø1Ø11Ø. A deve ser convertido um *bit* de cada vez: 1 AND Ø é Ø, 1 AND 1 é 1, Ø AND Ø é Ø, Ø AND 1 é Ø, Ø AND Ø é Ø, 1 AND 1 é 1, Ø AND 1 é Ø e por fim 1 AND Ø é Ø. Assim obtemos a resposta final de Ø1ØØØ1ØØ e é este o novo valor de A. Lembremos a regra: o um só aparece quando um *bit* de A é um e o *bit* correspondente de B também.

Claro que AND B não é a única possibilidade; podemos usar AND C ou AND 37h; existem muitas possibilidades. Como exercício veja o leitor se consegue ver o que AND A faz, independentemente do valor inicial de A.

Uma outra instrução muito semelhante é OR. OR, tal como AND, é escrito com o nome de um registo ou de uma constante numérica a seguir à palavra OR e o seu efeito é mudar o valor do registo A comparando-o *bit* por *bit* com o valor do que esteja escrito depois de OR. A regra é um pouco diferente, mas igualmente fácil de lembrar: se um *bit* de A for um, ou (OR) o *bit* correspondente do número a que o está a comparar for um, então esse *bit* de A torna-se um, em caso contrário esse *bit* de A permanecerá zero. Para melhor compreensão desta regra, estudemos o exemplo dado a seguir:

11000101	11000101	11000101
CPL	AND	OR
	01010110	01010110
igual	igual	igual
00111010	01000100	11010111

Assim Ø OR Ø é Ø, Ø OR 1 é 1, 1 OR Ø é 1 e 1 OR 1 é 1. Lembremos as regras: só aparece o um se o *bit* de A for um ou (OR) o *bit* correspondente em comparação for um.

O *flag carry* tem também o seu lugar nisto tudo. A instrução CPL muda o valor de A, mas sem alterar o valor de nenhum dos *flags*. Para começar, o valor do *flag* zero irá depender do valor final de A ser ou não zero. É este *flag* que é testado em instruções com JR Z ou RET NZ. Uma outra característica muito útil de AND e OR é que o *flag carry* será sempre *reset*. Isto é, depois de uma instrução AND ou OR, *carry* será sempre igual a zero. Isto quer dizer que a instrução AND A pode ser utilizada em vez de ADD A,Ø para posicionar o *flag carry* a zero sem alterar nenhum dos registos.

Vejamos agora mais uma intrução do género de AND e OR. Esta

instrução tem o nome de XOR (*eXclusive OR* — ou exclusivo). Tal como AND e OR, XOR escreve-se com um nome de registo ou constante numérica a seguir à palavra XOR e a sua função é alterar A comparando-o *bit* por *bit* com o registo ou número dado. Aqui a regra é a seguinte: se um dado *bit* de A for um ou o *bit* correspondente do número a que o está a comparar for um, mas não ambos, então esse *bit* de A torna-se um, caso contrário torna-se zero. Para usar o nosso exemplo:

11000101	11000101	11000101	11000101
CPL	AND	OR	XOR
	01010110	01010110	01010110
igual	igual	igual	igual
00111010	01000100	11010111	10010011

Aqui temos Ø XOR Ø é Ø, Ø XOR 1 é 1, 1 XOR Ø é 1 e 1 XOR 1 é Ø. É mais fácil lembrar esta regra da seguinte forma: só aparece o um se, e apenas se, o *bit* de A for diferente do *bit* correspondente de comparação.

Temos em seguida dois exercícios, só de verificação; não são difíceis. O primeiro é verificar que o efeito de XOR FF em A é o mesmo de CPL e o segundo é verificar que o efeito de XOR A em A é o mesmo de LD A,ØØ.

Tal como AND e OR, XOR também posiciona o *flag* zero segundo o valor final de A e faz o *carry* igual a zero.

ROTAÇÕES

O grupo de instruções que iremos ver a seguir é de um tipo bastante diferente. São as instruções *shift* e *rotate*, que operam sobre qualquer registo na posição de memória e não apenas sobre A.

Consideremos a primeira instrução: RL (que significa *rotate left* — rode à esquerda). Podemos escrever RL A, RL B, etc., mas não podemos escrever RL 3F. É necessário ter o nome de um registo depois de RL e não uma constante numérica. RL muda todos os *bits* do registo em questão uma posição para a esquerda. O que era inicialmente o *bit* mais à esquerda passa para o *carry* e o valor inicial do *carry* passa para o *bit* mais à direita; assim, se partirmos do princípio que o *carry* vale zero e A contém 11ØØØ1Ø1, a seguir a

uma instrução RL A, o *carry* será 1 e A irá conter 1ØØØ1Ø1Ø. Veja se consegue compreender porque é que RL A tem o mesmo efeito que ADC A,A. Observe-se que RL B, RL C, etc., não podem ser simuladas em termos de ADC.

A instrução seguinte é RR (*rotate right*, ou seja, rode à direita). RR move todos os *bits* do registo em questão uma posição para a direita. O que foi anteriormente o *bit* mais à direita passa para o *carry* e o valor anterior do *carry* passa para o *bit* mais à esquerda. Se *carry* = Ø e A 11ØØØ1Ø1, então RR A mudará as coisas de maneira que *carry* = 1 e A Ø11ØØØ1Ø. Tente o leitor entender porque é que RR C repetido oito vezes é o mesmo que RLC uma só vez.

A seguir temos umas instruções semelhantes, RLC e RRC, que significam respectivamente «rode à esquerda sem BASIC» e «rode à direita sem *carry*». Tal como anteriormente, cada *bit* do registo em questão é movido uma posição para a esquerda ou para a direita, mas aqui o *bit* que desaparece de um lado reaparece do outro; assim, por exemplo, se A contiver 11ØØØ1Ø1, RLC A irá mudar A para 1ØØØ1ØØ11 — o valor inicial de *carry* é irrelevante. O seu valor final será o do *bit* que muda de extremidade. Neste caso, um.

Esperamos que tenha ficado clara a diferença entre RL e RR e entre RLC e RRC. RL e RR têm ciclos de nove *bits*, enquanto que RLC e RRC têm ciclos de oito *bits*. RL D repetido oito vezes deixará também D inalterado. RL, RR, RLC e RRC mudarão todos os outros *flags* além do *carry* de acordo com o valor final da operação.

Por falar em *flags*, é agora a altura de voltarmos a explorar aquele estranho *flag* PV, que significa *Parity/oVerflow;* o seu uso para detectar excessos já nós vimos, mas, sempre que utilizamos uma das instruções a nível de *bit* já referidas, PV torna-se um ou zero, de acordo com uma regra diferente. Sempre que o resultado de uma das instruções referidas tiver um número par de «uns», dizemos que a paridade é par, e, sempre que o resultado tiver um número ímpar de «uns», dizemos que a paridade é ímpar. Instruções como JP PO (*jump* se a paridade for ímpar) ou RET PE (retorne se a paridade for par), por exemplo, reflectirão este número par ou ímpar.

Se pretendermos utilizar RL, RR, RLC ou RRC no registo A (em mais nenhum dos outros), há uma maneira mais curta de o fazer.

Quatro instruções — muito fáceis de aprender: RLA em vez de RL A, RRA em vez de RR A, RLCA em vez de RLC A e RCCA em vez de RCC A. O facto de faltarem os espaços é muito importante. RLA (etc.) executa a operação desejada mas sem alterar os *flags* à excepção do *carry*. Recordemos que só se pode usar estas instruções com o registo A — não podemos escrever RLE, temos de escrever RL E. O espaço tem significado.

Em seguida iremos ver as instruções *shift*, que são muito semelhantes às instruções *rotate*. Estas instruções estão ligadas à multiplicação e divisão por dois. A primeira delas é SLA, que multiplica por dois. SLA é a abreviatura de *Shift Left Arithmetic* (desloque à esquerda aritmeticamente) e deve ser escrita com o nome de um registo a seguir à palavra SLA. Em binário, o que acontece na realidade é que cada *bit* do registo é movido uma posição para a esquerda, o *bit* que anteriormente se encontrava mais à esquerda muda para o *carry* e o *bit* mais à direita torna-se zero. Compreende-se porque é que SLA A faz a mesma coisa que ADD A,A? (Note-se que não existe notação «curta» para esta instrução; SLAA, por exemplo, não é permitido.) O efeito de SLA em qualquer registo é multiplicar o valor desse registo por dois e esta é, talvez, a melhor maneira de nos lembrarmos. Um exemplo:

```
11ØØØ1Ø1
SLA
igual
1ØØØ1Ø1Ø
```

A seguir temos SRA ou *Shift Right Arithmetic* (desloque à direita aritmeticamente), cuja função é dividir por dois. Para esta instrução os números entre ØØ e 7F são considerados negativos. (Veja-se a tabela 2 do Apêndice 1.) Por outras palavras, supondo que D contém um valor de FE (menos dois), a seguir a uma instrução SRA, D conterá FF (menos um). Em binário, esta instrução desloca todos os *bits* uma posição para a direita, indo o *bit* que anteriormente se encontrava mais à direita para o *carry* e o *bit* mais à esquerda permanecendo sem alterações. Assim o *bit* mais à esquerda é sempre igual ao segundo *bit* mais à esquerda, depois de uma instrução SRA. Vamos demonstrar isto mais uma vez por meio de exemplos:

11000101	11000101
SLA	SRA
igual	igual
10001010	11100010

Por fim temos mais uma instrução *shift*, chamada SRL (*Shift Right Logical* — desloque à direita logicamente). Mais uma vez, a sua função é dividir por dois, mas, com esta instrução, todos os números, mesmo os que se encontram entre 8Ø e FF, são considerados positivos. (Veja-se a tabela 1 do Apêndice 1.) Por outras palavras, se D contém FE, este não é considerado como menos dois, mas sim como duzentos e cinquenta e quatro. Depois de uma instrução SRL D, D passará a conter 7F ou seja, cento e vinte e sete e não FF. Em binário, SRL move cada *bit* uma posição para a direita, indo o *bit* inicialmente mais à direita para o *carry*, e tornando-se zero o *bit* mais à esquerda. Por exemplo:

11000101	11000101	11000101
SLA	SRA	SRL
igual	igual	igual
10001010	11100010	01100010

SET E RESET

Quando escrito em binário, um *byte* tem oito *bits* de comprimento. Cada um destes *bits* está numerado. O *bit* mais à esquerda é chamado *bit* número sete e o *bit* mais à direita é chamado zero. Os *bits* do meio são numerados da esquerda para a direita segundo uma ordem descendente, como se poderia esperar.

Em código máquina, podemos operar com os *bits* individuais dos registos. Dispomos das instruções SET e RES (abreviatura de *reset*). Por exemplo, SET 7,H tornará um o *bit* número sete do registo H. *Set*, recordemos, significa mudar para um. RES 5,D irá tornar zero o *bit* número cinco do registo D. *Reset*, é claro, quer dizer mudar para zero. Note-se que só podemos operar sobre *bits* específicos dos registos. Não podemos, por exemplo, dizer RES B,C — antes da vírgula só podemos colocar constantes numéricas (entre Ø e 7) e depois da vírgula só nomes de registo. No entanto em

muitos casos podemos escrever «(HL)», isto é, a posição de memória do endereço contido em HL, como se fosse um registo e, assim, AND (HL), OR (HL), XOR (HL), RL (HL), RR (HL), RLC (HL), RRC (HL), SLA (HL), SRA (HL), SRL (HL), SET 3,(HL) e RES 3,(HL) funcionarão.

Também podemos testar *bits* individuais. Existe uma instrução chamada BIT que opera da maneira que a seguir indicamos.

BIT 4,B (por exemplo) irá testar o *bit* número quatro do registo B. O valor de B não é alterado, mas o *flag* zero irá reflectir o resultado do teste. O *bit* número quatro de B é zero ou não zero e, assim, instruções como JR Z ou CALL NZ funcionarão como seria de esperar depois de uma instrução BIT. Claro que qualquer registo e também (HL), e não apenas B, pode ser testado e qualquer outro *bit*, não apenas o *bit* quatro, pode também ser testado.

Como já decerto se observou, neste capítulo não demos os códigos hex de nenhuma das instruções antes dadas. Teremos de procurá-los no Apêndice 4, no fim do livro. Todas as instruções de código máquina são aí apresentadas juntamente com os respectivos códigos hexadecimais. A nossa descrição de binário está completa. Agora, depois de tanta teoria, chegou a altura de utilizarmos tudo o que aprendemos, fazendo qualquer coisa útil.

CAPÍTULO SETE

Impressão no «écran»

IMPRESSÃO DE UM TABULEIRO DE DAMAS

Para escrevermos um programa tão extenso como o de um jogo de damas precisamos de um programa BASIC bastante potente para o podermos carregar. Segue-se uma versão de HEXLD — chamada HEXLD 2 — que apresenta melhoramentos em relação ao anterior. Um desses melhoramentos é a possibilidade de introduzir cadeias de caracteres, tais como «ser ou não ser», depois incorporados no código máquina, um carácter de cada vez. Para o conseguir, ao correr o programa é necessário introduzir: ser ou não ser: *(to be or not to be)* — isto é, o texto deve estar entre dois pontos (o que é muito importante).

Eis a listagem de HEXLD 2:

```
10 INPUT "ESCREVER PARA";A$
20 PRINT "ESCREVER PARA";A$
30 GO SUB 200
40 PRINT
50 LET A$=""
60 IF A$="" THEN INPUT A$:PRINT A$
70 IF CODE A$=58 THEN GO TO 300
80 LET Y=CODE A$-48: IF Y>9 THEN LET Y=Y-7
90 LET Z=CODE A$(2)-48:IF Z>9 THEN LET Z=Z-7
100 POKE X,16*Y+Z
110 LET X=X+1
120 LET A$=A$(3 TO)
130 GO TO 60
200 LET X=0
210 FOR I=1 TO 4
220 LET Y=CODE A$-48:IF Y>9 THEN LET Y=Y-7
230 LET X=16*X+Y
240 LET A$=A$(2 TO)
250 NEXT I
```



```
260 RETURN
300 LET A$=A$(2 TO)
310 POKE X,CODE A$
320 IF CODE A$=226 THEN POKE X,13
330 LET A$=A$(2 TO)
340 LET X=X+1
350 IF CODE A$<>58 THEN GO TO 310
360 LET A$=A$(2 TO)
370 GO TO 60
```

Este programa é basicamente o mesmo que HEXLD excepto por duas características. Primeira: é necessário introduzir o endereço do início (em hexadecimal) a partir do qual o código máquina será carregado, e segundo, é permitido introduzir cadeias de dados, utilizando os seus próprios códigos em vez de códigos hex — essa é a função da rotina que começa na linha 300. Para introduzir texto é necessário apenas colocar dois pontos de cada lado. Para introduzir o carácter ENTER como parte do texto é preciso introduzir em seu lugar STOP (*symbol shift* A). Para sair do programa, apagam-se as aspas usando EDIT e depois introduz-se STOP (sem aspas).

SUB-ROTINAS COM DADOS

Vamos ver algumas formas de uso. Uma das sub-rotinas mais úteis talvez seja a que escreve no *écran* uma cadeia de caracteres. Existe uma instrução em código máquina chamada RST 10 (que significa CALL 0010), que, no *Spectrum*, tem o efeito de escrever um só carácter. Experimente-se este programa, carregando-o no endereço 7000h.

```
3E2A      START    LD A,"*"
D7                 RST 10
18FB               JR START
```

Para podermos ver o programa correr, juntam-se as seguintes linhas de BASIC a HEXLD 2:

```
700 INPUT "ENDEREÇO DE EXECUÇÃO";A$
710 GO SUB 200
720 RANDOMIZE USR X
```

e depois fazemos RUN 7ØØ e introduzimos "7ØØØ". Repare-se na maneira como USR foi aqui utilizado — não com um número mas, sim, com uma variável, X. O endereço que USR chama é introduzido em hex na linha 7ØØ. Notou-se qualquer coisa estranha ao correr o código máquina? Experimente-se agora este programa em código máquina, um pouco diferente:

```
AF              XOR A           (Mais simples que LDA A,Ø)
323C5C          LD (TVFLAG),A
3E2A    LOOP    LD A,"*"
D7              RST 1Ø
18FB            JR LOOP
```

Repare-se que a diferença está em que a variável de sistema TVFLAG (endereço 5C3C) é primeiramente carregada com zero antes de ser executado o resto do programa. Quando correr, vê-se o *écran* encher-se de asteriscos, e muito depressa (muito mais depressa do que PRINT "*";/GO TO 1). A instrução RST 1Ø, no *Spectrum*, imprime o carácter cujo código estiver no registo A na posição corrente de impressão no *écran*.

O que queremos é uma sub-rotina que escreva qualquer mensagem, desde «sim» até «oh, que linda manhã», em amarelo e vermelho a piscar. Suponhamos que tal sub-rotina existe e se chama S__Print *(string print)*. Gostaríamos de usar uma instrução do género CALL S__PRINT com os dados *PAPER* amarelo, *INK* vermelha, *FLASH* activado e com a frase «oh, que linda manhã». (As palavras em itálico representam os caracteres de controle que se obtêm premindo ambos os *shifts* 6/, ambos os *shifts/caps shift* 2/, ambos os *shift/caps shift* 9.) Eis como empregar a rotina:

```
CD????          CALL S__PRINT
48656C6C6F      DEFM Hello
ØØ              DEFB ØØ
```

Aqui, DEFM significa DEFina uma Mensagem. Na verdade não se trata de uma instrução em código máquina mas de uma indicação

para especificar dados dentro de um programa. Procurando na tabela de códigos de caracteres (Apêndice 5, do fim do livro, ou na página 183 do manual do *Spectrum*) vê-se que 48 é o código hexadecimal para «H», 65 para «e», 6C para «l» e 6F para «o». DEFB refere-se também a dados — significa DEFina um *Byte*. DEFB C9 é, portanto, uma outra maneira de escrever RET. No exemplo acima dado, utilizamos o *byte* ØØ para indicar o fim da cadeia — o fim dos dados a usar por S__PRINT.

Temos de nos certificar, no entanto, de que a parte de dados nunca é executada, uma vez que estes *bytes* não fazem sentido em código máquina. Vejamos como escrever uma sub-rotina como S__PRINT e que ao mesmo tempo faça com que essa zona nunca seja executada (neste caso, a palavra «Hello» e o *byte* ØØ).

Já dissemos antes que CALL funciona fazendo PUSH do endereço de retorno e em seguida salta para o endereço requerido. RET funciona de maneira inversa — faz POP do valor no topo da pilha e depois salta para ele. Assim, se «Hello» vier imediatamente a seguir a uma instrução CALL, o endereço no topo da pilha (durante o decurso da sub-rotina) será o endereço do primeiro carácter da zona de dados — neste caso o «H» —, endereço esse que obtemos no registo HL com a simples instrução POP HL. Se então incrementarmos HL de uma unidade e fizermos PUSH HL, o efeito do proximo RETorno será um salto para o próximo endereço na linha — o «e». Podemos detectar o fim da zona de dados, procurando o *byte* ØØ (que não ocorre dentro do texto). Estudemos esta sub-rotina.

```
E1          S__PRINT    POP HL
7E                      LD A,(HL)
23                      INC HL
E5                      PUSH HL
A7                      AND A
C8                      RET Z
D7                      RST 1Ø
18F7                    JR S__PRINT
```

As primeiras quatro linhas têm por fim obter o carácter guardado no endereço de retorno corrente, incrementar este endereço e tornar a guardá-lo na pilha. As duas linhas seguintes provocarão o retorno da sub-rotina se o *byte* encontrado for ØØ. Note-se que nesse caso

AND A irá comparar A com ØØ. Se examinarmos a rotina com atenção veremos que a instrução RET (na verdade RET Z ou RETorno se Zero, mas o efeito é exactamente o mesmo) irá retornar ao *byte* a seguir a ØØ e não a ØØ mesmo. Por fim, se ØØ ainda não foi alcançado, a instrução RST 1Ø será executada e o carácter que se encontra no registo A aparecerá no *écran*. A rotina toda será então repetida o número de vezes necessário, até se encontrar o indicador de fim de mensagem — o *byte* ØØ.

Introduzamos o programa HEXLD 2 para poder carregar código máquina. Corramo-lo, introduzindo 7ØØØ para endereço de início, e introduzindo seguidamente tudo o que se encontra na coluna da esquerda:

```
E1          S__PRINT    POP HL
7E                      LD A,(HL)
23                      INC HL
E5                      PUSH HL
A7                      AND A
C8                      RET Z
D7                      RST 1Ø
18F7                    JR S__PRINT
AF          START       X OR A
323C5C                  LD (TVFLA G),A
CDØØ7Ø                  CALL S__PRINT
:fundo amarelo cor vermelha flash activo Oh, mas que linda manhã:
                        DEFM mensagem
ØØ                      DEFB ØØ
C9                      RET
```

Agora veja-se a utilidade da rotina BASIC em HEXLD 2, que começa na linha 3ØØ. Imagine-se como seria aborrecido ter de escrever 11Ø61ØØ212Ø14F682C2Ø... etc., em vez de *:fundo amarelo, tinta vermelha*, flash activo, oh, que linda manhã:. O efeito é exactamente o mesmo. Agora, com RUN 7ØØ para fazer correr o código máquina, introduz-se "7ØØ9" como endereço de início. Este é o endereço da etiqueta START do programa acima apresentado.

Se observarmos a linha 32Ø de HEXLD 2 notaremos que, de cada vez que introduzirmos a palavra «STOP» (*symbol shift* A) no texto, ela será imediatamente substituída por um carácter ENTER. Isto sucede apenas por uma questão de conveniência. Podemos

introduzir um carácter *enter*, se quisermos, apagando as aspas e escrevendo CHR$ 13, mas é muito mais simples e prático escrever apenas *symbol shift* A. É claro que se precisarmos da palavra «STOP» bastará escrever «S», «T», «O», «P», por extenso.

Agora, vamos precisar de criar um carácter gráfico definível. Introduzamos como um comando directo:

```
FOR I=Ø TO 7:INPUT X:POKE USR "A"+I,X:NEXT I
```

e a seguir

```
Ø
6Ø
126
126
126
126
6Ø
Ø
```

Desta forma definimos o gráfico A como uma peça de damas. O programa em código máquina que se segue constitui a primeira parte do jogo de damas (DRAUGHTS). É a rotina que nos permite desenhar o tabuleiro. Na coluna da direita, o símbolo a é utilizado para representar o carácter gráfico A. Devemos escrevê-lo a partir do endereço 7ØØØh.

```
E1          S__PRINT    POP HL
7E                      LD A , (HL)
23                      INC HL
E5                      P USH HL
A7                      AND A
C8                      RET Z
D7                      RST 1Ø
18F7                    JR S __ P R INT
AF          START       XOR A
323C5C                  LD (TVFLAG),A
CDØØ7Ø                  CALL S__PRINT
```

```
2031323334353637380D    DEFM     1 2 3 4 5 6 7 8
3120902090209020903 10D DEFM  1   @   @   @   @ 1
3290209020902090203 20D DEFM  2 @   @   @   @   2
3320902090209020903 30D DEFM  3   @   @   @   @ 3
3490209020902090203 40D DEFM  4 @   @   @   @   4
3520902090209020903 50D DEFM  5   @   @   @   @ 5
3690209020902090203 60D DEFM  6 @   @   @   @   6
3720902090209020903 70D DEFM  7   @   @   @   @ 7
3890209020902090203 80D DEFM  8 @   @   @   @   8
203132333435363738      DEFM     1 2 3 4 5 6 7 8
00                      DEFB 00 (Fim dos dados)
C9                      RET (Retorno ao BASIC)
```

Se fizermos agora RUN 700 e introduzirmos «7009» como endereço inicial (o endereço da instrução XOR A) vê-se aparecer, no *écran*, instantaneamente, um tabuleiro de damas. Talvez de momento não se pareça muito com um tabuleiro de damas (mais tarde corrigiremos isso); mas pelo menos está tudo no seu lugar.

Há ainda uma coisa que precisamos completar — por enquanto não podemos fazer SAVE (gravação) do código máquina que esteja guardado na parte superior da memória. Acrescentem-se a HEXLD 2 as linhas seguintes:

```
400 INPUT "SAVE 7000 ATÉ";A$
410 GO SUB 200
420 SAVE "HEXLD 2" LINE 500
430 SAVE "GRAPHIC A" CODE USR "A",8
440 SAVE "M/CODE" CODE 28672,X-28671
450 VERIFY ""
460 VERIFY "" CODE
470 VERIFY "" CODE
480 STOP
500 LOAD "" CODE USR "A",8
510 LOAD "" CODE
520 STOP
```

Para gravar o código máquina escreve-se RUN 400. Aparecerá a mensagem usual «Start tape then press any key» três vezes de seguida. Será feito automaticamente um VERIFY do programa e do

código máquina (portanto é preciso rebobinar a fita e passá-la outra vez). Como se vê, o carácter «peça das damas» é também guardado em fila pela linha 430. Nunca se deverá gravar utilizando SAVE "HEXLD 2" como é costume — é bom criar o hábito de escrever RUN 400. A verificação automática é muito útil porque não corremos o perigo de a esquecer.

A ORGANIZAÇÃO DO «ÉCRAN»

Vamos mudar um pouco de assunto. Vamos explorar o *écran* por uns momentos, guardando o programa (para o que se escreve RUN 400).

Agora, só para vermos o que acontece, peço ao leitor que escreva, como um comando directo, o seguinte:

```
FOR I=16384 TO 23295:POKE I,41:NEXT I:PAUSE 0
```

Não é um resultado curioso? Não só pelo facto de estarmos a fazer POKE no *display file* — a zona de memória onde se guarda a imagem — mas, sim, pela ordem estranha em que as coisas acontecem. Este é o mapa de memória do *Spectrum* — a princípio parece mais confuso que o necessário. Assim, penso que é conveniente examinar a organização do *écran* mais de perto para podermos compreender melhor as coisas. Repare-se na figura 7.1: dá-nos o endereço da posição de impressão para qualquer ponto do *écran* e também o endereço do *byte* de atributos correspondente. Por exemplo, considere-se o ponto cujas coordenadas PRINT AT são 5,4. De acordo com o diagrama, os primeiros dois dígitos do endereço desta posição são 40. O terceiro dígito é A (uma vez que a fila número cinco é a sexta fila no sentido descendente) e o quarto dígito é 4. Isto dá-nos o endereço 404A.

Experimente-se correr este programa em código máquina:

```
21A440      LD HL,40A4
36FF        LD (HL),FF
C9          RET
```

Agora, confirmemos que a pequena linha que apareceu no *écran* se encontra, de facto, nas coordenadas 5,4.

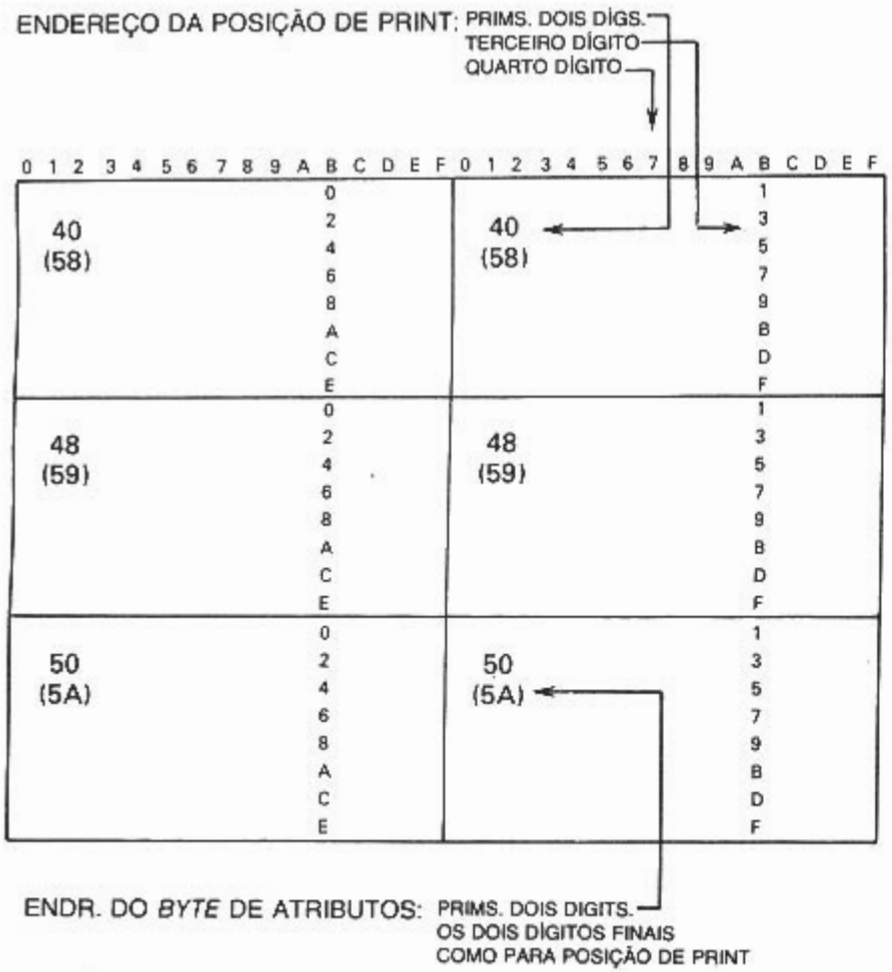


Figura 7.1

A partir do diagrama vê-se que o endereço do *byte* de atributos para essa posição é 58A4. Vejamos o seguinte programa em código máquina:


```
21A458          LD HL,58A4
36C7            LD (HL),C7
C9              RET
```

Que aconteceu? Porque usámos FF da primeira vez e C7 da segunda? Vamos explorar um pouco mais o mecanismo de impressão.

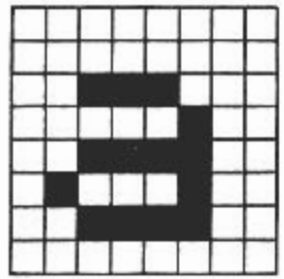
Uma parte, especialmente interessante, da ROM é a zona final — os três quartos de K que vão de 3DØØ até 3FFF. Não se trata de uma rotina mas sim de uma tabela — uma tabela muito longa, na verdade a mais longa da ROM. Guarda as matrizes de pontos de todos os caracteres, excepto os caracteres gráficos, utilizados pelo computador — os caracteres com os códigos de 2Ø a 7F. Cada carácter ocupa oito *bytes* da tabela; assim, por exemplo, os caracteres a, b e c são representados por ØØ ØØ 38 Ø4 3C ØØ, ØØ 2Ø 2Ø 3C 22 22 3C ØØ e ØØ ØØ 1C 2Ø 2Ø 2Ø 1C ØØ. Veja-se a figura 7.2.

Conseguem-se ver as letras a, b e c a partir dos dígitos da figura 7.2 O padrão é constituído pelas posições dos «uns» entre os «zeros». O diagrama também mostra como as letras ficarão quando aparecerem no *écran*. Se o código de qualquer carácter (códigos de 20 a 7F apenas) é X o endereço no qual esta «expansão em *pixels*» se guarda na ROM é 8 * X + 3CØØ e, assim, o endereço do padrão de «a» é 3F08 (uma vez que 8 * 61 + 3CØØ = 3FØ8).

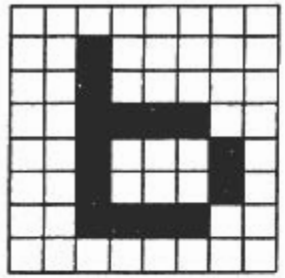
Seria agora muito vantajoso fazer uma pequena experiência. Carregue e corra o leitor este programa em código máquina. Veja se adivinha o que ele faz antes que o expliquem.

```
21A440                  LD HL,40A4      (Lembrar: esta é a
                                         posição AT 5,4.)
11083F                  LD DE,3F08      (O endereço do padrão
                                         de «a».)
0608                    LD B,08
1A          LOOP        LD A,(DE)
77                      LD (HL),A
24                      INC H           (Note-se: INC H, não
                                         INC HL)
13                      INC DE
10FA                    DJNZ LOOP
C9                      RET
```

00	0	0	0	0	0	0	0	0
00	0	0	0	0	0	0	0	0
38	0	0	1	1	1	0	0	0
04	0	0	0	0	0	1	0	0
3C	0	0	1	1	1	1	0	0
44	0	1	0	0	0	1	0	0
3C	0	0	1	1	1	1	0	0
00	0	0	0	0	0	0	0	0



00	0	0	0	0	0	0	0	0
20	0	0	1	0	0	0	0	0
20	0	0	1	0	0	0	0	0
3C	0	0	1	1	1	1	0	0
22	0	0	1	0	0	0	1	0
22	0	0	1	0	0	0	1	0
3C	0	0	1	1	1	1	0	0
00	0	0	0	0	0	0	0	0



00	0	0	0	0	0	0	0	0
00	0	0	0	0	0	0	0	0
1C	0	0	0	1	1	1	0	0
20	0	0	1	0	0	0	0	0
20	0	0	1	0	0	0	0	0
20	0	0	1	0	0	0	0	0
1C	0	0	0	1	1	1	0	0
00	0	0	0	0	0	0	0	0

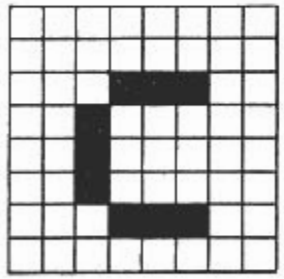


Figura 7.2

Se introduzir o programa correctamente, ao corrê-lo a letra «a» será escrita no *écran* na posição 5,4, mas sem mudar as cores nessa posição. Repare na instrução INC H. Esta é a razão pela qual a ordenação dos *bytes* no *écran* é como nos aparece. Se a posição de impressão para um quadrado tem como endereço o valor de HL, os

oito *bytes* representando a «expansão em *pixels*» de um carácter devem ser guardados nos endereços HL, HL + Ø1ØØ, HL + Ø2ØØ, HL + Ø3ØØ, HL + Ø4ØØ, HL + Ø5ØØ, HL + Ø6ØØ e HL + Ø7ØØ.

A instrução INC H, na realidade, faz o mesmo que ADD HL,Ø1ØØ, uma vez que nunca ocorre *overflow*.

Imagine-se agora que em vez do mapa de memória do *écran* ser assim, os endereços consecutivos correspondiam às linhas sucessivas de *pixels*, do canto superior esquerdo ao inferior direito na ordem supostamente lógica. Se assim acontecesse, os oito *bytes* de cada carácter teriam de ser guardados nos endereços HL, HL + ØØ2Ø, HL + ØØ4Ø, HL + ØØ6Ø, HL + ØØ8Ø, HL + ØØAØ, HL + ØØCØ e HL + ØØEØ. Uma vez que a instrução ADD HL,ØØ2Ø não existe, a única maneira de apontar HL para a posição da próxima linha de *pixels* será fazendo PUSH DELD DE,ØØ2Ø/ADD HL,DE/POP DE. Isto é muito mais aborrecido que fazer simplesmente INC H. A única razão pela qual podemos fazer INC H é pela maneira inteligente como o *écran* está construído. (Parabéns a Clive por esta faculdade.)

No entanto, com toda esta esperteza, nós ainda temos problemas pelo facto de a divisão do *écran* em segmentos (veja-se a figura 7.2) nos criar dificuldades ao passar de um segmento para outro. Por exemplo, se o endereço de um quadrado em particular for HL, o endereço do próximo quadrado no *écran* é usualmente HL + ØØØ1. Se HL contém 4ØFF, no entanto, a próxima posição de impressão terá o endereço 48ØØ — e não 41ØØ (que é o da segunda fila de *pixels* do primeiro quadrado do *écran*). Supondo que HL contém o endereço da posição corrente de impressão, a seguinte rotina irá alterá-lo de forma a afastá-lo sempre para a posição seguinte:

```
CB1C    RR H
CB1C    RR H
CB1C    RR H
23      INC HL
CB14    RL H
CB14    RL H
CB14    RL H
```

Mais ainda, a sequência de instruções RR H/RR H/ RR H/LD BC,ØØ2Ø/ADD HL,BC/RL H/RL H/RL H deixará em HL, em qualquer caso, o endereço do próximo quadrado inferior.

Consideremos agora o *byte* de atributos. A cada quadrado do tabuleiro corresponde um único *byte* de atributos (em vez dos oito *bytes* no *display file* principal). Este *byte* guarda toda a informação necessária sobre as cores desse quadrado. Encaramos este *byte* de duas maneiras. Suponhamos que a cor do fundo para o nosso quadrado é P e que a cor da tinta é I. Vamos definir ainda mais duas variáveis: FL, que será um se o quadrado piscar ou zero se o quadrado não piscar; e BR, que será um se o quadrado for brilhante ou zero se o quadrado for normal. O *byte* dos atributos requerido é então 8Ø * FL + 4Ø * BR + 8 * P + I.

No entanto isto não nos dá uma visão muito clara sobre a constituição do *byte* dos atributos. A melhor maneira é examinar este *byte* em binário.

É assim:

FL BR P2 P1 PØ I2 I1 IØ

P2, P1 e PØ representam a cor do fundo (em binário) e I2, I1 e IØ representam a cor da tinta (em binário). Repare-se que uma vez que as cores possíveis de fundo e tinta são apenas Ø, 1, 2, 3, 4, 5, 6 ou 7, só precisamos de três *bits*. Para melhor compreensão, devem escrever-se os números de zero a sete em binário e comparar os cinco *bits* mais à esquerda.

Utilizando este conhecimento dos atributos, apresentamos uma rotina, curta, que contrasta a cor da tinta com a do fundo. Partimos do princípio de que o A contém o *byte* de atributos.

```
CB6F        BIT 5,A
2803        JR Z
E6F8        AND F8          Fazer cor preta
C9          RET
F607        OR Ø7           Fazer cor branca
C9          RET
```

O *bit* número cinco de A é, na verdade, o *bit* número dois da cor do fundo, será zero apenas quando o fundo for preto, azul, vermelho ou magenta (isto é, escuro). Será um se a cor do fundo for verde, *cyan*, amarelo ou branco (isto é, claro). Vê-se, pois, como a

rotina acima dada decide se a cor de tinta deve ser preta ou branca e como é feita a mudança de cor em cada caso.

E agora alguns truques. Eis uma sub-rotina cuja finalidade é apontar HL para um novo quadrado no *display file* e DE para o *byte* correspondente na zona dos atributos. É necessário que HL já esteja apontando para a primeira fila de *pixels* de um quadrado e que DE esteja apontando para o *byte* de atributos correspondente a esse quadrado. (Veja-se mais uma vez a figura 7.1.) É preciso ainda que BC contenha o deslocamento requerido. (Contando FF como um quadrado à esquerda, ØØØ1 como um quadrado à direita, FFEØ como um quadrado acima e ØØ2Ø como um quadrado abaixo, isto é o número de quadrados entre o início e o final.) Esta sub-rotina tem uma característica muito interessante no fim:

```
7D          MOVE    LD A,L
CB1C                RR H
CB1C                RR H
CB1C                RR H
09                  ADD HL,BC
CB14                RL H
CB14                RL H
CB14                RL H
EB                  EX DE,HL
09                  ADD HL, DE
EB                  EX DE.HL
AD                  XOR L
CB67                BIT 4,A
C8                  RET Z
37                  SCF
CB5F                BIT 3,A
C8                  RET Z
7D                  LD A,L
17                  RLA
AD                  XOR L
CB67                BIT 4,A
C0                  RET NZ
37                  SCF
C9                  RET
```

A sub-rotina parte do princípio de que nunca quereremos mover mais do que sete quadrados em qualquer direcção de uma só vez.

Além do efeito sobre HL e DE, no retorno o *carry* será zero se o deslocamento for aceitável. Se o deslocamento tiver acidentalmente cruzado o extremo esquerdo ou direito do *écran*, o *carry* será um. Repare-se no uso da instrução EX DE, HL (*EXchange* — troque — DE com HL) e a maneira como foi utilizada para somar BC a DE. Este é um efeito muito útil. A lógica desta sub-rotina é a seguinte:

Se o *bit* quatro não for alterado, o deslocamento é permitido. Caso contrário...

Se o *bit* três não for alterado, o deslocamento não é permitido. Caso contrário...

Se o *bit* três for diferente do *bit* quatro o deslocamento é permitido. Caso contrário...

O deslocamento não é permitido.

Será um bom exercício ver, primeiro, como esta lógica funciona e, segundo, como esta lógica é implementada no programa.

Por fim, duas rotinas em código máquina que serão interessantes. A primeira faz *scroll* do *écran* e a segunda, a mesma operação mas em sentido contrário. Nenhuma destas duas rotinas modifica a posição de impressão (como acontece com RST 1Ø) mas ambas apagam a linha não desejada, ou no topo ou na base, conforme o caso. A figura 7.1 ajuda muito a compreender o funcionamento deste programa.

Primeira parte: fazer *scroll* para cima. USR deve indicar o endereço do *label* «UP».

```
C5          UPLD    PUSH BC
D5                  PUSH DE
E5                  PUSH HL
EDBØ                LDIR
E1                  POP HL
24                  INC H
D1                  POP DE
14                  INC D
C1                  POP BC
C9                  RET
212040      UP      LD HL,4020
110040              LD DE,4000
Ø1E01A              LD BC,1AEØ
EDBØ                LDIR
```



```
21E047              LD HL,47EØ
11E040              LD DE,40EØ
ØE20                LD C,20
3E10                LD A,1Ø
CD????      U__LOOP CALL UPLD
3D                  DEC A
20FA                JR NZ,U__LOOP
62                  LD H,D
1EE1                LD E,E1
ØB                  DEC BC
3EØ8                LD A,Ø8
36ØØ        U__BLANK LD (HL),ØØ
CD????              CALL UPLD
3D                  DEC A
20F8                JR NZ,U__BLANK
265A                LD H,5A
54                  LD D,H
3A485C              LD A,(BORDCR)
77                  LD (HL),A
EDBØ                LDIR
C9                  RET
```

Segunda parte: fazer *scroll* para baixo. USR deve indicar o endereço do *label* «DOWN».

```
C5          DOWNLD  PUSH BC
D5                  PUSH DE
E5                  PUSH HL
EDB8                LDDR
E1                  POP HL
25                  DEC H
D1                  POP DE
15                  DEC D
C1                  POP BC
C9                  RET
21DF5A      DOWN    LD HL,5ADF
11FF5A              LD DE,5AFF
Ø1EØ1A              LD BC,1AEØ
EDB8                LDDR
211F5Ø              LD HL,5Ø1F
111F57              LD DE,571F
ØE2Ø                LD C,2Ø
```

```
3E10                LD A,10
CD????    D__LOOP   CALL DOWNLD
3D                  DEC A
20FA                JR NZ,D__LOOP
62                  LD H,D
1E1E                LD E,1E
0B                  DEC BC
3E08                LD A,08
3600      D__BLANK  LD (HL),00
CD????              CALL DOWNLD
3D                  DEC A
20F8                JR NZ,D__BLANK
2658                LD H,58
54                  LD D,H
3A8D5C              LD A,(ATTR__P)
77                  LD (HL),A
EDB8                LDDR
C9                  RET
```

Repare-se na parte de código máquina no fim das rotinas que apaga as linhas não desejadas. E também na diferença entre as variáveis de sistema BORDCR e ATTR__P. O último guarda os atributos para a linha superior, enquanto BORDCR guarda os atributos para a linha inferior. O programa BASIC que se segue destina-se a mostrar ambas as rotinas de *scroll* em funcionamento. Não é um jogo terrivelmente emocionante ou um programa para produzir padrões artísticos geniais, mas mostra exactamente o que faz o código máquina de que estivemos a tratar. Podemos, é claro, inserir estas rotinas em qualquer programa; há um ou dois jogos gráficos que muito beneficiarão se os inserirmos. Este programa produz um padrão às riscas a atravessar o *écran*, com cada risca composta por um carácter ao acaso do conjunto de caracteres do *Spectrum*. O padrão no *écran* «congelará», à espera que se lhe indique o que fazer. Premindo a tecla cursor *up* faz-se com que o padrão se mova para cima, enquanto que a tecla cursor *down* fará mover o padrão para baixo. Não é necessário premir a tecla *shift*.

```
10 DIM A$(22,36)
20 FOR I=1 TO 22
30 LET B$=CHR$ (INT (96*RND)+32)
40 FOR J=0 TO 5
50 LET B$=B$+B$
60 NEXT J
70 LET A$(I)=CHR$ 16+CHR$ INT (8*RND)+CHR$ 17
   +CHR$ INT  (8*RND)+B$
80 PRINT A$(I)
90 NEXT I
100 LET A=1
110 PAUSE 0
120 LET B$=INKEY$
130 LET B=A+1:IF B=23 THEN LET B=1
140 LET C=A-1:IF C=0 THEN LET C=22
150 IF B$="6" THEN PRINT AT 0,USR down;A$(C): LET A=C
160 IF B$="7" THEN PRINT AT 21,USR up;A$(A): LET A=B
170 GO TO 110
```

Note-se a instrução INPUT"" na linha 17Ø, que tem o efeito de apagar as duas linhas inferiores do *écran*.

UM dicionário de código máquina

REGISTOS EM CÓDIGO MÁQUINA

Além dos registos A, B, C, D, E, H e L e dos *flags* PV e *carry*, existem ainda alguns outros registos e *flags*, apesar de nem todos serem úteis. Vamos falar primeiro dos registos.

IX é um par de registos; no entanto não podemos separá-lo como fazemos com HL. Todas as instruções em código máquina que envolvam HL (sem parêntesis) podem ser escritas com IX em vez de HL. (Há três excepções a esta regra: EX DE, HL, ADC HL e SBC HL,.) O código hex destas instruções é DD seguido do código da instrução correspondente usando HL. Todas as instruções em código máquina que envolvam (HL) (com parêntesis) podem ser escritas com (IX + dd) em vez de (HL) — o dd representa qualquer *byte*. Por exemplo, uma vez que há uma instrução LD (HL), Ø3, também há uma instrução LD (IX + 2A),Ø3. O *byte* de deslocamento 2A pode ser muito útil. [Há uma excepção a esta regra. A instrução correspondente a JP (HL) é JP (IX) e não JP (IX + dd).] O código hex para estas instruções é DD seguido do código hex da instrução correspondente para (HL), com o *byte* de deslocamento colocado *sempre no terceiro byte* do código hex, mesmo que isto signifique inseri-lo entre dois *bytes* do código da instrução com (HL). Por exemplo, o código hex para LD (HL),03 é 3603; portanto o código hex para LD (IX − 2A),Ø3 é DD 362AØ3.

IY é também um par de registos. Utiliza-se exactamente da mesma maneira que IX, excepto que os códigos hex das instruções com IY requerem o *byte* FD em vez de DD. Apesar do facto de ambos estes nomes de registos começarem com I, não têm a parte alta em comum e são completamente independentes um do outro.

F é o registo dos *flags*, por vezes chamado «registo de estado» *(status register)*. De vez em quando este registo coabita com A na esperança de passar despercebido.

Cada *bit* de F tem uma função específica. Já vimos alguns. São eles:

Bit 7 : o *flag* de sinal ou S
Bit 6 : o *flag* zero ou Z
Bit 5 : não é utilizado
Bit 4 : o *half carry* ou H
Bit 3 : não é utilizado
Bit 2 : o *flag* de paridade/*overflow* ou PV ou só P
Bit 1 : o *flag* de subtracção ou N
Bit Ø : o *flag carry* ou C (não confundir com o REgisto C)

O *flag half carry* testa a ocorrência de *carry* do *bit* três para o *bit* quatro no decurso das instruções aritméticas sobre quantidades de oito *bits*, ou, no caso das instruções sobre pares de registos, do *bit* onze para o *bit* doze, será um se houver *carry*, zero se não houver.

O *flag* de subtracção é afectado por qualquer instrução que envolva subtracção, caso em que se torna um, ou adição, caso em que se torna zero. Note que não há instrução que permita testar directamente estes *flags*. Tem por finalidade principal permitir a instrução DAA (ver adiante).

Não existem instruções para alterar directamente o valor de F; no entanto, podemos fazer com que F tenha um valor de, digamos, xx usando LD C,xx/PUSH BC/POP AF. Do mesmo modo, podemos ler os *flags* H e N usando PUSH AF, POP BC e examinar então os *bits* de C.

Há ainda mais um *flag* chamado IFF1. Não está incluído nos *flags* de F. Só pode ser posicionado por meio das instruções EI e DI. Sugerimos que, para mais informações sobre o assunto, se estude o seguimento neste capítulo.

SP é o ponteiro de pilha. É um par de registos como BC ou DE, no entanto, os seus dois *bytes* componentes não podem ser separados. SP aponta sempre para a parte superior da pilha. Se se alterar o valor de SP cria-se uma nova pilha no endereço dado. Endereços imediatamente abaixo do valor corrente de SP terão muito provavelmente os seus conteúdos alterados sem aviso prévio, devido a uma coisa que se chama «rotina de processamento de interrupções do *Z8Ø*» (ver DI).

A é um registo que o leitor já deve conhecer muito bem. Também se lhe chama por vezes acumulador, e tem uma ou duas utilizações que os outros registos não possuem (exemplo ADD A,Ø6).

B, C, D, E, H e L. Estes são registos com os quais já devemos estar totalmente familiarizados.

A' (repare-se na linha: A' não é o mesmo que A). É um registo útil para guardar temporariamente o valor de A quando não nos é conveniente utilizar a pilha. Só há uma instrução em código máquina que usa A' — EX AF, AF' — que troca o valor de A com A'.

F' é um registo muito estranho. Só temos acesso a ele por meio da instrução EX AF,AF'; mais nenhuma instrução lhe altera o valor. Quando se executa a instrução EX AF,AF', o registo F' é trocado pelo registo dos *flags* F. Portanto este registo não é muito útil, excepto para guardar temporariamente os valores dos *flags*.

B', C', D', E', H' e L' são registos de um só *byte*. Não têm acesso directo; no entanto, a instrução EXX troca B com B', C com C', D com D', E com E', H com H' e L com L'. Escrevemos por vezes BC', DE' e HL' para representar os pares de registos B'C', D'E' e H'L'. No *Spectrum* o par de registos HL' tem de ter o valor 2758 antes de podermos tentar o retorno a BASIC; de outro modo o *Spectrum* entrará em quebra (com o que não precisamos de preocupar-nos se não alterar HL', já que o seu valor inicial é 2758).

I e R são dois registos simples, mas nem um nem o outro têm muita utilidade. Destinam-se antes de mais ao *hardware* interno do *Spectrum* .

O CONJUNTO COMPLETO DAS INSTRUÇÕES

Já vimos até agora um bom número de instruções do *Z80*, mas vamos ampliar um pouco mais o vocabulário. Eis a lista pormenorizada de todas as instruções de que dispomos. Apresentá-la-emos por ordem alfabética de modo a que este capítulo represente uma espécie de dicionário de instruções em código máquina. Precisamente por esta razão vamos rever aquelas que já foram apresentadas anteriormente. Devem ser fixadas o melhor que pudermos.

ADC aparece de duas formas: ADC A,r e ADC HL,s. O r, aqui, representa A, B, C, D, H, L, ou uma constante numérica ou o conteúdo de um endereço (HL), (IX + d) ou (IY + d). ADC A,r é uma instrução de um só *byte*, excepto ADC A, (IX + dd) e ADC A, (IY + dd), ambas de três *bytes*, que calcula a soma A mais r mais *carry*. O resultado é guardado em A. ADC HL,s é uma instrução de dois *bytes* que calcula HL mais s mais *carry* e guarda o resultado em HL. s representa aqui BC, DE ou HL. Compreende-se porque é que, ignorando os *flags*, ADC A,A faz o mesmo que RLA?

ADD é muito semelhante a ADC, excepto que o valor inicial do *flag carry* não é utilizado no cálculo. É no entanto alterado pelo resultado da operação. Existem apesar disso duas diferenças grandes entre ADC e ADD. Primeiro o conjunto das instruções ADD HL,s (onde s tem o mesmo significado que o descrito em ADC) tem um só *byte* e em segundo é possível utilizar mais dois conjuntos de instruções ADD IX,s (em que s representa BC, DE ou IX) e ADD IY,s (neste caso s representa BC, DE ou IY).

AND só tem uma forma: AND r. O valor do registo A é alterado *bit* a *bit*. Se esse *bit* for zero, mantém-se inalterado; de outro modo toma o valor do *bit* correspondente de r. Assim, AND ØØ dá sempre zero e AND FF deixa A inalterado. AND altera cada um dos *flags*; o *flag carry*, em particular, será sempre tornado zero.

BIT é escrito sob a forma BIT n,r, onde n é um número entre zero e sete. Esta instrução altera apenas o *flag* zero, dependendo do valor do *bit* de r em questão. Se o *bit* for zero, o *flag* zero será um, se não, será zero. Este facto aproveita-se por meio de instruções como JR Z (que provoca um salto se o *bit* testado era zero), ou RET NZ (que retorna se o bit não era zero). BIT não altera o valor de nenhum dos registos, nem o do *flag carry*. É uma instrução de dois *bytes* [excepto BIT n,(IX + dd) e BIT n,(IY + dd), que são de quatro *bytes*] que apesar de não ser empregada com frequência consegue, mesmo assim, ser muito útil.

CALL já nós vimos que é semelhante a GO SUB. É uma instrução de três *bytes*. A sua acção exacta é a seguinte: PUSH o endereço de retorno (isto é, o valor corrente de PC) na pilha e depois salta para o endereço indicado.

Já que o endereço de retorno (agora na pilha) é empregado pela instrução RET, é muito importante que uma sub-rotina não altere a pilha. Só pode fazer PUSHes dentro de uma sub-rotina se em seguida fizer POPes em igual número antes de retornar. CALL também pode ser empregado com condições; por exemplo, CALL

Z, pq (pq é um endereço absoluto), que significa se o *flag* zero é um, então CALL pq; senão, prossiga com a próxima instrução.

CCF ou *complement carry flag*. Se o *flag carry* era zero, passa a ser um, se era um passa a zero.

CP escrita na forma CP r calcula o resultado da subtracção A menos r mas o resultado não é guardado em lado algum. O valor anterior de A (e, é claro, de r) mantém-se sem alterações. Por outro lado esta instrução altera todos os *flags* (tal como a subtracção, ver SUB r) e assim, instruções condicionais como JP Z ou JP C podem ser empregadas em seguida. CP r, seguida de JR Z, fará um salto se A for igual a r (uma vez que A menor que r, neste caso, é zero) e assim por diante.

CPD pode ser visto como CP (HL) seguida de DEC HL e seguida ainda por DEC BC. O *flag* PV torna-se zero ou um, conforme o valor final de BC seja igual ou diferente de zero. O *flag* zero é posicionado conforme o resultado de CP (HL) — torna-se um se A = (HL), zero caso contrário.

CPDR é basicamente o mesmo que CPD excepto que a instrução é repetida sucessivamente — por assim dizer uma espécie de ciclo automático. CPDR significa ComPare com Decremento e Repetição. O ciclo terminará num dos dois casos possíveis: 1.º ou a comparação encontra um conteúdo de (HL) igual ao de A e o *flag* zero torna-se um; note-se que neste caso (HL) aponta para o endereço anterior, isto é, A = (HL + 1); 2.º ou BC alcança zero e nesse caso o *flag* PV é *reset*.

CPI é semelhante a CPD, excepto que HL é incrementado em vez de decrescer.

CPIR é semelhante a CPDR, excepto que HL é incrementado em vez de decrescer. Neste caso se o *flag* Z termina como um, isso indica que A = (HL−1), — HL aponta para o endereço seguinte.

CPL é uma abreviatura de *complement*. O registo A é alterado *bit* por *bit*.

Se qualquer *bit* começa como zero, termina em um e vice-versa. Por exemplo se A vale 11Ø1Ø1Ø1 (binário), CPL muda-o para 0Ø1Ø1Ø1Ø/(binário).

Isto é equivalente a subtrair A de FF. Esta instrução não afecta os *flags*.

DAA é uma instrução um tanto complicada. Como exemplo, vamos admitir que queremos somar dezasseis a vinte e seis *sem* converter os números em hex. LD A,16 e depois ADD A,26 parece plausível. Infelizmente, como a máquina trabalha em hex o valor final de A será 3C e não 42. DAA (*Decimal Adjust Accumulator*, acumulador de ajuste decimal) executado imediatamente a seguir vai mudar A de 3C para 42. O seu funcionamento é bastante complicado — toma em atenção o valor dos *flag* meio *carry* e N (isto é, se a operação anterior foi adição ou subtracção) e ajusta o valor de A em função destes elementos. Isto funciona sempre. Por exemplo, a sequência LD A,42 e depois SUB Ø6 deixa A novamente com 3C mas desta vez DAA muda A para 36, já que quarenta e dois menos seis são trinta e seis. A instrução altera todos os *flags* da maneira apropriada.

DEC é mais uma instrução que apresenta duas formas: DEC r (um só registo) ou DEC s (um par de registos). DEC r é muito simples de compreender: o valor do registo r é decrementado (diminui de um ou, se o valor inicial for, ØØ, para FF), o *flag carry* não se altera e o *flag* zero, como seria de esperar, posiciona-se conforme o resultado. DEC s, no entanto, é diferente porque o *flag* zero não sofre alteração; aliás, nenhum dos *flags* é alterado. Assim, DEC BC/JR NZ,−3 é ou um ciclo infinito ou não tem efeito! É necessário lembrarmo-nos disto, para evitar que muitos dos nossos programas falhem.

DEFB não é, em rigor, uma instrução em código máquina; é o que chamamos uma directiva. A palavra DEFB tem de ser seguida por um ou mais *bytes* de dados, cada um deles separado por uma vírgula. Estes valores estão geralmente em hex, o que nem sempre é necessário. Por exemplo, DEFB 3A,C1,45d,11Ø1111Øb,"f", *amarelo*, é válido. Estes *bytes* são inseridos no programa na altura em que ocorrerem e na ordem em que estiverem escritos. Deve haver o cuidado de nunca tentar executar esses *bytes*, uma vez que o *Z80* não os sabe distinguir do programa.

DEFM é semelhante a DEFB, excepto que os dados que seguem a palavra DEFM têm de ser uma cadeia de caracteres (sem aspas). As próprias vírgulas do texto serão interpretadas como caracteres e não como separadores. Por exemplo, DEFM SOLSTICE fará com que os *bytes* 534F4C5354494345 sejam inseridos no programa. Também se podem inserir caracteres de controle numa directiva DEFM,

desde que os tornem distintos dos vulgares (isto é, se estiverem sublinhados ou em itálico) DEFM HELLO *enter* THERE e DEFM *ink blue* WHAT? são ambas válidas, apesar de ser necessário estabelecer uma convenção acerca dos espaços de cada lado das palavras de controle. É costume deixarem-se espaços por uma questão de clareza, de modo que DEFM X *enter* significaria 58ØD e DEFM X *space enter* significaria 582ØØD.

DEFM é a abreviatura de DEFina a Mensagem, em oposição a DEFB, que quer dizer DEFina *Bytes*.

DEFS é mais uma directiva. Significa DEFina eSpaço. A palavra DEFS deve ser seguida de uma constante numérica (só uma). Esta directiva provoca a inclusão no programa desse número de *bytes*, sem lhes atribuir nenhum valor concreto. Assim, DEFS Ø8 irá inserir oito *bytes* no programa mas sem querer saber o que esses *bytes* são. DEFS emprega-se essencialmente para definir «variáveis» na RAM; por exemplo, FRED DEFS Ø2 (FRED é um *label*.) e mais adiante, no programa, LD (FRED),HL.

DEFW é, por agora, a última directiva. DEFW significa «defina a palavra» *(Word)* e é usada de uma maneira muito semelhante a DEFB excepto que as quantidades a inserir têm o comprimento de dois *bytes;* por isso, DEFW 4ØØØ é equivalente a DEFB ØØ,4Ø. Repare-se em que os *bytes* foram trocados. São permitidos *labels* (etiquetas) e expressões em DEFW. DEFW 7ØØØ,FRED,ERIC+3 é válido.

DI é a abreviatura de «desligue as interrupções» *(Disable Interrupts)* e, apesar de parecer confuso, é na realidade muito simples de utilizar. No *Spectrum*, cinquenta vezes por segundo é enviado um sinal a um dos pinos, o pino INT (de INTERRUPT — interrupção) do *Z80*. Há um *flag* chamado IFF1 que significa 1.º *flip-flop* de interrupção, e o efeito de DI representa RES IFF1. Quando o *Z80* recebe um desses sinais de interrupção, verifica o valor do *flag* IFF1. Se for um, o computador age como se tivesse alcançado uma instrução RST 38 (ou CALL ØØ38), sendo o endereço de retorno colocado na pilha, o da instrução que estava para ser executada, ao ser recebida a interrupção. Se, no entanto, IFF1 for zero, o sinal de interrupção não tem efeito e a execução do programa prossegue normalmente. O *flag* IFF1 deve estar *set* antes de tentarmos retornar ao BASIC, caso contrário o *Spectrum* não responderá quando as teclas forem premidas.

DJNZ é a abreviatura de *decrement* B e *Jump* se não zero. Assim, se B é 7, DJNZ reduzi-lo-á a 6 e depois dará um salto para o endereço de destino. Se B é Ø, DJNZ torná-lo-á FF e mais uma vez haverá um salto. Se B for um, DJNZ torná-lo-á zero e não dará qualquer salto, antes a execução continuará com a próxima instrução. A forma desta instrução é DJNZ e, sendo e um só *byte*. Se B é decrementado para zero, e é ignorado. Se não, e indicará o comprimento de salto. O deslocamento calcula-se como em JR. (Notar que esta instrução não afecta os *flags*.)

EI significa «ligue as interrupções» *(Enable Interrupts)* e é o oposto de DI. Esta instrução é equivalente a SET IFF1. Veja-se DI para uma explicação mais completa[1].

EQU é a abreviatura de *EQUate* (iguale). Não se trata de uma instrução de código máquina mas sim de uma directiva. Cada directiva EQU necessita de ser acompanhada de uma etiqueta e a palavra EQU tem de ser seguida de um número (dentro do limite de ØØØØ a FFFF) ou uma expressão como ERIC+2. Esta directiva não produz a inserção de quaisquer *bytes* no programa, portanto não interessa em que parte do programa EQU aparece, apesar de estar convencionado colocarem-se todas as directivas EQU no princípio do programa. O seu efeito é atribuir um valor numérico (o valor dado) à etiqueta que a acompanha. Por outras palavras, se tiver ANNIE EQU 9ØØØ e depois, mais tarde, LD HL,(ANNIE) a instrução será compilada como LD HL,(9ØØØ).

EX é a abreviatura de *EXchange* (troque). Esta instrução provoca a troca dos valores contidos em certos pares de registos. Existem cinco instruções EX — são elas EX AF,AF';EX DE,HL;EX (SP),HL;EX (SP),IX e EX (SP),IY. Estas instruções não alteram nenhum dos *flags*. EX DE,HL substitui DE pelo valor inicial de HL e HL pelo valor inicial de DE. As últimas três são bastante interessantes — o valor anterior de HL (ou IX ou IY) é trocado com o valor no topo da pilha, logo LD BC,0123/PUSH BC/LD HL,4567/EX (SP),HL/POP BC deixará BC com 4567 e HL com 0123. EX (SP), HL não altera o ponteiro da pilha, SP e claro está, EX (SP),IX ou EX (SP),IY também não. É preciso ter em atenção que após a execução de EX AF,AF' se troca o conjunto de *flags* activo. Assim, os *flags* anteriores, embora não modificados, não são acessíveis. Por exemplo, a sequência CP 1A/EX AF,AF'/JP Z,8ØØØ não actua em função da comparação inicial, mas do valor do

flag Z inicialmente em F'.

EXX representa EX BC,B'C' seguida de EX DE,D'E' seguida ainda de EX HL,H'L'. Basicamente, cada um dos registos comuns (excepto A) é trocado com o registo correspondente alternativo. Tome-se atenção, no entanto, a que o par de registos HL' tem de ter o valor 2758 antes de retornar ao BASIC e, assim, se o alterar de qualquer maneira, deve voltar a carregá-lo com esse valor.

HALT é uma instrução que, uma vez executada, suspende a continuação do programa até ser recebida a próxima interrupção. Quando isto acontece, a instrução RST 38 (CALL ØØ38) será executada (no *Spectrum*) sendo o endereço de retorno o da primeira instrução a seguir a HALT. Note-se que o *flag* IFF1 tem de estar *set* se se quiser executar HALT sem perigo, senão o processador nunca reconhecerá o sinal de interrupção. Se isto acontecer, HALT provocará a paragem para sempre. Não haverá saída dessa situação a não ser que desliguemos a ficha. O efeito de HALT, naquilo a que se refere a nós, é esperar pelo início da emissão do próximo quadro de imagem para a TV, mais ou menos equivalente a PAUSE 1 em BASIC. Veja-se DI para uma explicação sobre o uso de IFF1.

IM. Perigo! Em circunstância alguma utilize esta instrução[2].

IN é bastante semelhante à instrução IN do BASIC. Apresenta duas formas. A primeira é IN A,(n), onde n é uma constante numérica. Esta é quivalente à instrução BASIC LET A=IN (256*A+n). A segunda forma é IN r,(C), onde r é um registo. Equivale à instrução BASIC LET r=IN (256*B+C). O argumento de IN refere-se a um dispositivo *hardware* exterior ao próprio processador *Z80* — um número diferente para cada dispositivo diferente. Na forma IN A,(n) os *flags* não se alteram; mas são alterados por IN r,(C) excepto o *carry*, que não é afectado.

INC. Já conhecemos esta instrução. INC r aumenta o valor do registo r de um, posicionando os *flags* excepto o *flag carry*, que não se altera. INC s aumenta o valor de s de um mas não altera nenhum *flag*.

IND significa IN com Decremento. Podemos pensar em IND como IN (HL),(C) seguida de DEC HL seguida de DEC B. O *flag carry* não é alterado mas o *flag* zero reflecte o novo valor de B.

INDR é como IND, mas a instrução repete-se as vezes necessárias para B atingir zero.

INI é como IND excepto que HL é incrementado em vez de decrescer.

INIR é como INDR excepto que HL é incrementado em vez de decrescer.

JP é o código máquina equivalente a GO TO. Se compreendemos GO TO 10 também compreendemos JP 73ØØ. O destino é um endereço, não um número de linha, mas o princípio é exactamente o mesmo. Dispomos também de saltos condicionais, por exemplo JP NZ,7300 significa IF não zero THEN salte para o endereço 7300 (por outras palavras se o zero *flag* não estiver *set*). Há uma outra forma de JP que também tem uma análoga no BASIC do *Spectrum*: saltos com destinos variáveis. Se compreendemos GO TO N também compreendemos JP (HL). JP (HL) significa GO TO HL. Nesta forma não pode haver condições: por exemplo, JP NC,(HL) não é possível. Apenas três pares de registos podem ser empregados como destinos variáveis: são eles HL, IX e IY. Representam, mesmo assim, instruções muito poderosas: HL pode conter o resultado de um cálculo, talvez mesmo gerado ao acaso.

JR é idêntica a JP, apenas um pouco menos forte e com um *byte* menos. Só quatro das oito condições podem ser empregadas: Z, NZ, C e NC. Isto significa, por exemplo, que JR PO é impossível. É também impossível JR (HL). JR não utiliza um endereço absoluto, o R significa relativo. Escrevemos a instrução como JR (ou JR Z, ou qualquer outra condição permitida), onde e é uma constante de *byte* que nos indica o número de *bytes* a saltar. JR Ø não tem efeito, uma vez que faz um salto para a frente zero *bytes*. JR FE é um ciclo infinito porque o controle saltará para a própria instrução JR FE. O deslocamento começa a contar a partir da instrução imediatamente a seguir à instrução JR e. Se o *byte* se encontra entre 8Ø e FF, o salto é para trás. Veja-se a tabela número 2 do Apêndice 1.

LD é a instrução mais utilizada em código máquina . Transfere informação de um lugar para outro. Apresenta muitas formas, a mais simples das quais é LD r1,r2, que transfere *bytes* de um registo para outro. Outras formas: LD A,(BC); LD A,(DE) e LD A,(HL); e os opostos LD (BC),A; LD (DE),A e LD (HL),A. Recordemos que os parêntesis significam o conteúdo de um endereço. Os registos simples (excepto I e R) e os pares de registos podem ser carregados com constantes numéricas, os pares de registos com o conteúdo de

qualquer endereço e inversamente todos os endereços podem ser carregados a partir de um par de registos (note-se que os pares de registos guardam dois *bytes*, não um, e estes são transferidos de e para o endereço indicado na instrução e o endereço imediatamente superior). Também é possível fazer LD A,(pq) e LD (pq),A (onde pq representa um endereço) e SP pode ser carregado a partir de HL, IX, iY ou (pq). (pq pode ser carregado a partir de SP mas HL, IX e IY não). Por outras palavras: não podemos, por exemplo, dizer LD HL,DE (é preciso fazer LD H,D, LD L,E ou vice-versa). Felizmente, uma vez que LD se emprega com tanta frequência, é muito fácil familiarizarmos-nos com as suas muitas formas. Nenhuma destas instruções tem qualquer efeito sobre os *flags*. Existem ainda as seguintes instruções envolvendo os registos I e R: LD I,A; LD R,A; LD A,I e LD A,R. As duas primeiras permitem carregar os registos I e R a partir de A. As duas últimas permitem obter em A os conteúdos de I e de R. Estas duas instruções são as únicas instruções LD a alterar os *flags*, excepto *carry*, que não é afectado. Em particular o *flag* P/v toma o valor do *flag* IFF[3].

LDD significa *LoaD* com Decremento. É efectivamente LD (DE),(HL) seguida de DEC HL, DEC DE e DEC BC todos de uma só vez. O *flag carry* e o *flag* zero não se alteram, assim como o *flag sign;* o *flag* PV, no entanto, torna-se zero se e apenas se o valor final de BC for zero. Logo, JP PO só fará o salto se BC for zero a seguir à instrução.

LDDR é semelhante a LDD mas a instrução é repetida até BC atingir zero.

LDI é semelhante a LDD excepto que DE e HL são ambos incrementados em vez de decrementados. BC ainda decresce.

LDIR é semelhente a LDI mas a instrução repete-se até BC atingir zero.

NEG significa torne NEGativo o acumulador (ou registo A). Funciona executando a subtracção ØØ menos A e todos os *flags* são alterados de maneira correspondente. Assim, S reflecte o sinal do resultado. Z será um se e apenas A for zero. NEG é equivalente a CPL seguida de INC A (ignorando os *flags*).

NOP é a abreviatura de Nenhuma OPeração. Tem uma função muito simples, que é consumir tempo. NOP não faz nada. Emprega-se para introduzir atrasos ou para substituir código máquina anterior quando corrigimos um programa. O seu equivalente BASIC mais próximo seria um REM em branco. Esta instrução é praticamente indispensável.

OR, apresentado na forma OR r, é quase o oposto a AND r. O valor do registo A é mudado *bit* por *bit*. Se um *bit* for um, permanece inalterado, senão toma como valor o do *bit* correspondente de r. Se A contém ØØ logo (ignorando os *flags*) OR r é o mesmo que LD A,r. OR FF é na realidade LD A,FF. Todos os *flags* mudarão conforme o resultado, PV testa a paridade e o *carry* torna-se zero.

ORG é uma directiva que não pode ser acompanhada de uma etiqueta. A palavra ORG deve ser seguida de um número entre ØØØØ e FFFF. Isto quer dizer que todo o código máquina que se seguir deve ser escrito a partir do endereço dado. Assim, ORG 7ØØØ seguido de LD A,Ø1 significa que a instrução LD A,Ø1 irá para o endereço 7ØØØ. A não ser que a próxima instrução seja outra vez uma directiva ORG, a instrução seguinte será depositada no endereço 7ØØ2 (uma vez que LD A,Ø1 é uma instrução de dois *bytes).*

OUT já nos é familiar do BASIC. Está escrita a vermelho na tecla marcada «O». O código máquina para a instrução OUT tem duas formas. A primeira é OUT (n),A — que é equivalente à instrução OUT 256*6+n,A do BASIC. A segunda é OUT (C),r que é equivalente em BASIC a OUT 256*B+C,r. OUT retira números do interior do *Z80* e envia-os para o *hardware* exterior. Não tem efeito nos *flags*.

OUTD significa OUT com Decremento. É equivalente a OUT (C),(HL) seguida de DEC HL seguida de DEC B. O *flag carry* não é alterado, mas o *flag* zero reflecte o novo valor de B.

OTDR significa OUT com Decremento e Repetição. Mudámos a ordem alfabética para manter a consistência. É equivalente a OUTD repetida até que B seja zero.

OUTI é semelhante a OUTD excepto que HL é incrementado em vez de decrescer.

OTIR é semelhante a OTDR excepto que HL é incrementado em vez de decrescer.

POP retira dois *bytes* do topo da pilha e carrega-os num par de registos. Os pares de registos BC, DE, HL, IX e IY podem ser utilizados. A instrução POP AF pode ainda ser utilizada, sendo AF um pseudo-registo construído a partir do acumulador (parte alta) e

dos *flags* (parte baixa). A acção de POP é a seguinte: a parte baixa do registo de destino é carregada com o conteúdo do endereço apontado por SP; seguidamente, SP é incrementado e o conteúdo do endereço apontado é carregado na parte alta do registo de destino; SP é incrementado uma segunda vez, de forma que no final o seu valor é de duas unidades superior ao valor inicial. (Note-se que RET é equivalente a POP BC.)

PUSH é o oposto de POP. Guarda o conteúdo de um qualquer dos pares de registo AF, BC, DE, HL, IX ou IY. A parte alta do registo é guardada em primeiro lugar, seguindo-se-lhe a parte baixa. O valor de SP é diminuído em duas unidades, apontando para a parte baixa do valor no topo da pilha. (PUSH HL, por exemplo, é equivalente a DEC SP/LD (SP),H/DEC SP/LD (SP), L.)

RES é a abreviatura de RESet, que significa «torne zero», portanto RES é a instrução que posiciona a zero qualquer *bit* de um registo ou posição apontada por (HL),(IX+dd) ou (IY+dd). Com esta instrução alteraremos *bits* individuais de qualquer registo. Por exemplo, para fazer o *bit* 3 de D igual a zero, bastará dizer RES 3,D. RES não afecta nenhum dos *flags*.

RET emprega-se para retornar de uma sub-rotina. Funciona fazendo POP de um endereço no topo da pilha para o contador de programa, provocando portanto um salto para esse endereço. É possível alterar o endereço para o qual uma sub-rotina retornará, alterando o valor do topo da pilha. Por exemplo, POP HL/INC HL/PUSH HL aumenta o endereço de retorno de um. Podemos, por exemplo, guardar um *byte* de dados imediatamente a seguir à instrução CALL e, dentro da sub-rotina, fazer POP HL/LD A, (HL)/INC HL/PUSH HL, que vai guardar esse *byte* em A ao mesmo tempo que assegura o retorno da sub-rotina ao endereço a seguir ao *byte* referido. Um outro truque é fazer *push* de um endereço de retorno «artificial» na pilha e então JP (ou JR) para uma sub-rotina, em vez de CALL. Agora, retorna-se a qualquer endereço! Se necessário, RETorne emprega-se com condições. Não altera os *flags*.

RETI não pode ser aplicado ao *Spectrum*[4].

RETN não pode ser aplicado ao *Spectrum*[5].

RL tem a forma de RL r. Cada *bit* do registo especificado move-se uma posição para a esquerda. O *bit* mais à esquerda muda para o

carry e o *bit* mais à direita toma o valor anterior do *carry*. Assim roda à esquerda *(left)*. Por exemplo, se B contiver 10010101 e o *carry* contiver zero, RL B deixa B com 00101010 e o *carry* com um. RL altera todos os *flags*. O *flag* PV testa a paridade do resultado.

RLA (note-se que não há espaço entre o L e o A). RLA é um modo mais eficiente de utilizar RL A! Esta instrução tem menos um *byte* e só altera o *flag carry*.

RLC significa Rode à esquerda *(Left)* sem *Carry*. É quase o mesmo que RL r. De facto, cada *bit* do registo em questão é movido uma posição para a esquerda. Aqui, no entanto, o *bit* inicialmente mais à esquerda torna-se tanto o novo valor de *carry* como o novo *bit* mais à direita. O anterior valor de *carry* não entra neste processo e todos os *flags* são alterados.

RLCA é uma instrução de um *byte* em vez de dois (ou quatro). RLCA é semelhante a RLC A, só que mais rápida; além disso, só o *flag carry* é alterado pela instrução.

RLD não deve confundir-se com o RL D, porque é completamente diferente. Funciona da seguinte maneira: escreve-se o valor de A e o conteúdo do endereço (HL) em hex. O segundo dígito — o da direita — hex de (HL) muda para a esquerda, de maneira que se torna o novo primeiro dígito de (HL). O valor anterior deste primeiro dígito substitui o segundo dígito — o da direita — de A, que por seu turno se torna no segundo dígito de (HL). Assim, se começarmos com A contendo 25 e (HL) contendo A3, RLD mudará as coisas para A=2A,(HL)=35. RLD é a abreviatura de Rode à esquerda *(Left)* Decimal.

RR é como RL, excepto que os *bits* se movem para a direita em vez de para a esquerda.

RRA é como RLA excepto que os *bits* se movem para a direita em vez de para a esquerda.

RRC é como RLC excepto que os *bits* se movem para a direita em vez de para a esquerda.

5RRCA é como RLCA excepto que os *bits* se movem para a direita em vez de para a esquerda.

RRD é como RLD excepto que os dígitos hex se movem para a direita em vez de para a esquerda.

RST é o mesmo que CALL excepto que esta instrução só tem um *byte* de comprimento! É muito menos forte, no entanto, por duas razões: primeira, não podemos usar condições (por exemplo RST 1Ø é possível mas RST NZ,1Ø não é); e segunda, o RST conduz-nos apenas a um dos oito endereços seguintes: ØØ, Ø8, 1Ø, 18, 2Ø, 28, 3Ø e 38. Uma vez que o *Spectrum* começa a executar a ROM a partir do endereço ØØØØ para a frente, RST ØØ tem o mesmo efeito que desligar e tornar a ligar a ficha.

SBC, tal como ADC, tem duas formas. A primeira é SBC A,r, que primeiro subtrai r de A e depois o *carry*. Identicamente, SBC HL,s subtrai tanto s como o *flag carry* de HL. SBC A,A é uma instrução muito útil — deixa o *carry* sem alterações mas altera A para ØØ se inicialmente não houver *carry* ou para FF se houver *carry*.

SCF significa *Set Carry Flag*. Todos os outros *flags* ficam inalterados.

SET é o oposto de RES. SET 4,H tornará um o *bit* 4 do registo H (por exemplo). Qualquer *bit* de qualquer registo ou posição de memória apontada por (HL),(IX+dd) ou (IY+dd) pode ser *set*.

SLA significa SHIFT à esquerda *(Left)* Aritmético. A forma desta instrução é SLA r. É semelhante a RL r excepto que o *bit* mais à direita é sempre substituído por zero. SLA r tem por efeito multiplicar o registo r por dois.

SRA quer dizer *Shift* à direita Aritmético. Qualquer registo ou posição de memória apontada por (HL),(IX+dd) ou (IY+dd) pode ser deslocado para a direita utilizando a forma SRA r. A instrução é semelhante a RR r excepto que o *bit* mais à esquerda se mantém inalterado. SRA r divide o conteúdo do registo r por dois, se esse número for visto como um número com sinal (na forma de complemento a dois — entre −128 e +127). (Apêndice 1, tabela 2.)

SRL quer dizer *Shift* à diReita Lógico. Mais uma vez, semelhante a RR excepto que aqui o *bit* mais à esquerda é sempre substituído por zero. SRL r divide o número contido no registo r por dois se esse número for considerado como número sem sinal (isto é, entre Ø e 225). (Apêndice Um, tabela 1.)

SUB escreve-se como SUB r (mas às vezes também como SUB A,r). Esta instrução subtrai r do registo A. Note-se que, ao contrário de ADD, não há instrução correspondente SUB HL,s. Querendo subtrair s de HL, temos antes de mais nada de pôr a zero o *flag carry* (geralmente através da instrução AND A) e então usar SBC HL,s.

XOR, que tem a forma XOR r, muda o valor de A *bit* por *bit*. Se qualquer *bit* de A é idêntico ao *bit* correspondente de r, esse *bit* torna-se igual a zero, de outro modo torna-se um. XOR altera todos os *flags* e o *flag carry* é sempre *reset*. Note-se que XOR FF é o mesmo que CPL (ignorando também os *flags*).

$, a directiva final[6]. Na verdade não é exactamente uma directiva, é só um símbolo especial que se emprega, por exemplo, numa directiva EQU. No entanto, se o cifrão for usado numa directiva EQU, a própria palavra EQU pode ser omitida! $ apenas significa o endereço do próximo *byte;* assim, FRED $+2 (que é uma maneira encurtada de escrever FRED EQU $+2) onde FRED é uma etiqueta, "define FRED como significando o próximo endereço mais dois". Um exemplo da sua utilização pode ser: LD (HL), ØØ/CHEAT $+1 (*cheat* é uma etiqueta) e, mais tarde, LD (CHEAT),A, que forma o princípio de um programa auto-alterável. Percebe-se, claro, que a etiqueta *cheat* foi definida como significando o endereço do segundo *byte* da instrução LD (HL),ØØ. Interessante, não é?

CAPÍTULO NOVE

Mais lugares onde guardar código máquina

Guardar código máquina acima de RAMTOP, como fizemos até agora, protege-o de ser apagado por NEW ou sobreposto por um programa, mas tem também a desvantagem de ser necessário fazer dois SAVES separados se quisermos guardar tanto o código máquina como algum programa de suporte em BASIC. Existem várias possibilidades alternativas que vamos explorar neste capítulo.

UTILIZANDO REM

Um lugar onde podemos guardar o código máquina é numa linha REM. Se num programa BASIC fizermos 1 REM seguido de um certo número de *bytes*, pode-se guardar código máquina no lugar desses *bytes* — isto é, o código máquina pode ser guardado entre a palavra REM e o fim da linha um. Vejamos como isto é possível. Admitindo que queremos guardar uma rotina de código máquina que tem o comprimento de cinquenta *bytes*, fazemos a primeira linha do programa:

```
1    REM 12345678901234567890123456789012345678901234567890
```

É uma REM com cinquenta *bytes* de comprimento depois da palavra REM. Se a rotina tivesse sessenta *bytes* de comprimento, precisaria de sessenta caracteres; se só tivesse três *bytes*, só precisaria de três caracteres a seguir à palavra REM. Não importa que caracteres são, mas contar de um a um para cima é uma boa maneira de não nos enganarmos a meio da contagem.

Aqui, querendo, utilizaremos HEXLD 2, mas para isso temos de lhe fazer algumas alterações. Primeiro, no entanto, só uma palavra ou duas de explicação. A linha dois de BASIC deve ser:


```
2 DEF FN J(X)=PEEK 23635+256*PEEK 23636+5+X
```

Com isto, a função FN J(X) calculará o endereço do carácter número (X+1) a seguir à palavra REM — o que é muito útil saber, como é óbvio. Se juntarmos então só mais uma linha a HEXLD 2:

```
255 LET X=FN J(X)
```

estaremos prontos. O número hex a introduzir como endereço inicial deve ser agora não um endereço absoluto mas um deslocamento em relação ao primeiro carácter a seguir à palavra REM de maneira que introduzindo "0000" se referirá ao primeiro carácter. Agora podemos introduzir um programa de código máquina como antes e será escrito, não na RAM vaga mas sim na REM linha um. Este código máquina correrá por meio de RUN 700 (usando HEXLD 2) ou como um comando directo: RANDOMIZE USR FN J(0).

Uma vez que esse programa em código máquina esteja em condições de funcionar, apaga-se o programa totalmente, excepto as linhas um e dois. Isto deverá fazer-se não através da introdução de NEW, mas pela introdução dos números de linha, um de cada vez. O programa em código máquina pode ser corrido por meio da instrução RANDOMIZE USR FN J(0) do BASIC. Ou, se quisermos corrê-lo a partir do quinto *byte*, por exemplo, RANDOMIZE USR FN J(4).

Há vantagens e desvantagens no uso de REMs — estudá-las-emos uma a uma. Primeiro as desvantagens.

A primeira desvantagem é que a linha um nunca pode ser listada — pelo menos não deve! Algumas instruções em código máquina farão com que a listagem seja «sabotada». Por exemplo, LD B,n (hex 06) provocará espaços estranhos, DEC C (hex 0D) produzirá efeitos curiosos, DJNZ e LD DE,mn frequentemente provocarão a mensagem «error K Invalid colour» e assim por diante. O resto do programa é facilmente listado (utilizando LIST 2, por exemplo).

Não se pode, numa REM destas, referir qualquer endereço absoluto dentro do programa em código máquina, porque o programa BASIC tem tendência a mover-se para cima e para baixo na memória se, por exemplo, o *Spectrum* estiver ligado a uma

microdrive Sinclair. FN J(Ø) pode dar respostas diferentes em diferentes ocasiões.

Outra desvantagem é que o comando NEW apaga-o.

A vantagem, no entanto, é que o código máquina faz parte integrante do programa BASIC. Assim, carrega-se ou guarda-se em fita simultaneamente com ele e não é necessária nenhuma instrução LOAD separada. VERIFY"" verificará tanto o programa como o código máquina de uma só vez. Ao usar REM para guardar código máquina surge por vezes o seguinte problema aflitivo: que fazer quando, por exemplo, depois de introduzir uma longa sequência de códigos, verificamos que não há caracteres suficientes a seguir à palavra REM para guardar o nosso código máquina completo? Se algum do nosso código máquina já aí se encontra, não nos é possível prolongar o comprimento da linha por meio de EDIT, como seria de esperar.

Praticamente todas as vezes que aparecer o carácter 1Ø (e mais alguns outros) ouvir-se-á um besouro irritante se tentarmos editar a linha, que será invisível a partir desse ponto. Qualquer ocorrência do *byte* OE na linha original será apagada automaticamente pela ROM, assim como os cinco primeiros *bytes* a seguir a ØE. Na listagem (se conseguirmos ter uma listagem) o *byte* ØE e os cinco primeiros *bytes* que o seguem são invisíveis, mas na verdade eles estão lá — se usarmos EDIT, por outro lado, eles desaparecerão sem deixar rasto.

O *Spectrum* emprega o *byte* ØE para dizer «segue-se um número em vírgula flutuante». Sempre que se usar um número decimal ou binário (com BIN) na listagem de um programa, a ROM automaticamente fará seguir este número por um *byte* ØE, seguido de cinco outros *bytes* que conterão o número, convertido em vírgula flutuante. Tanto o *byte* ØE como os cinco *bytes* que o seguem não são visíveis na listagem. É isto que causa tantos problemas em editar REMs com código máquina.

O único meio prático de prolongar o comprimento de uma REM é juntar duas ou mais. No princípio de cada linha de programa existem dois *bytes* invisíveis que guardam o comprimento dessa linha e portanto é necessário modificar-lhes os valores para conseguirmos o efeito desejado. A pequena rotina que se segue permite aumentar o comprimento de uma REM na linha um.

O primeiro passo é mudar o número de linha de DEF FN J para qualquer valor excepto um ou dois e então juntar uma nova linha dois, consistindo da palavra REM seguida de um número de caracteres arbitrário. Em qualquer altura do programa inserem-se as seguintes seis linhas (de qualquer modo, elas em breve serão apagadas):

```
LET B=FN J(-3)
LET A=B+PEEK B+256*PEEK (B+1)+4
LET A=A+PEEK A+256*PEEK (A+1)-B
POKE B,A-256*INT (A/256)
POKE B+1,INT (A/256)
STOP
```

A linha LET B=FN J (−3) pode, é claro, ser substituída por LET B=PEEK 23635+256*PEEK 23636+2. Correndo esta rotina, a linha dois tornar-se-á automaticamente parte da linha um. Pode-se agora apagar a rotina — a sua função já está cumprida. LISTe a linha um — a linha dois ainda parece estar separada, mas experimente-se deslocar o cursor para baixo: ele salta por cima da linha dois. Experimente-se apagar a linha dois, escrevendo o seu número de linha — ver-se-á que não se consegue porque agora o computador não reconhece a sua existência! Independentemente do aspecto da listagem a ROM ignorará por completo a linha dois, tomando-a como parte da linha um. Pode-se agora escrever por cima do marcador de fim de linha (o caracter ØD), no fim da linha um, sem sofrer más consequências.

Contrariamente, a seguinte rotina encurtará uma REM pelo menos em seis *bytes:*

```
LET A=FN J(-3)
LET B=PEEK A+256*PEEK (A+1)
LET C= o número de bytes que se deseja preservar mais 2
POKE A,C-256*INT (C/256)
POKE A+1,INT (C/256)
LET A=A+C+1
LET C=B-C-4
POKE A,13
POKE A+1,Ø
```

```
POKE A+2,2
POKE A+3,C-256*INT (C/256)
POKE A+4,INT (C/256)
STOP
```

Corre-se de novo a rotina e apaga-se. Agora LISTe o programa e vê-se aparecer uma nova linha dois. Apagando-a ao escrever o seu número de linha, a REM será tão curta quanto precisarmos que o seja.

USO DA ÁREA DE MEMÓRIA DESTINADA AO PROGRAMA

A figura 9.2 mostra a área de RAM a que nos referimos. As vantagens de utilizar esta área são as mesmas que para a REM. Há, no entanto, mais uma vantagem, a de não precisarmos de nos preocupar com a listagem. O código máquina será totalmente invisível mas mesmo assim formará parte integrante do programa.

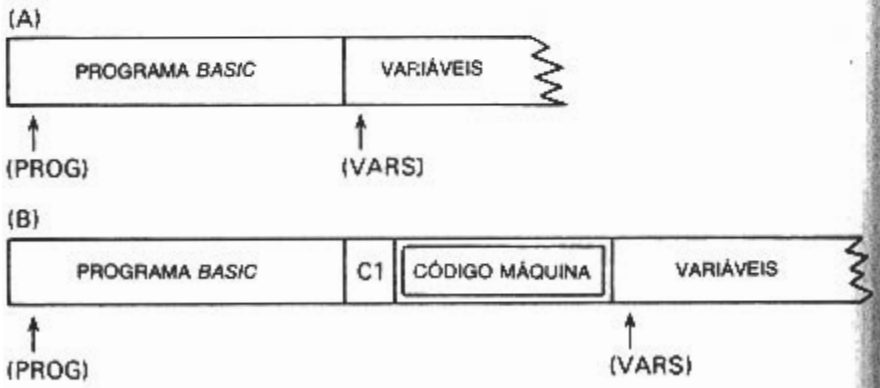


Figura 9.1

Para estabelecer esta área com, digamos, cinquenta *bytes*, é preciso fazer o seguinte:


```
CLEAR
DIM A$(45)                                  45=50 menos 5 (decimal)
LET A=PEEK 23627+256*PEEK 23628+51
                                            51=50 mais 1 (decimal)
LET B=INT (A/256)
PRINT B
POKE 23627,A-256*B
POKE 23628, o número impresso
```

O número na instrução DIM deve ser sempre menos cinco que o número de *bytes* que se deseja guardar. O número no fim da instrução seguinte é mais um que o números de *bytes* a guardar. O processo está agora completo e só precisamos de um meio de acesso ao código máquina. Isto consegue-se juntando a seguinte instrução BASIC ao programa BASIC (qualquer número de linha servirá):

```
DEF FN J(X)=PEEK 23627+256*PEEK 23628-5Ø+X
```

O número cinquenta refere-se ao facto de termos reservado cinquenta *bytes*. De desejássemos reservar cem *bytes*, a instrução DIM anterior seria DIM A$(95), a próxima instrução deveria terminar com +1Ø1 e a linha DEF FN deveria terminar com −1ØØ + X. FN J funcionará do mesmo modo que anteriormente para RM, de modo que FN J(Ø) dá o endereço do primeiro *byte* do código máquina, FN J(1) o endereço do segundo *byte* e assim por diante até FN J(49) para o último *byte* (ou 99, se reservarmos cem *bytes*, etc.).

O código máquina guardado nesta área será carregado, guardado e verificado juntamente com o programa BASIC; assim, só precisaremos de um SAVE. Devemos dar uma explicação a propósito do *byte* C1 no diagrama [figura 9.2(B)]. Não é preciso introduzi-lo directamente; é lá colocado automaticamente, pelo processo que utilizámos. C1 é o *byte* que o computador usa para representar o nome da matriz *string* A$. Se tivéssemos utilizado DIM B$ em vez de DIM A$, este *byte* seria C2. Não interessa qual o *byte* que se emprega para separar o BASIC do código máquina, desde que esse *byte* seja maior ou igual a hex 40.

PASSAGEM DE PARÂMETROS PARA ROTINAS EM CÓDIGO MÁQUINA

Vamos mudar de assunto. Vimos já um uso de FN e vamos ver agora mais outro; considere-se a instrução:

DEF FN U(X) = USR 28672.

(Se existe uma rotina em código máquina guardada na RAM a partir do endereço 28672.)

É interessante, não? Claro que o número em causa não precisa de ser 28672 — só é necessário que seja o endereço de um programa em código máquina. Agora, sempre que utilizamos FN U, chamaremos o programa de código máquina. Até agora tudo bem, mas não tirámos nenhuma vantagem em relação ao simples uso de USR. Mas agora pense-se no "X" entre parêntesis — o que lhe acontece?

Uma das variáveis de sistema é chamada DEFADD e guarda o endereço dos argumentos de um cálculo FN. Estes argumentos são guardados de um modo muito estranho (aliás como sempre). Em vez de passarmos horas em explicações, o melhor será entrarmos pelo fim sem hesitar e tentarmos então compreender.

Considere-se esta instrução BASIC:

DEF FN A(A,B,C,A$,B$,C$) = USR *um número*.

Há seis argumentos entre parêntesis e assim a ROM guarda seis registos destes valores. O endereço do primeiro registo é indicado pela variável de sistema DEFADD. Veja-se agora a figura 9.3, que mostra como estes registos estão organizados. Registos numéricos ocupam oito *bytes*, e registos *string* ocupam nove *bytes*. No caso de registos numéricos o primeiro *byte* é o nome do argumento, como aparece em DEF FN; o segundo *byte* é sempre o *byte* ØE. Os próximos cinco *bytes* guardam o número calculado entre os parêntesis de FN — por exemplo, FN A (3 + 3), etc., é permitido mas o resultado calculado é seis. No entanto, se este resultado estiver dentro dos limites hex ØØØØ a FFFF e for positivo, podemos ler o seu valor directamente como o quinto e sexto *bytes* do registo — o terceiro, quarto e sétimo *bytes* serão sempre zero e o oitavo *byte* será sempre 2C (vírgula) a não ser que o registo seja o último do conjunto, caso em que o oitavo *byte* será 29 (fechar parêntesis da

direita). Os registos de *string* são igualmente simples — os primeiros dois *bytes* guardam o nome do registo tal como aparece em DEF N; no entanto, uma vez que o nome acaba sempre em cifrão (ou dólar) então o segundo *byte* é sempre 24 (cifrão); o terceiro *byte* conterá sempre o valor ØE, o quarto *byte* não é utilizado e conterá «lixo», o quinto e o sexto *bytes* conterão o endereço a partir do qual a *string* resultante da avaliação da expressão em DEF FN efectivamente começa, os sétimo e oitavo *bytes* guardam o comprimento da *string* e finalmente o nono *byte* guarda a vírgula ou parêntesis da direita e dependendo de o registo ser ou não o último no grupo.

O essencial disto tudo é que DEF FN U(X) = USR qualquer coisa em conjunto com, digamos, FN U(X+Y) irá chamar uma rotina de código máquina. O valor de X + Y pode ser passado para a rotina em código máquina por LD HL,(DEFADD)/INC HL/INC HL/INC HL/INC HL/LD C, (HL)/IN LD B,(HL).

DISPOSIÇÃO DOS REGISTOS PARA FN A (A,B,C,A$,B$,C$)

PRIMEIRO REGISTO

A				PRIMEIRO PARÂMETRO			,
41	0E	00	00	aa	aa	00	2C

↑
(DEFADDI

SEGUNDO REGISTO

B				SEGUNDO PARÂMETRO			,
42	0E	00	00	bb	bb	00	2C

TERCEIRO REGISTO

C				TERCEIRO PARÂMETRO			,
43	0E	00	00	cc	cc	00	2C

QUARTO REGISTO

A	$			ENDR. DA *STRING*		CM. DA *STRING*		,
41	24	0E	??					2C

QUINTO REGISTO

B	$			ENDR. DA *STRING*		CM. DA *STRING*		,
42	24	0E	??					2C

SEXTO REGISTO

C	$			ENDR. DA *STRING*		CM. DA *STRING*		)
43	24	0E	??					29

Figura 9.2

Deixemos isto e passemos ao próximo capítulo.

CAPÍTULO DEZ

Um programa auxiliar

Agora que já sabemos mais ou menos o que é o código máquina, chegou a altura de aprendermos a manejá-lo um pouco melhor. Vamos escrever um novo programa, HEXLD 3, com o qual faremos dez coisas: 1.º, introduzir código máquina; 2.º, inserir código máquina entre rotinas já previamente existentes sem as estragar; 3.º, apagar o código máquina e tapar o buraco que ficou; 4.º, guardar em *cassette* e verificar código máquina; 5.º, listar código máquina; 6.º, substituir segmentos já existentes em código máquina por outros novos; 7.º, correr código máquina; 8.º copiar blocos de código máquina de um endereço para outro; 9.º, introduzir texto como dados; 10.º, converter decimal em hexadecimal e vice-versa. O mais importante deste programa é que as suas partes principais estão também escritas em código máquina, embora o restante seja em BASIC. Para fazê-lo funcionar, uma vez escrito, só é preciso introduzir um dos seguintes comandos:

RUN	P. listar o nosso código máquina.
RUN 100	Para escrever código máquina.
RUN 200	Para inserir código máquina.
RUN 300	Para apagar código máquina.
RUN 400	P. gravar e vfr. cód. máq.
RUN 500	Para cancelar todo o código máquina e começar de novo.
RUN 600	Para substituir código máquina por novo código.
RUN 700	Para correr código máquina.
RUN 800	Para copiar código máquina de um endereço para outro.
RUN 900	Para introduzir texto como dados.
PRINT FN H(A$)	Para converter hex em decimal.
PRINT FN H$(A)	Para converter decimal em hex.
PRINT FN P(A)	Para fazer um «PEEK duplo» em decimal.
PRINT FN P$(A$)	Para fazer um «PEEK duplo» em hex.

RANDOMIZE FN Q(X,Y)	Para fazer um «POKE duplo» em decimal.
RANDOMIZE FN R(A$,B$)	Para fazer um «POKE duplo» em hex.

Para carregar HEXLD 3 necessita-se de HEXLD 2. Os endereços empregados neste capítulo destinam-se às máquinas de 48 K. Nas outras, de cada vez que houver um endereço começado por F, deve usar-se 7. Por exemplo, usa-se 7FFE em vez de FFFE.

Vamos construí-lo aos poucos. Primeiro, para reservar algum espaço, escreve-se

```
16K: CLEAR 28671
48K: CLEAR 61439
```

Por agora, vamos escrever por cima de parte da zona de caracteres gráficos definíveis; não tem importância, porque não é permanente. Os endereços FFF6 a FFFF serão utilizados como «variáveis de sistema»; assim, escreveremos a primeira sub-rotina nos endereços imediatamente inferiores.

Esta sub-rotina tem o nome de C__PRINT *(Character Print)* e a sua função é escrever um qualquer carácter precedido de um espaço. A directiva ORG FFED quer dizer apenas «escreva esta sub-rotina no endereço FFED».

```
                    ORG FFED
F5      C__PRINT    PUSH AF
F5                  PUSH AF
3E20                LD A,«espaço»
D7                  RST 10
F1                  POP AF
D7                  RST 10
F1                  POP AF
C9                  RET
```

A sub-rotina requer que o carácter a escrever esteja no registo A. A sub-rotina de que precisaremos a seguir permite-nos escrever o valor do registo A em hexadecimal. Esta sub-rotina tem dois pontos de entrada *(entry-points):* H__PRINT *(Hex Print)*, que serve para escrever o valor hex do registo A, e SH__PRINT *(Space Hex Print)*, que serve para imprimir também o valor de A em hexadecimal, precedido de um espaço. Vamos escrever estas sub-rotinas:

```
                    ORG FFCB
F5      SH__PRINT   PUSH AF         Empilha o byte a escrever.
3E20                LD A,«espaço»
D7                  RST 10          Escreve um espaço.
F1                  POP AF          Recupera o byte a escrever.
F5      H__PRINT    PUSH AF         Guarda A p. uso posterior.
E6F0                AND F0          Isola o primeiro dígito.
1F                  RRA             Desloca o primeiro
IF                  RRA             dígito para a posição
IF                  RRA             conveniente no registo A.
1F                  RRA
C630                ADD A,"0"       Converte em carácter ASCII.
FE3A                CP 3A           Está este dígito entre A e F?
3802                JR C,HP__H
C607                ADD A,07        Se sim, determine o valor correcto.
D7      HP__H       RST 10          Escreve este dígito hex.
F1                  POP AF          Recupera o valor original de A.
E60F                AND 0F          Isola o segundo dígito.
C630                ADD A,"0"       Converte em carácter ASCII.
FE3A                CP 3A
3802                JR C,HP__L
C607                ADD A,07        Corrige se necessário.
D7      HP__L       RST 10          Escreve este dígito hex.
C9                  RET
```

Agora que já temos uma sub-rotina para escrever o registo A em hex, que tal uma sub-rotina que escreva todos os registos em hex? É o que faz a próxima sub-rotina. Tem o nome de REGS:

```
                ORG FF99
F5      REGS    PUSH AF         Empilha o registo dos
                                flags.
CDD0FF          CALL H__PRINT   Escreve o registo A em
                                hex.
78              LD A,B
CDCBFF          CALL SH__PRINT  Escreve o registo B em
                                hex.
78              LD A,C
CDCBFF          CALL SH__PRINT  Escreve o registo C em
                                hex.
7A              LD A,D
CDCBFF          CALL SH__PRINT  Escreve o registo D em
                                hex.
7B              LD A,E
CDCBFF          CALL SH__PRINT  Escreve o registo E em
                                hex.
7C              LD A,H
CDCBFF          CALL SH__PRINT  Escreve o registo H em
                                hex.
7D              LD A,L
CDCBFF          CALL SH__PRINT  Escreve o registo L em
                                hex.
F1              POP AF          Recupera o registo flags
                                original.
3E53            LD A,"S"
FCEDFF          CALL M,C__PRINT Escreve «S» se o flag S
                                está set.
3E5A            LD A,"Z"
CCEDFF          CALL Z,C__PRINT Escreve «Z» se o flag Z
                                está set.
3E50            LD A,"P"
ECEDFF          CALL PE,C__PRINT Escreve «P» se o flag
                                P/V está set.
3E43            LD A,"C"
DC
EDFF            CALL C,C__PRINT Escreve «C» se o flag C
                                está set.
C9              RET
```

A sub-rotina seguinte é muito astuta — a sua função é verificar se as interrupções *(interrups)* estão ou não activas. Isto significa uma coisa muito simples: o *Spectrum* tem duas espécies de operações

diferentes, chamadas «interrupções activas» e «interrupções não activas». Podemos mudar de um modo para o outro por meio das instruções EI *(Enable Interrupts)* e DI *(Disable Interrupts)*. Com as interrupções inactivas, o *Spectrum* trabalhará um pouco mais depressa, mas tem a desvantagem de o teclado não ser lido automaticamente cinquenta vezes por segundo (falaremos sobre o teclado no próximo capítulo). As interrupções têm de ser sempre activas antes de retornar ao BASIC. Esta rotina de verificação confia apenas no facto de que, se as interrupções estão activas, a variável de sistema FRAMES será incrementada cinquenta vezes por segundo, mas com as interrupções inactivas não será este o caso. Note-se que seria ainda muito mais simples utilizar, em vez da rotina referida, a simples instrução LD A, I — ou LD A,R — e testar o *flag* P/V:

```
                ORG FF89
2A785C  ICHECK  LD HL,(FRAMES)  HL: = dois bytes mais
                                baixos de (FRAMES).
01AA0A          LD BC,0AAA      Isto representa 1/50 de
                                segundo de atraso.
0B      I__LOOP DEC BC
78              LD A,B
B1              OR C
20FB            JR NZ,I__LOOP   Espera 1/50 de
                                segundo.
3A785C          LD A,(FRAMES)   A: = byte mais baixo de
                                (FRAMES).
BD              CP L            Compara com o byte
                                mais baixo anterior.
C9              RET             A sub-rotina retornará
                                com o flag zero
                                indicando o estado
                                corrente das
                                interrupções.
```

Construímos pouco a pouco a nossa selecção de sub-rotinas; a seguinte tem três endereços de entrada possíveis e a função de escrever um valor hexadecimal de quatro dígitos:

```
                ORG FF77
010000  U__ADDR LD BC,0000
2AF8FF  PR__ADDR LD HL,(ADDRESS)
7C      PR__HL  LD A,H
```

```
CDD0FF          CALL H__PRINT
7D              LD A,L
CDD0FF          CALL H__PRINT
3E20            LD A, «espaço»
D7              RST 10
C9              RET
```

Note-se que esta rotina pode ser chamada do BASIC usando PRINT USR 65399 (ou 32631 se a máquina for de 16 K — tendo, neste caso, de subtrair 32768 a todos os endereços dados). O zero a mais que aparece escrito é o valor final de BC (uma vez que utilizamos USR). Pode-se no entanto eliminá-lo escrevendo PRINT USR 65399;CHR$8;" "(CHR$ 8 é o espaço atrás).

Um outro artifício útil, que utilizaremos em breve, é a instrução RST Ø8 (que significa CALL ØØØ8) seguida do *byte* FF. Isto provoca um retorno imediato ao modo de comando do BASIC sem ter em conta a pilha e o valor de HL'. A rotina seguinte de código máquina utilizará este artifício. Infelizmente, há um pequeno problema que aparece sempre que se usa RST Ø8 — é que o registo IY tem de conter o valor 5C3A antes de o podermos usar, senão entraremos em quebra (mas será espantoso o uso de LD IY,1234/RST Ø8/DEFB FF).

Há também umas precauções a tomar no uso de RST 1Ø. Tal como RST Ø8, o registo IY tem de conter o valor 5C3A para que o sistema não entre em quebra. Mais ainda, RST 1Ø corrompe os valores dos registos A, A', B', C', D' e E' e ainda os registos dos *flags* F e F' e, portanto, temos de tomar cuidado. Tudo o que temos dito tem aplicação na próxima rotina...

Chamamos-lhe rotina de *break point* (ponto de paragem) e empregamo-la como ajuda à correcção dos erros no programa. Destina-se a escrever todos os registos e *flags*, incluindo os alternativos e também os registos IX e IY e o endereço corrente. Feito isto, provocará um retorno ao modo de comando BASIC. Para utilizar esta rotina basta escrever CALL BREAKPT em qualquer programa de código máquina e em qualquer altura.

```
                ORG FF2A
FDE5    BREAKPT PUSH IY
FD213A5C        LD IY,5C3A      Isto para que RST 10
                                possa funcionar.
```



```
DDE5            PUSH IX
Ø8              EX AF,AF'
F5              PUSH AF         Porque RST 1Ø
                                corrompe A' e F'.
Ø8              EX AF,AF'
D9              EXX
C5              PUSH BC         Preserva BC'.
D5              PUSH DE         Preserva DE'.
E5              PUSH HL         Preserva HL'.
D9              EXX
F5              PUSH AF         Porque RST 1Ø
                                corrompe F.
3E78            LD A,78
328F5C          LD (ATTR__T),A  Assegura impressão em
                                preto sobre branco
                                brilhante.
AF              XOR A
323C5C          LD (TVFLAG),A   Assegura que a
                                impressão esteja na
                                parte alta do écran.
3EØD            LD A,"enter"
D7              RST 1Ø          Carriage return.
F1              POP AF          Restaura os flags.
CD99FF          CALL REGS       Escreve os registos.
3EØD            LD A,"enter"    registos.
D7              RST 1Ø          Carriage return.
E1              POP HL
D1              POP DE
C1              POP BC
F1              POP AF
CD99FF          CALL REGS       Escreve todos os
                                registos alternativos.
3EØD            LD A,"enter"
D7              RST 1Ø          Escreve um carriage
                                return.
E1              POP HL          HL: = IX original.
CD7DFF          CALL PR__HL     Escreve o registo IX.
E1              POP HL          HL: = IY original.
```

```
CD 7DFF          CALL PR__HL     Escreve o registo IY.
E1               POP HL          HL: = endereço de
                                 retorno da sub-rotina.
2B               DEC HL
2B               DEC HL
2B               DEC HL
CD 7DFF          CALL PR__HL     Escreve o endereço do
                                 qual a sub-rotina foi
                                 chamada.
CD89FF           CALL I__CHECK   Flag zero set se as ints.
                                 estão inactivas.
3E44             LD A,"D"        «D» significa Disabled.
2801             JR Z,BC__I
3C               INC A           Se activas, muda A para
                                 «E» de Enabled
D7       BC__I   RST 10          Escreve «D» ou «E»
                                 conforme necessário.
DD213A5CEXIT     LD IY,5C3A      Assegura que RST 08
                                 funciona, se a rotina for
                                 chamada à etiqueta
                                 EXIT.
FB               EI              Interrupções activas para
                                 voltar ao BASIC.
CF               RST 08
FF               DEFB FF         Retorna ao modo de
                                 comando BASIC.
```

Pode-se experimentar agora esta rotina, escrevendo RUN 700 e introduzindo FF2A como endereço de partida. FF2A é o endereço da instrução inicial da rotina BREAKPT. Se a rotina funcionar, aparecerá o seguinte escrito no *écran* (o valor DE pode variar):

```
2A FF 2A ?? ?? 2D 2B Z P
00 17 21 36 9B 27 58 Z P
03D4 5C3A 2D28 E
```

As primeiras duas filas de números representam os valores correntes de todos os registos na ordem A, B, C, D, E, H e L (a seguir os *flags*) e A', B', C', D', E', H' e L' (a seguir os *flags* alternativos). Os símbolos Z e P indicam que só Z e P estão correntemente a um. A fila de baixo contém os valores de IX,IY e o endereço (neste caso na ROM) de qual a sub-rotina que foi

chamada. Note-se que o valor inicial DE pode variar e que o valor inicial de BC é o endereço de chamada de USR, de modo que, empregando 16 K em vez de 48, este valor será na verdade 7F2A e não FF2A.

Vamos tratar agora daquela que é realmente a primeira parte de HEXLD 3 — a rotina de listagem. Esta é uma sub-rotina para listar código máquima em hex utilizando três das nossas variáveis de sistema, que são:

BEGIN (guardada em FFF6) — o primeiro endereço que podemos listar;
ADDRESS (guardada em FFF8) — o endereço a listar em seguida;
LIMIT (guardada em FFFE) — o último endereço a listar, mais um.

Esta é a primeira parte de HEXLD 3 que podemos utilizar visivelmente desde já. Vamos fazê-lo em três partes: primeiro o código máquina; segundo, o BASIC; e terceiro, as variáveis acima indicadas. Vamos escrever o seguinte código máquina:

```
                 ORG FEF0
AF       H__LIST XOR A
323C5C           LD (TVFLAG),A     Selecciona a parte
                                   superior do écran.
2AF8FF           LD HL,(ADDRESS)   HL: = endereço a partir
                                   do qual listamos.
ED5BF6FF         LD DE,(BEGIN)     DE: = primeiro endereço
                                   permitido.
A7               AND A
ED52             SBC HL,DE
19               ADD HL,DE
3003             JR NC,HL__A
2AF6FF           LD HL,(BEGIN)     HL: = primeiro endereço
                                   a partir do qual podemos
                                   listar.
ED5BFEFFHL__A    LD DE,(LIMIT)     DE: = primeiro endereço
                                   ilegal.
22F8FF   HL__LOOP LD (ADDRESS),HL  Guarda endr. corrente.
A7               AND A
ED52             SBC HL,DE
19               ADD HL,DE         Teste para ver se o
                                   endereço é permitido.
```

```
305F              JR NC,EXIT        Sai se tiver acabado.
3E0D              LD A,"enter"
D7                RST 10            Escreve carriage return.
CD7DFF            CALL PR__HL       Escreve o endereço
                                    corrente.
7E                LD A,(HL)         A: = byte guardado
                                    neste endereço.
CDD0FF            CALL H__PRINT     Escreve este byte.
7E                LD A,(HL)         Restaura o byte em A.
FE20              CP 20             Se não for um carácter
                                    de controle...
3007              JR NC,HL__NEXT
FEA5              CP A5             E se o carácter não for
                                    uma key word...
3003              JR NC,HL__NEXT
CDEDFF            CALL C__PRINT     Então escreve este
                                    carácter.
23      HL__NEXT  INC HL            Aponta o pr. endereço.
18DE              JR HL__LOOP
```

E o BASIC:

```
1000 PRINT " LIST ";
1010 GO SUB 8000
1020 RANDOMISE USR 65264      Isto é FEF0 em decimal (H LIST).
8000 LET X=65528              Isto é FFF8 em decimal (ADDRESS).
8030 INPUT "ENDEREÇO"; A$     A numeração das linhas
8040 PRINT "ENDEREÇO"; A$     é deliberada.
8050 LET Y=0
8051 FOR I=1 TO 4             Ver a numeração das linhas.
8052 LET Z=CODE A$(I)-48
8053 IF Z>9 THEN LET Z=Z-7
8054 LET Y=16*Y+Z
8055 NEXT I
8060 POKE X,Y-256*INT (Y/256)
8070 POKE X+1,INT (Y/256)
8080 RETURN
```


Precisamos ainda atribuir valores às variáveis BEGIN e LIMIT. Para fazer isto escreve-se RUN e introduz-se (após indicação "WRITE TO") "FFF6", introduzindo seguidamente "F0FE" (isto é FEF0 — o endereço de H__LIST — com os *bytes* trocados) "0000", "0000", "0000", "F6FF" (que é o endereço FFF6 com os *bytes* trocados). Faz-se BREAK e está pronto.

Para ver H__LIST a funcionar, escreve-se RUN 1000. Pode-se introduzir qualquer endereço, mas ver-se-á que não se pode listar antes de FEF0 ou depois de FFF5 — podendo agora utilizar-se H__LIST para listar todo o programa.

Vamos agora ver mais algumas sub-rotinas. A primeira é SET__UP, que faz duas coisas. Primeiro, atribui a HL o menor dos valores de (ADDRESS) e (BEGIN) e redefine (BEGIN) com este mínimo, se necessário. Tudo isto é muito claro. A segunda função requer, no entanto, um conhecimento mais aprofundado sobre o funcionamento interno da ROM. A sua função é carregar BC com o número de *bytes* na *string* A$ e apontar HL para o *byte* imediatamente anterior no texto desta *string*. Para guardar variáveis BASIC a ROM reserva um certo espaço na RAM, que é apontado pela variável de sistema VARS. Se A$ é a única variável definida, VARS indicará um *byte* que será, de facto, o *byte* "A" (hex 41). (Seria "B" para B$ ou "T" para T$, e assim por diante). Logo a seguir, vem o comprimento da *string* e, depois, o próprio texto da *string*. Tendo isto em contra, eis a sub-rotina — note-se que também pode ser chamada a etiqueta LEN__A$.

```
                  ORG FED1
AF      SET__UP   XOR A
323C5C            LD(TVFLAG),A      Escreve na parte
                                    superior do écran.
3E0D              LD A,"enter"
D7                RST 10            Carriage return
2AF8FF            LD HL,(ADDRESS)
ED5BF6FF          LD DE,(BEGIN)
ED52              SBC HL,DE         (Note-se que RST 10 faz
                                    o flag carry reset.)
19                ADD HL,DE
3003              JR NC,SU__BEGIN
```

```
22F6FF                  LD(BEGIN),HL    (BEGIN): = mínimo
                                        de (BEGIN),
                                        (ADDRESS).
EB        SU__BEGIN     EX DE,HL        DE: = (ADDRESS).
2A4B5C    LEN__A$       LD HL,(VARS)    HL aponta para a
                                        variável BASIC A$.
23                      INC HL
4E                      LD C,(HL)
23                      INC HL
46                      LD B,(HL)       BC: comprimento da
                                        string A$.
C9                      RET
```

A próxima sub-rotina define (LIMIT) como o maior de (ADDRESS) ou (LIMIT). Por outras palavras, se ADDRESS indicar um endereço mais alto na memória de que aquele indicado por LIMIT, este será redefinido:

```
                        ORG FEC1
2AF8FF    NEW__LIM      LD HL,(ADDRESS)
ED5BFEFF  NL__2         LD DE,(LIMIT)
A7                      AND A
ED52                    SBC HL, DE
19                      ADD HL,DE
D8                      RET C
22FEFF                  LD (LIMIT),HL
C9                      RET
```

Vamos ver agora outra parte de HEXLD 3, a parte WRITE. Para evitar andarmos depressa de mais, vamos abrandar agora um pouco. Vamos apresentar a listagem do código máquina de uma só vez, mas no fim explicaremos tudo, de forma que cada qual possa voltar atrás e descobrir por si próprio o que faz cada parte.

```
                        ORG FE87
CDD1 FE   WRITE         CALL SET__UP
EB                      EX DE,HL        DE: = aponta para o
                                        primeiro byte antes do
                                        texto da string A$, HL: =
                                        endr. onde escrever.
```



```
CB38                    SRL B           Divide BC por dois para
                                        obter o número de bytes
                                        a escrever.
CB19                    RR C
CA70FF                  JP Z,EXIT       Sai se string está vazia.
CD7DFF                  CALL PR__HL     Imprime o endereço de
                                        escrita.
13        W__LOOP       INC DE          Aponta DE ap. para o
                                        primeiro dígito hex do pr.
                                        byte.
1A                      LD A,(DE)       A: = este carácter («Ø»
                                        a «9» ou «A» a «F»).
D7                      RST 10          Escreve o prim. díg. hex.
1A                      LD A,(DE)       Restaura A (RST 10
                                        destrói o seu conteúdo).
FE40                    CP 40
3804                    JR C,WR1        Salta entre «Ø» e «9».
E6DF                    AND DF          Passa p. maiúsculas.
D607                    SUB 07          Corrige o valor
                                        de A.
87        WR1           ADD A,A
87                      ADD A,A         Desloca A um dígito hex
                                        para a esquerda (de
                                        modo que 30 («Ø») seja
                                        ØØ,35 («5») seja 50 e
                                        assim por diante.
87                      ADD A,A
87                      ADD A,A
E5                      PUSH HL         Guarda HL.
67                      LD H,A          H: = primeiro dígito hex
                                        vezes dezasseis.
13                      INC DE          DE aponta para o
                                        segundo dígito hex.
1A                      LD A,(DE)       A: = segundo díg. hex.
D7                      RST 10          Escreve o 2.° díg. hex
1A                      LD A,(DE)
FE40                    CP 40
3804                    JR C,WR2        Salta se o dígito está
                                        entre «Ø» e «9».
E6DF                    AND DF          Converte em
                                        maiúsculas.
```

```
D607            SUB 07          Corrige o
                                código.
E60F    WR2     AND 0F          Considera apenas o
                                segundo dígito hex.
B4              OR H            Combina-o com o
                                primeiro dígito hex.
E1              POP HL          Restaura HL.
77              LD(HL),A        Guarda o byte no
                                endereço requerido.
23              INC HL
22F8FF          LD(ADDRESS),HL  Guarda o endereço do
                                byte seguinte.
0B              DEC BC
78              LD A,B
B1              OR C
20D4            JR NZ,W_LOOP
```

(O código máquina imediatamente a seguir a este é a sub-rotina NEW_LIM.)

Vamos explicar aqui alguma terminologia. HEXLD 3 é um programa em código máquina com a função de criar e correr outros programas em código máquina, o que pode criar muita confusão no simples emprego da palavra «programa»; para contornarmos este programa, referimo-nos a HEXLD 3 como «programa objecto», e o programa que está a ser editado como «programa sujeito». Em alguns casos limitados, HEXLD 3 examina-se a si próprio, sendo nesse caso programa objecto e sujeito ao mesmo tempo; no entanto é bom mantermos esta distinção e, num sentido muito abstracto, devemos considerar HEXLD 3 como operando sobre um conjunto separado de informação.

Se a *string* estiver vazia, saímos — não retornamos somente; retornamos ao modo de comando do BASIC. De outro modo, o endereço no qual desejamos escrever é «impresso» e entramos no ciclo principal.

Cada dois *bytes* da *string* correspondem a um *byte* hex. Por exemplo, a *string* conter "2A"implica o *byte* 2A — no entanto a própria *string* contém os códigos dos carácteres "2", que é 32 seguido de 41 que é o código de "A". Mais ainda, é também possível que a *string* contenha "2a" (isto é, com o a minúsculo); nesse caso, os códigos dos caracteres seriam 32 seguido de 61 — ambos estes

casos dariam como resultado o *byte* 2A. Para que isto aconteça, é necessária uma conversão pela qual os caracteres com o código superior a 40 (isto é "A" a "F" ou "a" a "f") sejam convertidos em maiúsculas (a instrução AND DF tem esse efeito); compreende-se porquê? A instrução OR 20 converte todas as letras em minúsculas, após o que se subtrai sete ao resultado. Isto realmente fecha o intervalo entre "9" e "A", de modo que "9" produz ainda o resultado 39 mas "A" produz agora o código 3A (tal como "a"). Será agora bastante fácil acompanhar o programa até à instrução LD (HL), A, logo a seguir à etiqueta WR2, onde o *byte* resultante é finalmente calculado e depositado no endereço requerido.

Depois de o endereço seguinte ter sido guardado, faz-se nova verificação. A variável LIMIT guarda o endereço do primeiro *byte* a seguir ao fim do programa sujeito, pelo que se escrevermos para um endereço maior ou igual a (LIMIT), este limite terá de ser alterado .

Mais um ponto a ter em conta — a instrução DEC BC não altera o *flag* zero, de modo que, para verificar se acabamos ou não, temos de empregar LD A,B/OR C. Passemos agora ao BASIC.

Faz-se RUN 1000 para verificar que a listagem da rotina WRITE está correcta. Escreve-se RUN para ler o valor a atribuir à variável BEGIN — para o fazer, deve-se introduzir "FFF6" e a seguir "00F0".

Faz-se BREAK agora, e quando estiver tudo em ordem, altera-se o BASIC de modo que se segue:

```
10 PRINT "List ";
20 GO SUB 8000
30 RANDOMIZE USR 65264
100 PRINT "Write ";
110 GO SUB 8000
120 CLEAR
130 INPUT LINE A$
140 RANDOMIZE USR 65159
150 GO TO 130
```

Devem-se também apagar todas as outras linhas entre 1 e 199 e também entre 1000 e 7999. Algumas das linhas na sub-rotina a partir de 8000 devem ser mudadas por razões estéticas.

```
8030 INPUT "ENDEREÇO"; LINE A$
8040 PRINT              ;A$
```

Agora faz-se SAVE deste programa antes de tentar corrê-lo. Para fazer isto, apagam-se as linhas 400 e 410 e muda-se 440 para SAVE "M/CODE" CODE 61440,4196 e depois RUN 400. De agora em diante, RUN listará código máquina e RUN 100 escrevê-lo-á. Já agora, repare-se que a instrução CLEAR no BASIC foi aí colocada para assegurar que A$ é a primeira variável existente na área das variáveis. Sem este CLEAR o código máquina não pode funcionar.

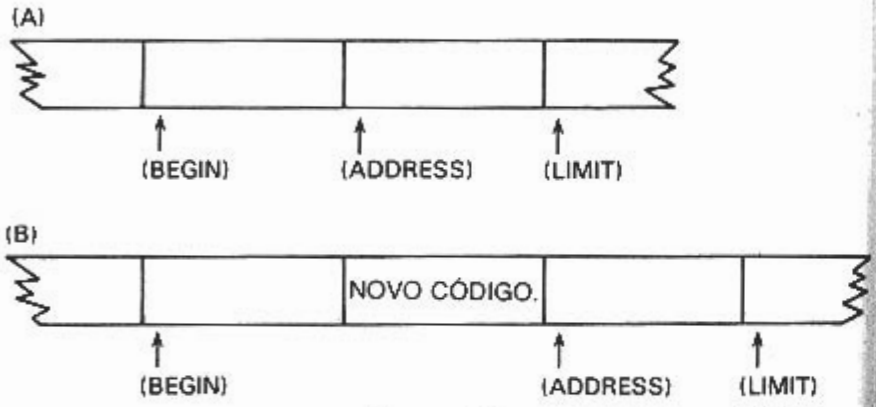


Figura 10.1

O código máquina seguinte executa uma tarefa bastante inteligente. Veja-se a figura 10.1: mostra o programa sujeito antes e depois de chamada a rotina. A rotina chama-se INSERT e esperamos que se perceba porquê. A variável ADDRESS funciona como uma espécie de cursor, de modo que aquilo que escrevemos é inserido na posição do curso e o próprio cursor se desloca para o fim dos *bytes* recém-inseridos. Nenhum código máquina é destruído por este processo, já que todo o código à direita do cursor (isto é, com o endereço superior) se move para cima na memória. Segue-se o código máquina para realizar tal tarefa. Para iniciar a inserção emprega-se RUN 100. Note-se que premindo "enter", só por si, se provoca BREAK.

		ORG FE63	
CDE8FE	INSERT	CALL LEN__A$	
CB38		SRL B	
CB19		RR C	BC: = número de *bytes* a inserir.
CA70FF		JP Z,EXIT	Termina se a *string* a introduzir é vazia.
C5		PUSH BC	Empilha o número de *bytes* a inserir.
2AFEFF		LD HL,(LIMIT)	HL: = aponta p. o primeiro *byte* p. além do programa sujeito.
ED5BF8FF		LD DE,(ADDRESS)	DE: = endereço no qual vamos inserir.
A7		AND A	
ED52		SBC HL,DE	HL: = número de *bytes* do programa sujeito que precisam ser movidos.
23		INC HL	Mais um, para evitar *crash* se o número de *bytes* a mover for zero.
44		LD B,H	
4D		LD C,L	BC: = número de *bytes* a mover.
E1		POP HL	HL: = número de *bytes* a inserir.
ED5BFEFF		LD DE,(LIMIT)	DE: = valor anterior de LIMIT.
19		ADD HL,DE	HL: = novo valor de LIMIT.
22FEFF		LD (LIMIT),HL	Guarda este valor.
EB		EX DE,HL	DE: = novo LIMIT.
			HL: = velho LIMIT.
EDB8		LDDR	Muda todos os *bytes* necessários.

Observe-se que não existe instrução de retorno no fim desta rotina e também que a rotina acima apresentada não altera o valor de ADDRESS, conforme o prometido. Isto resulta de o endereço a seguir à rotina ser o da instrução em código máquina, início da rotina WRITE, que substituirá os *bytes* agora sem utilidade no

espaço recém-inserido, redefinindo ADDRESS e retornando. Precisamos de mais algum BASIC agora e para isso escreveremos:

```
500 PRINT "BEGIN ADDRESS";
510 INPUT A$: PRINT A$
520 LET X=0
521 FOR I=1 TO 4
522 LET Y=CODE A$-48: IF Y>9 THEN LET Y=Y-7
523 LET X=16*X+Y
524 NEXT I
530 LET Z=INT (X/256)
540 LET Y=X-256*Z
550 POKE 65526,Y          Este é o endereço de BEGIN em decimal.
560 POKE 65527,Z
570 POKE 65534,Y          Este é o endereço de LIMIT em decimal.
580 POKE 65535,Z
590 STOP
```

Agora, que esperamos ter a possibilidade de inserir código, é muito importante assegurarmo-nos que HEXLD 3 nunca esteja a apontar para si próprio quando testarmos essa inserção (de outro modo será uma grande complicação) e por isso a rotina que escrevemos na linha 5ØØ tem a função de fazer com que os apontadores BEGIN e LIMIT se mantenham afastados de HEXLD 3 apontando para qualquer lugar seguro. Voltemos à rotina INSERT.

Apagam-se todas as linhas de BASIC entre 2ØØ e 299 e introduza-se o seguinte:

```
200 PRINT "Insert ";
210 GO SUB 8000
220 CLEAR
230 INPUT LINE A$
240 RANDOMIZE USR 65123
250 GO TO 230
```

O nosso único problema agora é que a rotina SAVE não funcionará devidamente (porque apagámos a sub-rotina anterior em 2ØØ) e, assim, teremos de fazer mais alterações. Para isso, apagam-se todas as linhas entre 4ØØ e 49Ø e recomeça-se:


```
400 SAVE "Hexld 3" LINE 450
410 SAVE "Hexld 3 mc" CODE 63488,2048 (Isto irá guardar todo o
F800 código de F800 a FFFF.)
420 SAVE " " CODE FN P(65526),FN P(65534) - FN P(65526) + 1
430 VERIFY "": VERIFY "" CODE: VERIFY "" CODE
440 STOP
450 CLEAR 61439          (Isto liberta a memória de F000 para cima.)
460 PAPER 7: INK Ø
470 LOAD "" CODE: LOAD "" CODE
480 STOP
```

E para completar...

```
9000 DEF FN P(X)=PEEK X+256*PEEK (X+1)
```

Agora escrevemos RUN 4ØØ para fazer SAVE do programa. Nesta fase estamos prontos para uma pequena demonstração. Escreve-se RUN 5ØØ para afastar os apontadores para longe de HEXLD 3 e introduz-se "FØØØ". Vê-se a mensagem «9 STOP statement 59Ø:1» aparecer na parte de baixo do *écran*. Escreve-se agora RUN 1ØØ para escrever novo código e introduz-se novamente o endereço FØØØ. Veremos o *écran* ficar completamente em branco, por causa da instrução CLEAR na linha 120. Agora introduzimos a sequência ØØ/Ø1/Ø2/Ø3/Ø4/Ø5/Ø6/Ø7/ /Ø8/Ø9/ØA/ØB/ØC/ØD/ØE/ØF (onde «/» significa ENTER). Prima-se ENTER só por si para terminar. Agora, se escrevemos RUN e introduzimos FØØØ vemos aparecer a listagem deste código máquina. Agora uma parte interessante: escrevemos RUN+ØØ e desta vez introduzimos o endereço FØØ4. A seguir escrevemos os seguintes *bytes:* 2Ø/3432/2Ø (mais uma vez «/» significa ENTER). Saímos carregando ENTER só por si. Para vermos o que aconteceu, escrevemos RUN e introduzimos FØØØ:

Os quatro *bytes* introduzidos na rotina INSERT foram inseridos sem destruir o código anterior.

E agora uma coisa totalmente diferente...

Observe-se a figura 10.2, que mostra as condições requeridas para o processo DELETE. Como se vê, aparece aqui uma nova variável —ADD2, que é, neste caso, o último endereço a apagar. Todo o código máquina de (ADDRESS) a (ADD2) deve ser

apagado e o intervalo tapado com o restante código de endereço superior a ADD2, deslocado para baixo na memória:

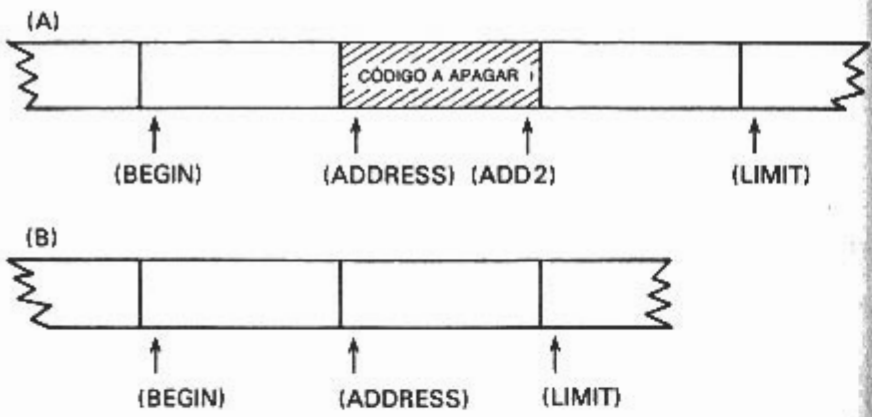


Figura 10.2

O código máquina é:

```
                    ORG FE49
2AFEFF   DELETE     LD HL,(LIMIT)      HL: = primeiro byte p.
                                       além do programa
ED5BF                                  sujeito.
AFF                 LD DE,(ADD2)       DE: = último byte a
                                       apagar.
13                  INC DE             DE: = primeiro byte a
                                       mover.
A7                  AND A
ED52                SBC HL,DE          HL: = número de bytes
                                       a mover.
44                  LD B,H
4D                  LD C,L             BC: = número de bytes
                                       a mover.
Ø3                  INC BC             Mais um para evitar
                                       crash se for feita uma
                                       tentativa de apagar o
                                       último byte.
2AF8FF              LD HL,(ADDRESS)    HL: = endereço para
                                       onde deslocar dados.
EB                  EX DE,HL           DE: = endr. p. onde
                                       deslocar dados, HL: =
                                       endr. do 1.º byte a mover.
EDBØ                LDIR               Move todos os bytes
                                       necessários.
1B                  DEC DE             DE: = aponta o primeiro
                                       byte para além do
                                       programa sujeito actual.
ED53FEFF            LD (LIMIT),DE      Guarda este novo limite.
C9                  RET
```

E o BASIC é:

```
300 PRINT " Delete ";          Sim, há um espaço antes do D.
310 GO SUB 330
320 STOP
330 GO SUB 8000
340 PRINT «cinco espaços To»;
350 GO SUB 8020
360 RANDOMIZE USR 65097
370 RETURN
8010 GO TO 8030
8020 LET X=65530
```

Com RUN 1ØØ introduziremos o novo código máquina, mas é necessário escrever RUN 5ØØ e introduzir "FØØØ" de modo a, depois, mudar os apontadores. Uma pequena demonstração apenas para se observar que funciona.

Escreve-se RUN 5ØØ, introduz-se "FØØØ". Escreve-se RUN 1ØØ e a seguir FØØØ/ØØ/Ø1/Ø2/Ø3/Ø4/Ø5/Ø6/Ø7/ /Ø8/Ø9/ØA/ØB/ØC/ØD/ØE/ØF/ (onde «/» representa ENTER). Sai-se carregando ENTER só por si. RUN 3ØØ — a parte crucial — e introduza-se então FØØ5 seguido de FØØ8. RUN seguido de FØØØ será suficiente para provar que temos razão.

Como se pode observar, LDDR foi empregado na rotina de inserção e LDIR em DELETE. Ao mover blocos de memória para cima ou para baixo é importante começar na extremidade certa. Por exemplo, supondo que queríamos mover um bloco de nove *bytes* de FØØ1 a FØØØ, se fizermos HL=FØØ1 e DE=FØØØ, podemos empregar LDIR sem problemas. No entanto, partindo de outro lado,

com HL contendo inicialmente F009 e DE F008 e a seguir transferindo de F008 (o qual foi agora mesmo substituído) para F007, a seguir de F007 (o qual mais uma vez acaba de ser transferido) para F006 e assim por diante, o resultado seria «ligeiramente» incorrecto! Vamos agora aumentar HEXLD 3 sem acrescentar mais nenhum código máquina. Esta é a rotina REPLACE:

```
600 PRINT "Replace  ";
610 GO SUB 330
620 GO TO 220
```

Astucioso, não? REPLACE funciona chamando DELETE seguida de INSERT. Ao correr o programa são solicitados dois endereços; o primeiro e o último endereços do bloco de código que queremos substituir. O novo código então introduzido será inserido, automaticamente, neste ponto. Vamos experimentar procurando cada qual inventar a sua própria demonstração, o que não é difícil.

Agora vamos deitar fora todo o «lixo» da linha 500, escrevendo o seguinte código máquina:

```
                    ORG FE3E
2AF8FF   BEGINMC    LD HL,(ADDRESS)
22F6FF              LD (BEGIN),HL
2B                  DEC HL
22FEFF              LD (LIMIT),HL
C9                  RET
```

Apagamos agora todas as linhas de BASIC entre 500 e 599 e substituímo-las pelo seguinte:

```
500 PRINT "Begin ";
510 GO SUB 8000
520 RANDOMISE USR 65086
530 GO TO 120
```

A rotina em 500 tem agora o seguinte efeito: cancelar qualquer programa sujeito anterior e começar de novo. Note-se que a rotina fica automaticamente à espera que se introduza novo código.

Como se vê, já conseguimos completar a maior parte do que em princípio queríamos de HEXLD 3. Vamos agora ver a rotina na linha 700.

Primeiro o código máquina. É bastante complicado e difícil de compreender todo de uma só vez.

```
                    ORG FE10
E5       H__RUN     PUSH HL          Empilha o valor inicial de
                                     HL para o caso de o
                                     programa depender dele.

211CFE              LD HL,R__CH
E3                  EX (SP),HL       Restaura HL e empilha o
                                     endereço de retorno
                                     «artificial» R__CH.
ED4BF8FF            LD BC,(ADDRESS)  BC: = endereço de
                                     chamada de USR.
79                  LD A,C           A: = endr. de chamada
                                     de USR (parte baixa)
                                     caso o programa sujeito
                                     dependa dele.
C5                  PUSH BC          Empilha o endereço de
                                     chamada de USR.
C9                  RET              Chama a sub-rotina
                                     USR.
F5       R__CH      PUSH AF          Empilha o registo dos
                                     flags.
D9                  EXX              Selecciona o conjunto de
                                     registos alternativos.
D5                  PUSH DE          Empilha o valor de DE'.

115827              LD DE,2758

A7                  AND A
ED52                SBC HL,E         Flag zero set se HL' for
                                     igual a 2758.
19                  ADD HL,DE        Restaura HL'.
D1                  POP DE           Restaura DE'.
D9                  EXX              Rest. tds. os registos.
200B                JR NZ,R__EXIT    Salta se HL' <> 2758.
C5                  PUSH BC
E5                  PUSH HL
```

```
CD89FF          CALL I__CHECK   Vf. se as interrupções
                                estão ou não activas.
E1              POP HL
C1              POP BC
2802            JR Z,R__EXIT
F1              POP AF          Elimina o últ. item na
                                pilha.
C9              RET             Retorna a BASIC.
F1      R__EXIT POP AF          Restaura A e
                                os flags.
E3              EX (SP),HL      Faz o valor no topo da
                                pilha ser 0003 de modo
                                que o terceiro item na
                                terceira linha de saída
                                de BREAKPT seja 0000.
210300          LD HL,0003
E3              EX (SP),HL
C32AFF          JP BREAKPT
```

E o BASIC que àcompanha esta rotina é:

```
700 PRINT "Run ";
710 GO SUB 8000
720 RANDOMIZE USR 65040
730 STOP
```

Devemos apagar todas as outras linhas de BASIC entre 700 e 799. Vamos agora tentar explicar o esquema deste complexo código máquina.

A primeira parte do programa (de H__RUN a R__CH menos um) assegura-nos que o endereço de retorno do programa sujeito é a etiqueta R__CH (RUN CHECK), mas de maneira que os registos contenham os mesmos valores iniciais que teriam se o programa sujeito fosse directamente chamado utilizando USR. Repare-se no modo pelo qual a sequência PUSH BC/RET é empregado para simular "JP (BC)". Ao retornar do programa sujeito, o controle estará no endereço R__CH, onde se fazem duas verificações separadas para ver, primeiro, se HL' contém o valor 2758 e, segundo, se as interrupções estão activas. Se qualquer destas verificações falha, a rotina BREAKOUT toma o controle, escrevendo o conteúdo de todos os registos e *flags*, incluindo os alternativos, os registos IX e IY e o número de referência 0000, que indica que o teste do programa sujeito falhou. Efectua-se então um retorno ao modo de comando do BASIC.

A razão destas verificações é simples: se uma dessas duas condições falhar e houver uma tentativa de retorno ao BASIC teremos *crash!* Para verificar este facto, pode-se fazer correr os programas EXX/RET e DI/RET.

Agora, um caso interessante: vamos modificar (usando EDIT) duas das linhas do BASIC do modo que se segue:

```
130 INPUT USR 65399;CHR$ 8;LINE A$
230 INPUT USR 65399;CHRS 8;LINE A$
```

Quando correr, veremos um efeito interessante. Vamos escrever RUN 500, e a seguir F000 e introduzir algum código. Repare-se na informação extra ao fundo do *écran*. Aparece porque USR 65399 é o endereço da rotina U__ADDR que carrega BC com zero e em seguida escreve (ADDRESS) em hex seguido de um espaço. Porque não fizemos XOR AF/LD (TVFLAG),A, este texto é escrito na zona inferior do *écran* e não na zona superior, como de costume. Como efeito secundário de ser uma instrução INPUT, o valor final de BC é também escrito (zero) e CHR$ 8 (espaço atrás) está lá para o apagar. A rotina COPY vem a seguir. (Não é caso para admiração: estamos quase a acabar.) Ei-la:

```
                        ORG FDD5
ED5BF8FF H__COPY LD DE,(ADDRESS) DE: = primeiro byte a
                                 mover.
2AF AFF          LD HL,(ADD2)    HL: = último byte a
                                 mover.
23               INC HL          HL: = primeiro byte para
                                 além deste.
A7               AND A
ED52             SBC HL,DE       HL: = número de bytes
                                 a mover.
44               LD B,H
4D               LD C,L          BC: = número de bytes
                                 a mover.
```

```
2AFCFF          LD HL,(ADD3)      HL: = endereço para
                                  onde deslocar os
                                  dados.
A7              AND A
ED52            SBC HL,DE
19              ADD HL,DE
3006            JR NC,HC__1       Salta se a deslocação
                                  for para endereços
                                  superiores na memória.
EB              EX DE,HL          DE: = endereço para
                                  onde deslocar os
                                  dados.
                                  HL: = endereço do
                                  primeiro byte a mover.
EDB0            LDIR              Move o bloco dos dados
                                  conforme requerido.
D5              PUSH DE
1808            JR HC__2
09      HC__1   ADD HL,BC
E5              PUSH HL
2B              DEC HL            HL: = endereço do últ.
                                  byte da nova posição
                                  para o código a mover.
EB              EX DE,HL          DE: = endr. do últ. byte
                                  do código na posição
                                  final. HL: = endr. do 1.º
                                  byte a ser movido
09              ADD HL,BC
2B              DEC HL            HL: = endereço do
                                  último byte de dados.
EDB8            LDDR              Move o bloco dos dados
                                  conforme requerido.
2AFCFF  HC__2   LD HL,(ADD3)      HL: = endereço do início
                                  do bloco deslocado.
ED5BF6FF        LD DE,(BEGIN)     DE: = primeiro byte do
                                  programa sujeito.
A7              AND A
ED52            SBC HL,DE
19              ADD HL,DE
3003            JR NC,HC__3
22F6FF          LD (BEGIN),HL     Redefine (BEGIN) se
                                  necessário.
```



```
E1      HC__3   POP HL
CDC4FE          CALL NL__2        Redefine (LIMIT) se
                                  necessário.
C370FF          JP EXIT
```

```
800 PRINT " Copy ";
810 GO SUB 8000
820 PRINT "Until ";
830 GO SUB 8020
840 PRINT "três espaços To".
850 LET X=65532
860 GO SUB 8030
870 RANDOMIZE USR 64981
```

E finalmente a secção TEXT:

```
                ORG FDB9
CDD1FE  TEXT    CALL SET__UP      BC: = comprimento da
                                  string A$, DE: = endr. a
                                  partir do qual iremos
                                  escrever a string.
                                  HL: = endr. anterior
                                  ao texto.
78              LD A,B
B1              OR C
CA70FF          JP Z,EXIT         Com string vazia, volta.
E5              PUSH HL
CD7AFF          CALL PR__ADDR     Escreve o endr.
E1              POP HL            corrente.
23              INC HL            HL aponta para o texto
                                  da string.
7E      T__LOOP LD A,(HL)         A: = próximo carácter
                                  do texto.
D7              RST 10            Escreve este carácter.
EDA0            LDI               Transfere este byte p. o
                                  programa.
EAC7FD          JP PE,T__LOOP     Repete até terminar.
ED53F8FF        LD (ADDRESS),DE   Guarda o endr. final.
C3C1FE          JP NEW__LIM       Ajusta (LIMIT) se
                                  necessário.
```

Repare-se no ciclo bastante estranho que aparece no processo: LDI seguido de JP PE. LDI pode ser visto como um passo de LDIR, ou alternativamente, como LD (DE),(HL), além de decrementar BC. O *flag* P funciona de uma maneira invulgar neste caso. Olhando rapidamente o que vem no Apêndice Quatro sobre LDI, vemos que o *flag* P se torna zero se e só se BC atingir finalmente zero. Precisamos portanto de uma instrução que signifique «salte se P é um»; JP PE faz exactamente isto. Note-se que neste caso não existe instrução JR PE, o que nos obriga a empregar um endereço absoluto. LDI foi usado em vez de LDIR pela necessidade de incluir RST 10 no ciclo.

O BASIC para acompanhar este código máquina é:

```
900 PRINT "Text ";
910 GO SUB 8000
920 CLEAR
930 INPUT USR 32631;CHR$ 8;LINE A$
940 RANDOMISE USR 64953
950 GO TO 930
```

Agora, uma curta sub-rotina para nos preparar o caminho antes de passarmos às funções definidas pelo utilizador. É uma sub-rotina que extrai o primeiro argumento de uma fracção definida pelo utilizador. Para recordar como os registos na área das funções definidas pelo utilizador são constituídas, é conveniente recapitular, lendo o fim do capítulo anterior.

```
                    ORG FDAE
2A0B5C  FN__ARG     LD HL,(DEFADD)   HL aponta p. a zona de
                                     parâmetros da função.

23                  INC HL
23                  INC HL
23                  INC HL
23                  INC HL
5E                  LD E,(HL)
23                  INC HL
56                  LD D,(HL)        DE: = prim. argumento.
C9                  RET
```


A primeira função que vamos definir empregando esta sub-rotina será chamada FN__H; a sua função é converter hexadecimal em decimal:

```
                    ORG FD8F
CDAEFD  FN__H       CALL FN__ARG     DE: = aponta p. a
                                     string.
23                  INC HL
46                  LD B,(HL)        B: = cm. da string.
210000              LD HL,0000       HL: = total (inicialmente
                                     igual a zero).
1A      FNH__LOOP   LD A,(DE)        A: = próximo carácter
                                     ASCII.
13                  INC DE           DE: = aponta para o
                                     próximo carácter.
FE40                CP 40
3804                JR C,FNH—CHR     Salta se o carácter
                                     estiver entre «0» e «9».
E6DF                AND DF           Converte em maiúscula
                                     se necessário.
D607                SUB 07           Corrige o código.
D630    FNH__CHR    SUB 30           A contém agora o dígito
                                     hex necessário.
29                  ADD HL,HL
29                  ADD HL,HL
29                  ADD HL,HL
29                  ADD HL,HL        Multiplica HL por
B5                  OR L             dezasseis.
6F                  LD L,A           Soma o novo dígito hex.
10EC                DJ NZ FNH__LOOP  Repete até terminar.
44                  LD B,H
4D                  LD C,L           BC: = valor calculado.
C9                  RET
```

E o BASIC é apenas:

```
9040 DEF FN H(A$)=USR 64911
```

Se não se conseguiu compreender o seu funcionamento — toda aquela evolução na área das funções definidas pelo utilizador — sugerimos uma nova leitura do fim do capítulo anterior. O processo

contrário é um pouco mais complicado, uma vez que não é possível retornar uma *string* como valor de USR. Temos de o fazer em duas fases. Chamaremos à primeira FN K$.

```
                ORG FD69
CDAEFD  FN__K$  CALL FN__ARG   DE: = primeiro
                                argumento da função.
010700          LD BC,0007
09              ADD HL,BC
46              LD B,(HL)       B: = segundo
                                argumento da função.
04              INC B
7B              LD A,E
05              DEC B
280B            JR Z,FNK__2
05              DEC B
2804            JR Z,FNK__1
7A              LD A,D
05              DEC B
2804            JR Z,FNK__2
1F      FNK__1  RRA
1F              RRA
1F              RRA
1F              RRA
E60F    FNK__2  AND 0F          A: = dígito a retornar.
FE0A            CP 0A
3802            JR C,FNK__3     Salta se entre «0» e «9».
C607            ADD A,07
C630    FNK__3  ADD A,30        Converte em carácter
                                ASCII.
4F              LD C,A
0600            LD B,00         BC: = cód. do cr.
C9              RET
```

E agora algumas funções úteis:

```
9010 DEF FN P$(A$) = FN H$(FN P(FN H(A$)))
9050 DEF FN H$(X) = FN K$(X,3) + FN K$(X,2) + FN K$(X,1)
+ FN K$(X,0)
9060 DEF FN K$(X,Y) = CHR$ USR 64873
```


Por fim os POKEs duplos:

```
                ORG FD5A
CDAEFD  FN__Q   CALL FN__ARG   DE: = endr. de POKE.
010700          LD BC,0007
09              ADD HL,BC
4E              LD C,(HL)
23              INC HL
46              LD B,(HL)       BC: = argumento de
                                POKE.
EB              EX DE,HL        HL: = endr. de POKE.
71              LD (HL),C
23              INC HL
70              LD (HL),B       POKE número de dois
                                bytes.
C9              RET
```

```
9020 DEF FN Q(X,Y) = USR 64858
9030 DEF FN R(A$,B$) = FN Q(FN H(A$),FN H(B$))
```

E agora um pouco de arrumação!

Se quisermos escrever REM *gráfico A gráfico B gráfico C ...gráfico U* teremos uma surpresa muito desagradável: todos os caracteres de gráficos foram corrompidos. (Na realidade foram substituídos por código máquina de HEXLD 3.) Podemos remediá-lo e ao mesmo tempo dar uma boa amostra de algumas das funções do nosso programa da seguinte maneira:

```
RUN 800
Copiar desde endr. 3E08
até endereço 3EAF
```

Estes *bytes* são as matrizes de *pixels* das maiúsculas A a U.
Isto é RAM livre.

```
Para endereço FCB2
RANDOMIZE FNR("5C7B",
"FCB2")
```

POKE duplo que carrega o número hex FCB2 (o endr. acima) nos endrs. 5C7B e 5C7C (a variável de sistema UDG) – endr. do início da zona dos caracteres gráfs. «p. uso do utilizador».

Agora se escrevermos REM *gráfico A gráfico B gráfico C ...gráfico U*, obteremos na realidade letras maiúsculas de A a U. Podemos utilizar agora os gráficos definidos pelo utilizador exactamente do mesmo modo de sempre, lembrando, é claro, querendo fazer POKE directamente (em vez de USR "A", etc.), que a zona correspondente começa no endereço FCB2.

Vamos tentar embelezar um pouco as coisas, mas não juntaremos mais nada — apenas queremos dar-lhe uma aparência mais bonita. Começaremos por melhorar a rotina em 8000. Apagamos todas as linhas de 8000 até 8999 e em seu lugar escrevemos o seguinte:

```
8000 GO SUB 8060
8010 RANDOMIZE FN R("FFF8",A$)
8020 RETURN
8030 GO SUB 8060
8040 RANDOMIZE FN R("FFFA",A$)
8050 RETURN
8060 INPUT "ENDEREÇO"; LINE A$
8070 IF LEN A$<>4 THEN CLEAR: GO TO 1E4
8080 PRINT "ENDEREÇO"; A$
8090 RETURN
```

Melhoramentos são: primeiro, o uso de FN R para fazer um POKE duplo ou em ADDRESS ou em ADD2; segundo, a verificação do comprimento de A$ — note-se o procedimento se a verificação falhar: CLEAR limpará a pilha GO SUB (obviamente muito importante, uma vez que nesta altura estamos com pelo menos dois GO SUB pendentes) e também limpa o *écran* e GO TO 1E4 (que significa GO TO 10000) é apenas um meio de voltar ao modo de comando (STOP teria feito exactamente a mesma coisa, mas o código de erro zero é mais estético que um código 9); e terceiro, não são referidas quaisquer outras variáveis na rotina, além de A$.

Porque a sub-rotina 8000 não requer senão A$ e também para fazer as listagens mais bonitas, apagamos as linhas 120, 220 e 920 (todas as que anteriormente diziam CLEAR).

Porque a rotina para definir ADD2 foi movida de 8020 para 8030, precisamos mudar:


```
350 GO SUB 8030
830 GO SUB 8030
850 GO SUB 8060
860 RANDOMIZE FN R("FFFC",A$)
```

E, para embelezamento da rotina SAVE:

```
410 SAVE"Hexld 3 mc"CODE64690,846      Não há necessidade
de guardar a mais, agora que sabemos o comprimento do programa.
440 GO TO 1E4                          Porque é mais bonito que STOP.
450 CLEAR 64689                        Porque não é preciso reservar
                                       mais memória que a necessária.
460 LET P=7: LET I=0                   Defina cor e fundo.
461 INK I
462 PAPER P
463 BORDER P
464 POKE 23624,I+8*P
465 FLASH 0
466 BRIGHT 0
467 OVER 0
468 INVERSE 0
469 CLEAR
470 LOAD "" CODE: LOAD "" CODE
471 RANDOMISE FN Q(23675,64690)        POKE duplo na variável
UDG.                                   de sistema UDG.
480 GO TO 1E4                          Em vez de STOP.
```

Sempre que usarmos HEXLD 3 para criar um novo programa devemos mudar as linhas 420 (para reflectir o nome do novo programa), 450 se precisarmos de reservar memória e 460 se quisermos um esquema de cores diferente.

Dispomos agora de um programa completo para editar código máquina. RUN 400 fará SAVE e VERIFY de tudo o que for necessário, incluindo os gráficos definidos pelo utilizador. Temos uma desculpa por avançarmos demasiado depressa: embora ainda não sejam de fácil compreensão algumas partes deste capítulo, o que interessa é que agora há uma ferramenta com a qual podemos

criar e manipular o nosso código máquina. Para referência, incluímos aqui uma lista de todos os endereços usados neste programa.

16K		48K		
7CB2	31922	FCB2	64690	Carac. gráfs. definíveis
7D5A	32090	FD5A	64858	FN__Q
7D69	32105	FD69	64873	FN__K$
7D8F	32143	FD8F	64911	FN__H
7DAE	32174	FDAE	64942	FN__ARG
7DB9	32185	FDB9	64953	TEXT
7DD5	32213	FDD5	64981	H__COPY
7E10	32272	FE10	65040	H__RUN
7E1C	32284	FE1C	65052	R-CH
7E3E	32318	FE3E	65086	BEGIN__MC
7E49	32329	FE49	65097	DELETE
7E63	32355	FE63	65123	INSERT
7E87	32391	FE87	65159	WRITE
7EC1	32449	FEC1	65217	NEW__LIM
7EC4	32452	FEC4	65220	NL__2
7EDI	32465	FED1	65233	SET__UP
7EE8	32488	FEE8	65256	LEN__A$
7EF0	32496	FEF0	65264	H__LIST
7F2A	32554	FF2A	65322	BREAKPT
7F70	32624	FF70	65392	EXIT
7F77	32631	FF77	65399	U__ADDR
7F7A	32634	FF7A	65402	PR__ADDR
7F7D	32637	FF7D	65405	PR__HL
7F89	32649	FF89	65417	I__CHECK
7F99	32665	FF99	65433	REGS
7FCB	32715	FFCB	65483	SH__PRINT
7FD0	32720	FFD0	65488	H__PRINT
7FED	32749	FFED	65517	C__PRINT
7FF6	32758	FFF6	65526	A variável BEGIN.
7FF8	32760	FFF8	65528	A variável ADDRESS.
7FFA	32762	FFFA	65530	A variável ADD2.
7FFC	32764	FFFC	65532	A variável ADD3.
7FFE	32766	FFFE	65534	A variável LIMIT.

CAPÍTULO ONZE

Exploração do teclado

Chegou agora a altura de vermos como se utilizam algumas das outras sub-rotinas, tão bem escondidas na ROM. Falaremos de três dessas sub-rotinas, que nos permitirão a leitura do teclado e localização da tecla, se houver alguma, que está a ser premida. Veremos também os processos pelos quais estas sub-rotinas funcionam, porque é sempre muito útil um pouco de conhecimento sobre o fundo das questões.

A primeira destas sub-rotinas que realiza uma leitura do teclado começa no endereço 0287. Podemos ter-lhe acesso chamando simplesmente esse endereço, isto é CALL KEY SCAN ou CD 8E02 em hex. Dá uma resposta satisfatória mas que ainda não é aquela que esperaríamos. Vamos ver exactamente o que ela faz. Esta rotina retorna um valor no par de registos DE. Na verdade, ela retorna valores separados e independentes, um para D e outro para E. Eis como estes valores são interpretados.

A figura 11.1 mostra o teclado do *Spectrum*. Como se vê, cada tecla está associada a um número. Todos os números se encontram entre 00 e 27, e uma vez que existem exactamente 28 teclas (em hex, note-se) e nenhum número se usa duas vezes, cada valor

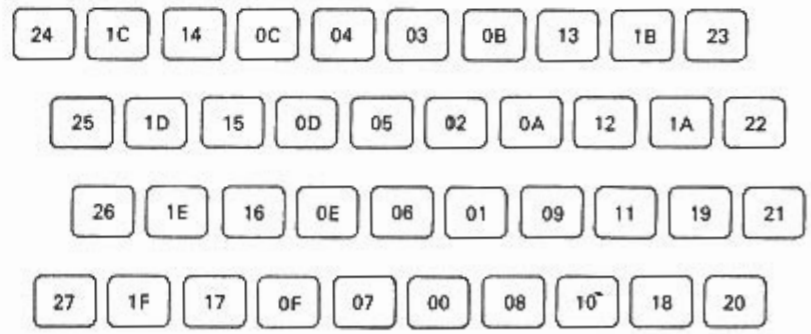


Figura 11.1

associado a uma tecla é único e exclusivo dessa tecla. Este número é o código de tecla *(Key code)*; por exemplo, o código de tecla de «A» é 26 e o código de tecla de «U» é ØA. Se experimentarmos ligar os pontos por uma linha, veremos que os números estão dispostos numa forma de espiral, em sentido contrário ao do andamento dos ponteiros do relógio, para fora, começando em «B». É um método muito útil de lembrar o código de teclado para qualquer tecla sem ter de voltar atrás à tabela a todo o momento.

O valor deixado em DE pela sub-rotina KEY SCAN baseia-se neste código de teclas. Se nenhuma tecla estiver premida, o valor em DE será FFFF. Suponhamos no entanto que uma, e só uma, estaria premida na altura em que se fez CALL KEY SCAN. Neste caso o valor de D continua a ser o mesmo FF, mas o valor de E será o código dessa tecla. Um ponto importante é que as teclas *caps shift* e *symbol shift* não são tratadas de modo diferente das outras teclas — *symbol shift* só por si produz FF18 da mesma maneira que «S» só por si produz FF1E.

Suponhamos agora que *caps shif* se mantém premida enquanto premimos uma outra tecla. Neste caso, o valor de D não será FF mas sim 27; note-se que este é o código da tecla *caps shift*. O registo E conterá o código da outra tecla que está a ser premida. Veja-se que, apesar de neste caso *caps shift* receber um tratamento diferente das outras teclas, o mesmo não se passa com *symbol shift*. Por outras palavras, *caps shift* e *symbol shift* carregados simultaneamente dão 2718, do mesmo modo que *caps shift* e «3» produzem 2714.

Por fim, se *symbol shift* mantiver premida enquanto premirmos outra tecla que não *caps shift*, o registo D conterá 18 e o registo E conterá o código da tecla em questão. Repare-se que 18 é o código da tecla *symbol shift*. Como *caps shift* é considerado «mais importante», ambos premidos ao mesmo tempo darão 2718 em vez de 1827.

A sub-rotina retorna mais um elemento importante de informação: o *flag* zero. Todos os casos acima mencionados terão como resultado o *flag* zero estar *set* (isto é, de modo que JR Z dará um salto), mas, se premirmos demasiadas teclas ao mesmo tempo, o *flag* zero estará *reset*. Se for este o caso, o valor retornado pelo par de registos DE será, em geral, sem sentido. É sempre prudente

verificar o *flag* zero antes de tomar em consideração o valor retornado em DE.

Deve já ser bastante claro que cada tecla individual produz um código único no par de registos DE, com ou sem qualquer dos *shifts*; no entanto, infelizmente não podemos determinar se, digamos, «Q», «B» e «T» estão todos premidos ao mesmo tempo. Em código máquina até isto é possível, por isso temos uma grande vantagem sobre o BASIC, onde só podemos utilizar INKEY$. Aprofundaremos a forma de utilizar este facto um pouco mais adiante no capítulo.

A sub-rotina KEY SCAN tem, no entanto, uma grande desvantagem: destrói por completo os valores anteriores de todos os registos! Se quisermos guardá-los temos de utilizar a pilha da seguinte maneira:

```
F5          PUSH AF
C5          PUSH BC
E5          PUSH HL
CD8E02      CALL KEY_SCAN
E1          POP HL
C1          POP BC
F1          POP AF
```

Queremos agora tornar estes números, um tanto obscuros, em verdadeiros códigos de carácter. Estes códigos de carácter são guardados numa tabela na ROM, começando no endereço 2Ø5. Na realidade existem aqui diversas tabelas, mas é a primeira que nos interessa. Chama-se KEY TABLE (código de tecla) e contém a lista das teclas dispostas de uma maneira já nossa conhecida:

```
Ø2Ø5 DEFM B H Y 6 5 T G V
Ø2ØD DEFM N J U 7 4 R F C
Ø215 DEFM M K I 8 3 E D X
Ø21D E-controle L O 9 2 S Z
Ø225 enter espaço P Ø 1 Q A
```

É a mesma ordem da numeração dos códigos de tecla, de modo que, para obtermos um resultado já com bastante sentido, basta interligar a sequência de instruções LD HL,Ø2Ø5/LD D,ØØ/ADD

HL,DE/LD A,(HL) que retorna em A o código de um carácter. Repare-se, no entanto, em dois pormenores desta tabela. Primeiro, as letras da tabela são apenas maiúsculas, às teclas numéricas correspondem apenas códigos de números (e não códigos das funções de edição); é um pouco como quando ligamos *caps lock*. Em segundo lugar, o carácter apresentado na posição correspondente à tecla *symbol shift* é *E-controle*, que é o carácter ØE (ver o Apêndice Cinco). Isto por duas razões: 1.ª, *symbol shift* não tem um código de carácter; 2.ª, as teclas *shift* são por vezes utilizadas para mudar para o modo E (modo de extensão).

Tendo feito uma leitura válida do teclado e decidido que não foram premidas demasiadas teclas, o problema seguinte é decidir se alguma tecla «real» foi premida. Lembremo-nos que nenhuma tecla produz FFFF, *caps shift* só por si dá FF27, enquanto *symbol shift* por si só dá origem a FF18. Estas três condições devem ser rejeitadas. A sub-rotina KEY TEST, a partir do endereço Ø31E, resolve este problema. Desde que DE comece com o resultado de uma leitura válida do teclado (como se descreveu um pouco acima), CALL KEY TEST fará uma série de coisas:

1.º B conterá o conteúdo anterior de D;
2.º D conterá zero;

Após o que uma de duas acções terá lugar:

1.º Se DE começar como FFFF, FF27 ou FF18, o registo A terminará com FF,27 ou 18 respectivamente e o *carry* será zero;
2.º Se DE não começar como FFFF, FF27 ou FF18, o registo A terminará com o código do *carácter de base* para a tecla desejada (isto é, o carácter apropriado, tirado da tabela KEY TABLE acima referida) e o *flag carry* será *set*.

Esta é portanto uma rotina bastante útil, mas existe uma terceira sub-rotina na ROM ainda mais útil. Esta sub-rotina chama-se KEY CODE (chave de código) e converte o carácter de base de qualquer tecla num verdadeiro código de carácter do *Spectrum*. Requer que os registos contenham os valores certos; as regras são:

O registo E tem de conter o carácter de base correcto.

Sem *shifts:*	B tem de conter FF.
Com *caps shift:*	B tem de conter 27.
Com *symbol shift:*	B tem de conter 18.

Para o modo G:	C tem de conter Ø2.
Para o modo E:	C tem de conter Ø1.
Para o modo K:	C tem de conter ØØ e D ØØ.
Para o modo L:	C tem de conter ØØ, D tem de conter Ø8 e o *bit* número três da variável de sistema FLAGS 2 tem de ser zero.
Para o modo C:	C tem de conter ØØ, D tem de conter Ø8 e o *bit* número três da variável de sistema FLAGS 2 tem de ser um.

Obtemos uma leitura correcta do teclado carregando C com a variável de sistema MODE e D com a variável de sistema FLAGS (não importa que D contenha ou não Ø8; qualquer que seja o seu valor, só o *bit* número três é tomado em consideração).

Estando na posse de toda esta informação, podemos escrever uma sub-rotina, muito curta, em código máquina que faz tudo aquilo que queremos (pelo menos em termos de leitura de teclado, utilizando estas sub-rotinas da ROM).

Eis a listagem:

```
CD8E02  SCAN    CALL KEY SCAN
200F            JR NZ,VOID
CD1E03          CALL KEY TEST
300A            JR NC, VOID
5F              LD E,A
0E00            LD C,00
1608            LD D,08
CD3303          CALL KEY CODE
A7              AND A
C9              RET
37      VOID    SCF
C9              RET
```

Se escrevermos esta sub-rotina em qualquer ponto da RAM, fazemos uma leitura imediata do teclado em qualquer momento empregando CALL SCAN, e lê-se o código do carácter no registo A. A sub-rotina retornará o *carry* em um se o resultado da leitura do teclado for nulo, isto é, se demasiadas teclas estavam premidas ou se uma das teclas *shift* estava premida isoladamente ou ainda se nenhuma tecla estava premida.

O endereço de KEY CODE é Ø333.

Observemos este caso por meio de um exemplo (que é a melhor maneira ou, pelo menos, a mais interessante). Vamos escrever um programa para imprimir no *écran* caracteres em tamanho grande. Para relembrar como é que os padrões dos caracteres são guardados na ROM, basta reler o Capítulo Sete; podemos agora dedicar-nos a uma programação a sério.

Devemos reparar que os códigos dos caracteres gráficos foram arranjados de uma forma bastante engenhosa. Como se sabe, cada símbolo gráfico é composto de quatro quartos de quadrado. Se atribuirmos a cada um quarto um número, 1, 2, 4 ou 8 (ver a figura 11.2), calcularemos o código de qualquer carácter gráfico somando apenas os números correspondentes aos quartos que têm a cor *ink* e somando 80 ao resultado. Por exemplo, o carácter gráfico 6 tem o quarto superior esquerdo e o quarto inferior direito ambos de cor *ink* (sendo 2 e 4 os números correspondentes a estes quartos); portanto o seu código é 2+4+80 ou 86. Se lermos o manual *Sinclair* ou o Apêndice Cinco deste livro veremos que é isso que se passa.

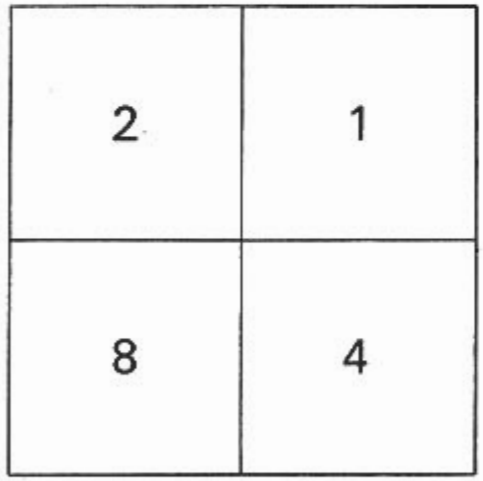


Figura 11.2

Vamos escrever a sub-rotina SCAN, listada acima, começando em qualquer endereço conveniente, e vamos introduzir o seguinte programa (imediatamente a seguir a SCAN, por exemplo):


```
AF          LARGE     XOR A
323C5C                LD (TVFLAG),A      Selecciona para PRINT
                                         a parte superior do écran.
210040      AT__0__0  LD HL,4000
22845C                LD (DF__CC),HL
212118                LD HL,1821
22885C                LD (S__POSN),HL    PRINT AT 0,0.
CD????      WAIT      CALL SCAN          Leitura do teclado.
38FB                  JR C,WAIT          Espera que uma nova
                                         tecla seja premida.
FE20                  CP 20
D8                    RET C              Retorna ao BASIC se for
FE80                  CP 80              premido um carácter não
                                         imprimível.
DO                    RET NC
6F                    LD L,A
2600                  LD H00             HL: = código do
                                         carácter premido.
29                    ADD HL,HL
29                    ADD HL,HL
29                    ADD HL,HL          Multiplica o código do
                                         carácter por oito.
11003C                LD DE,3C00
19                    ADD HL,DE          Soma 3C00 a este
                                         número (ver capítulo 7
                                         acerca deste assunto).
0E04                  LD C,04
0604        OUTER-    LD B,04
            LOOP
56                    LD D,(HL)          Transfere duas filas de
23                    INC HL             pixels para DE.
5E                    LD E,(HL)
23                    INC HL
3E08        INNER-    A,08               (08 tornar-se-á 80 ao ir
            LOOP                         quatro vezes à esquerda.)
CB13                  RL E               Calcula qual o carácter
17                    RLA                gráfico a ser escrito.
CB13                  RL E
17                    RLA
CB12                  RL D
17                    RLA
```

```
CB12        RL D
17          RLA
D7          RST 10          Escreve este carácter.
10EF        DJNZ INNERLOOP  Pr. posição a escrever.
3E0D        LD A,"enter"
D7          RST 10          Fim da linha corrente.
0D          DEC C
20E3        JR NZ,OUTERLOOP
18BE        JR AT_0_0
```

Repare-se naquelas instruções no início, a escreverem nas variáveis de sistema. Carregar DF_CC com 4000 e S_POSN com 1821 equivale a fazer (em BASIC) PRINT AT 0,0; na realidade o que acontece quando se encontra PRINT AT 0,0; é que se mudaram DF_CC e S_POSN. É a rotina RST 10 que utiliza estes valores para decidir onde imprimir.

Agora, se correrrmos este programa, escrevendo RANDOMIZE USR ????? (onde ????? é o endereço em decimal da etiqueta LARGE), o *écran* permanece inalterado. Carreguemos na tecla G e vejamos o que acontece. Agora carreguemos na tecla A.

É interessante, não é verdade? Se, por acaso, corremos este programa com *caps lock* ligado, não será tão divertido como se o tivéssemos desligado. Pode-se sair do programa premindo ambos os *shifts* juntos ou, na verdade, qualquer coisa que não possa ser impressa no espaço de um carácter; por exemplo, STOP.

Vamos por momentos pôr de parte a programação e dedicar-nos um pouco à aprendizagem. Como dissemos no início do capítulo, vamos aprofundar um pouco mais o processo de leitura do teclado, de modo a escrevermos as nossas próprias rotinas de leitura do teclado sem ter de recorrer às da ROM. A vantagem, claro está, é poder reescrevê-las de modo a permitir que várias teclas sejam premidas ao mesmo tempo. Isto passa-se como a seguir descrevemos.

O teclado está dividido fisicamente, apesar de não vermos essas divisões, em oito segmentos diferentes. Vemos como estes segmentos estão constituídos na figura 11.3. Cada segmento corresponde a um porto de entrada diferente, o que significa que temos de empregar instruções IN nas nossas rotinas.

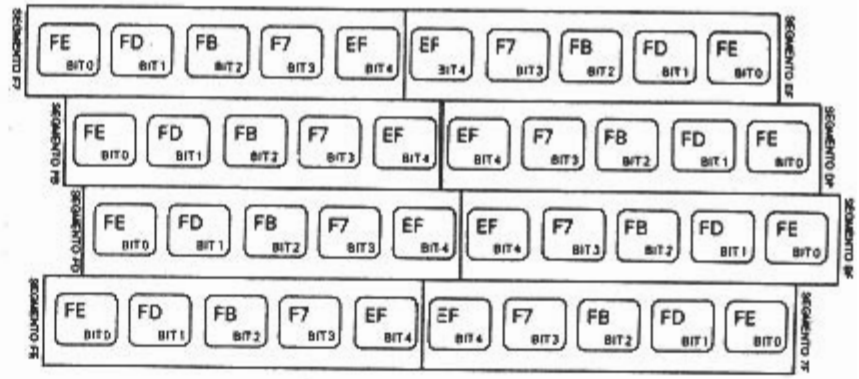


Figura 11.3

Gostaríamos de realçar bem o seguinte: na explicação que se segue, as teclas *shift* são consideradas exactamente em pé de igualdade com qualquer outra tecla. O código máquina não faz distinções. É o programa da ROM que testa e trata de maneira diferente estas teclas; sem empregar a ROM os *shifts* não são privilegiados. A tecla *symbol shift* deve ser a partir de agora vista como uma entidade física exactamente como, por exemplo, a tecla «X» — na verdade, se quiséssemos, podíamos muito bem programar uma rotina que faça uma leitura do teclado e considere a tecla «X» como um *shift!*

O teclado está portanto dividido em oito secções e só podemos ler uma delas de cada vez. Podemos fazê-lo de dois modos:

1.º Carregar A com o número de secção e depois utilizar a instrução IN A, (FE);

2.º Carregar B com o número de secção, C com o número FE e empregar a instrução IN r, (C) (onde r é um dos seguintes registos: A, B, C, D, E, H ou L).

Para fazermos uma leitura da secção BF, o método mais simples é LD A, BF e em seguida IN A, (FE). Esta acção deixa no registo A um valor. O significado desse valor é o seguinte:-os *bits* sete, seis e cinco deste resultado não têm interesse e, para nos descartarmos deles, utilizamos a instrução OR E0. Se nenhuma das teclas da secção BF estivesse premida na altura do IN A, (FE), após OR E0 A

ficaria com o valor FF. Se uma e uma só tecla estivesse premida na secção BF, quaisquer outras teclas doutras secções poderiam estar premidas, mas de momento não são consideradas. O valor agora contido no registo A é o número hex escrito na figura 11.3 na tecla premida. Se mais de uma tecla da secção FB estiver premida, o registo A conterá o resultado de um AND entre os valores para as diferentes teclas carregadas. Suponhamos que a tecla «J» esteja premida. A figura 11.3 atribui a esta tecla o número F7, de modo que se fizer a leitura do segmento BF como foi descrito, A terá o valor F7. Se em vez disso estiver premida a tecla «L», A fica com o valor FD. Agora, se «J» e «L» estiverem premidas simultaneamente, A conterá o valor F5 (=F7 AND FD).

Como se vê, a cada tecla está associado um *bit*. Para a tecla J é o *bit* 3 e para a tecla L é o *bit* 1. Quando fizermos a leitura de qualquer secção empregando IN, o resultado será, em binário, ???b4b3b2b1bØ. Os «b» serão um se a tecla correspondente não estiver carregada; caso contrário, serão zero. Uma vez desembaraçados dos três primeiros *bits* (com OR EØ), uma maneira simples de ver as coisas é imaginar que A começa com FF e depois tem um *bit reset* por cada tecla dessa secção que esteja premida.

GRAFFITI

É necessária apenas uma pequena modificação na versão original para obtermos um programa verdadeiramente interessante, demonstrando a imensa rapidez que o código máquina nos proporciona relativamente ao BASIC. Neste programa GRAFFITI, ao carregarmos numa tecla aparecerá no *écran* uma versão muito aumentada do carácter requerido.

Um outro aspecto interessante neste programa é que se utiliza PRINT AT de um modo geral em vez de definir as variáveis de sistema, como no anterior. Para obter PRINT AT D,E; só são necessários os seguintes passos LD A,"*control-AT*"/RST 1ØLD A,D/RST 1Ø/LD A,E/RST 1Ø. Note-se que este processo é muito semelhante à versão BASIC: PRINT CHR$ 22;CHR$ D;CHR$ E;. Segue-se a listagem do programa.

```
AF          GRAFFITI  XOR A
323C5C                LD (TVFLAG),A     Selecciona p. PRINT a
                                        parte superior do écran.
ED62                  SBC HL,HL         HL: = ØØØØ (as
                                        coordenadas de PRINT AT).
E5          MAIN      PUSH HL           Empilha coordenadas AT.
CD????      PAUSE     CALL SCAN         (Já listadas neste capít.)
30FB                  JR NC,PAUSE       Espera que o dedo
                                        largue a tecla.
CD????      WAIT      CALL SCAN
38FB                  JR C,WAIT         Espera que uma nova
                                        tecla seja premida.
FE20                  CP 20
3848                  JR C,EXIT         Termina se um carácter
                                        de controle for premido.
FE80                  CP 80
3044                  JR NC,EXIT        Termina se um token for
                                        premido.
6F                    LD L,A
2600                  LD H,00           HL: = cód. do carácter.
29                    ADD HL,HL
29                    ADD HL,HL
29                    ADD HL,HL
11003C                LD DE,3C00
19                    ADD HL,DE         HL: = aponta para a
                                        expansão em pixels do
                                        carácter requerido.
ØE04                  LD C,04
D1          OUTER-    POP DE            DE: = coordenadas de
            LOOP                        PRINT AT.
3E16                  LD A, «control-AT»
D7                    RST 10            PRINT AT...
7A                    LD A,D
D7                    RST 10            D,...
7B                    LD A,E
D7                    RST 10            E.
14                    INC D             Aponta p. a pr. linha abaixo.
D5                    PUSH DE           Empilha as coords. AT
                                        mais uma vez.
0604                  LD B,04
56                    LD D,(HL)         Transfere duas linhas de
                                        pixels para DE.
23                    INC HL
5E                    LD E,(HL)
23                    INC HL
```

3E08	INNER-LOOP	LD A,08	Isto tornar-se-á 80 quando deslocado.
CB13		RL E	
17		RLA	
CB13		RL E	
17		RLA	
CB12		RL D	Calcula qual o carácter
17		RLA	gráfico a escrever.
CB12		RL D	
17		RLA	
D7		RST 10	Escreve este símbolo.
10EF		DJNZ INNERLOOP	
0D		DEC C	
20DC		JR NZ,OUTERLOOP	
E1		POP HL	HL: = coordenadas AT.
7D		LD A,L	A: = n.º de coluna do quad. mais à esquerda do cr. grande já escrito.
FE1C		CP 1C	
2008		JR NZ,NEXT__CHR	Salta a não ser que seja o último carácter permitido nesta linha.
2E00		LD L,00	Número de coluna igual a zero. O número de linha já é o correcto.
7C		LD A,H	A: = número de linha.
FE14		CP 14	
C8		RET Z	Retorna se cinco linhas de caracteres grandes tiverem sido escritas.
18AF		JR MAIN	Repete para o próximo carácter.
1104FC	NEXT__CHR	LD DE,FC04	
19		ADD HL,DE	Move as coordenadas AT para o pr. carácter.
18A9		JR MAIN	E outra vez para o próximo carácter.
F1	EXIT	POP AF	«Limpa» a pilha.
C9		RET	

CAPÍTULO DOZE

Damas – primeira parte

Podemos agora introduzir e editar código máquina. Chegamos assim à altura de fazer qualquer coisa de útil e (esperamos) interessante: um jogo de damas. Devemos ter certos cuidados com este programa. Eis como deverá ser a parte em BASIC.

```
10  RAMDOMIZE USR qualquer coisa
20  INPUT LINE A$
30  RANDOMIZE USR qualquer coisa
```

Como se pode ver, a grande maioria do programa estará escrita inteiramente em código máquina. Queremos guardar o nosso código máquina no endereço mais baixo que pudermos. Se escrevermos RUN 700 e introduzirmos o endereço da rotina BREAKPT de HEXLD 3 (7F2A ou FF24), veremos imediatamente os valores iniciais de cada um dos registos. O valor inicial de DE é o endereço do primeiro *byte* livre da RAM (pode-se verificá-lo, juntando um *byte* ao programa BASIC — um espaço, por exemplo — e experimentando mais de uma vez: DE apontará um *byte* acima), o que é, obviamente, muito útil.

Uma vez que a área do BASIC está sujeita a frequentes variações de tamanho, não podemos utilizar este endereço para começo do nosso programa, pois se o fizermos será certamente destruído logo que carregarmos nalgum botão. Portanto, sugerimos que se guarde o código máquina de damas (DRAUGHTS) a partir do endereço 6800. Para iniciar, PRINT FN H ("6800") diz-nos que esse endereço inicial é, em decimal, 26624 e, portanto, vamos ajustar HEXLD 3 para nos prepararmos para este programa. Escreve-se ou edita-se o seguinte:

```
420  SAVE «Damas» CODE FN P(32758),FN P(32766) – FN
     P(32758) + 1 (para aqueles que têm 16 K)
420  SAVE «Damas» CODE FN P(65526),FN P(65534) – FN
     (65526) + 1 (para aqueles que têm 48 K).
450  CLEAR 26623
```

```
CLEAR 26623                    (como um comando directo).
460  LET P=5: LET I=Ø
     RUN 46Ø                   (para fazer fundo cyan e cor preta).
```

Será necessário carregar na tecla de espaço (para sair na linha 470) antes de podermos fazer mais qualquer coisa.

Por fim, vamos fazer RUN 5ØØ, introduzir 68ØØ e escrever qualquer coisa — ØØ, por exemplo. Saímos premindo a tecla ENTER só por si. Nesta altura, gravamos o programa empregando RUN 4ØØ, apesar de na realidade ainda não termos começado, porque é bom guardar em fita o que é, de facto, um programa editor do programa de damas e também porque assim não haverá necessidade de voltar a escrever as linhas acima indicadas.

A primeira parte do programa já nós vimos. É a que, no Capítulo Sete, imprimia um tabuleiro de damas. Em primeiro lugar, será preciso redefinir o carácter *gráfico A*. Para isto, introduz-se como um comando directo FOR I = Ø TO 7: INPUT X: POKE USR "A" + I,X:NEXT I e em seguida os seguintes números: Ø/6Ø/126/126/126/126/6Ø/Ø. Agora se escrevermos REM *gráfico A* veremos aparecer a forma de uma pequena gota. Escreve-se RUN 1ØØ e introduz-se o endereço 6837 (note-se: 6837 e não 68ØØ — por razões que, mais tarde, se tornarão claras). Introduz-se agora a rotina que imprime o tabuleiro de damas, apresentada no Capítulo Sete; teremos de a escrever toda de novo e lembrando que o endereço S_PRINT é agora 6837 e não 7ØØØ, de modo que as instruções CALL S_PRINT sejam traduzidas por CD3768 e não CDØØ7Ø.

Podemos facilitar as coisas parando onde começa a zona de dados e depois alterar a linha 93Ø de modo a que fique INPUT USR 32631;CHR$ 8;A$ (16 K) ou INPUT USR 65399;CHR$ 8;A$ (48 K) (isto é, apagando a palavra LINE).

Podemos então introduzir todo o texto utilizando RUN 9ØØ. Para fazer entrar no texto um carácter *enter* é necessário mover o cursor para a direita das aspas e escrever +CHR$ 13. O endereço de início do texto é 6867.

RUN 9ØØ.

Endereço do texto 6867
"espaço 12345678" + CHR$ 13
"1 *espaço A gráfico espaço A gráfico espaço A gráfico espaço A gráfico* 1" + CHR$ 13... e assim por diante até...
"espaço 12345678" + CHR$ Ø E CHR$ FN H("C9").

Agora se fizermos RUN 7ØØ e introduzirmos 684Ø, veremos aparecer um tabuleiro de damas ainda um pouco primitivo. Juntemos mais uma linha de BASIC: 715 CLS, para nos livrarmos da mensagem "RUN ADDRESS" quando utilizarmos RUN 7ØØ. Se fizermos novamente RUN 7ØØ e introduzirmos 684Ø veremos a mesma imagem do tabuleiro sem a mensagem "RUN ADDRESS".

Vamos introduzir a seguinte sub-rotina a partir do endereço 684Ø:

```
3E04      ROW      LD A,04
71        ROW__2   LD (HL),C      Define a cor da próxima
                                  peça de damas.
23                 INC HL
23                 INC HL        HL aponta para a próxima
                                  peça de damas.
3D                 DEC A
20FA               JR NZ,ROW__2
19                 ADD HL,DE     Aponta p. a pr. linha.
CB45               BIT Ø,L
2002               JR NZ,ROW__3
23                 INC HL        O ponteiro deve ser
23                 INC HL        ajustado p. as linhas pares.
10EF      ROW__3   DJNZ ROW      Repete para o número
                                  necessário de linhas.
C9                 RET
```

Agora introduzimos o seguinte código máquina a partir do endereço 68C5 (imediatamente antes da última instrução RET):

```
210058             LD HL,ATTRS   HL aponta p. atributos.
111600             LD DE,0016
ØEØA               LD C,ØA
Ø6ØA      YELLOW   LD B,ØA
          __1
```

```
3630    YELLOW__2   LD (HL),30          Cor do pr. quadrado: preto em amarelo.
23                  INC HL              Aponta o pr. quadrado.
10FB                DJNZ YELLOW__2
19                  ADD HL,DE           HL aponta p. o começo da pr. linha.
0D                  DEC C
20F5                JR NZ,YELLOW__1
212158              LD HL,ATTRS+21      HL aponta p. o prim. quadrado do tabuleiro.
1E18                LD E,18
0E08                LD C,08
0608    WHITE__1    LD B,08
3638    WHITE__2    LD (HL),38          Cor do quadrado: preto sobre branco.
23                  INC HL              Aponta p. o pr. quadrado.
10FB                DJNZ WHITE__2
19                  ADD HL,DE           Indica HL p. o prim. quadrado da pr. fila.
0D                  DEC C
20F5                JR NZ,WHITE__1
212258              LD HL,ATTRS+22      HL aponta para o primeiro quadrado preto.
1D                  DEC E
010703              LD BC,0307          B: = número de linhas com a cor a definir.
                                        C: = cor da peça de damas.
CD4068              CALL ROW            Faz três filas brancas.
010002              LD BC,0200
CD4068              CALL ROW            Faz duas filas pretas.
010203              LD BC,0302
CD4068              CALL ROW            Faz três filas vermelhas.
```

Se fizermos RUN 700 e introduzirmos 6852, o tabuleiro aparecerá completo, em cores.

Vamos agora tratar do programa; é a *segunda* instrução RANDOMIZE USR. O que queremos que aconteça é que a linha 710 seja repetida logo que o controle retorne a BASIC, além de precisarmos de uma maneira de fazer BREAK. O programa entrará

num ciclo infinito se não encontrarmos um meio, em código máquina, de interromper a sua execução. Vamos continuar com as damas. Eis a rotina de que precisamos e que nos permitirá sair e retornar ao modo de comando do BASIC. Se o primeiro carácter introduzido (na linha 710 do programa BASIC) for STOP — *(symbol shift A)* e se não for este o caso, asseguraremos que, no retornar ao BASIC, a próxima instrução executada seja a linha 710. Esta rotina, que deve ser escrita a partir do endereço 6901, contém ainda mais uns artifícios que iremos ver pormenorizadamente mais adiante.

```
2A4B5C  MOVE    LD HL,(VARS)        HL aponta para a variável BASIC A$.
23              INC HL
5E              LD E,(HL)           E: = comprimento de A$.
23              INC HL
23              INC HL              HL aponta p. o texto de A$.
7E              LD A,(HL)
FEE2            CP "STOP"
2002            JR NZ,A$__OK
CF              RST 08
FF              DEFB FF             Retorna ao BASIC se A$ começar com «STOP».
21C602  A$__OK  LD HL,710d          HL: = o n.º de linha 710.
22425C          LD (NEWPPC),HL      Especifica salto p. linha 710.
3E01            LD A,01
32445C          LD (NSPPC),A        Especifica a primeira instrução da linha.
C9              RET
```

O BASIC pode agora ser actualizado; de outro modo, saltar para a linha 710 não fará sentido. Corrigem-se todas as linhas BASIC entre 700 e 799 como se segue:

```
700 RANDOMIZE USR 26706
710 INPUT LINE A$
720 RANDOMIZE USR 26881
```

Se escrevermos RUN 7Ø0, o que primeiro vemos é aparecer o já familiar tabuleiro de damas e também qualquer coisa de novo — um *prompt* de INPUT. Veremos que, independentemente do que se introduzir, o resultado será novamente o mesmo *prompt* de INPUT. O que se passa é que a linha 71Ø está a ser executada a cada retorno da rotina USR da linha 72Ø — imagine-se um GO TO 71Ø escondido no código máquina. Para interromper o programa, introduzimos STOP *(symbol shift A)*. Agora, vamos ver como este "GO TO 71Ø" automático apareceu. Para consegui-lo atribuímos valores a duas variáveis de sistema. A primeira tem o nome de NEWPPC e guarda o número da próxima linha do programa BASIC a ser executada. Se tivéssemos carregado esta variável com, digamos, 1ØØØd, ao retornar ao BASIC a próxima linha a executar seria a linha 1ØØØ. De facto, podemos ter um controle ainda mais fino, porque a variável de sistema NSPPC guarda o número da instrução, no interior da linha, a partir do qual a execução continua, de modo que podemos simular "GO TO 1ØØØ, instrução 4", se quisermos. É importante notar que este "GO TO" não tem efeito imediato. O *Spectrum* continuará a executar o código máquina até atingir um RET final. Então, e só então, os valores de NEWPPC e NSPPC serão tidos em conta pelo interpretador de BASIC, na ROM, efectuando o salto.

É agora que o programa vai começar a sério. Partimos do princípio de que uma jogada foi introduzida como A$, que é a primeira variável da zona de variáveis. Explicamos como introduzir uma jogada (ver a figura 12.1). Existem 64 quadrados, mas só em 32 deles podemos jogar (os pretos). Cada quadrado tem coordenadas de 11 a 88. Repare-se como são escritos sem vírgulas a separá-los. O primeiro dígito refere-se ao número escrito nos lados do tabuleiro e o segundo, ao número no topo (ou base). Há quatro diferentes direcções possíveis para o movimento das peças. Serão chamadas A (para cima à esquerda), B (para cima à direita), C (para baixo à direita) e D (para baixo à esquerda). Para introduzir um movimento é só escrever as coordenadas iniciais e uma letra (A, B, C ou D). Deve-se introduzir estes valores sem espaços entre eles. Por exemplo, para nos movermos do quadrado 61 para o quadrado 52 é preciso introduzir 61B. Para introduzir um duplo salto a partir do quadrado 65, primeiro na direcção A e depois na direcção B, introduzimos 65AB.

Para o computador fazer o seu trabalho necessita de uma zona de memória que possa interligar como uma espécie de bloco de notas. A área que escolhemos para este «espaço vago» é a que se encontra entre os endereços 68ØØ e 6836. Para preencher este espaço de trabalho (que será chamado BOARD__2) precisamos inserir algum código máquina. Vamos inseri-lo no endereço 69Ø1 e depois passamos à sua explicação.

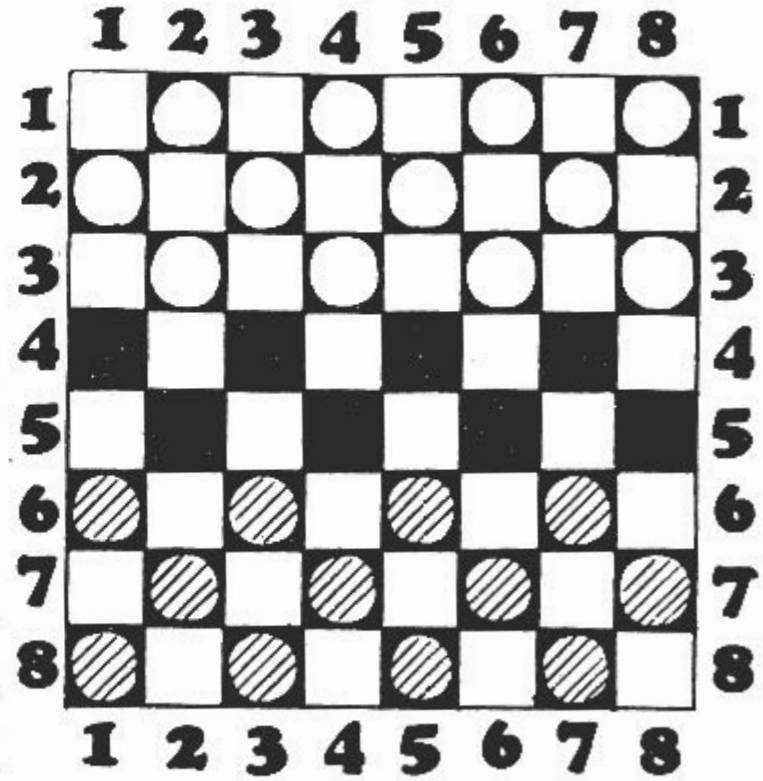


Figura 12.1a

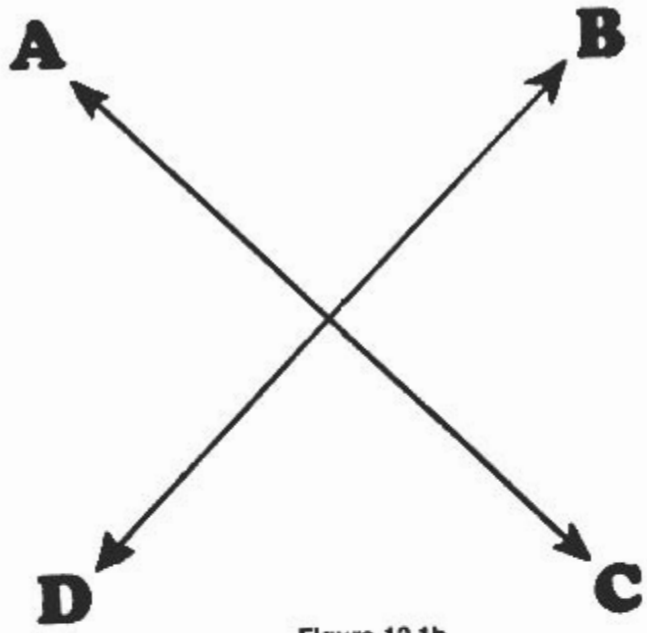


Figura 12.1b

```
212258 COPY__B      LD HL,ATTRS+22   HL aponta p. o prim.
                                     quad. onde pode jogar.
110668              LD DE,BOARD__2+06 DE aponta p. o prim.
                                     quadrado útil no
                                     espaço de trabalho.

0E04                LD C,04
0604   C__LOOP__A   LD B,04
EDA0   C__LOOP__B   LDI               Transfere um quadrado.
03                  INC BC            Restaura BC.
23                  INC HL            Aponta o pr. quadrado.
10FA                DJNZ C__LOOP__B
13                  INC DE            Salta um quadrado no
                                      espaço de trabalho.

D5                  PUSH DE
111700              LD DE,0017
19                  ADD HL,DE         Aponta para o pr.
                                      quadrado no écran.

D1                  POP DE
0604                LD B,04
EDA0 C__LOOP__C     LDI               Transfere um quadrado.
```



```
03                  INC BC            Restaura BC.
23                  INC HL            Aponta o pr. quadrado.
10FA                DJNZ C__LOOP__C
13                  INC DE
13                  INC DE            Salta dois quadrados
                                      no espaço de trabalho.

D5                  PUSH DE
111900              LD DE,0019
19                  ADD HL,DE         Aponta o próximo
                                      quadrado no écran.

D1                  POP DE
0D                  DEC C
20DE                JR NZ,C__LOOP__A
```

Para compreendermos esta sub-rotina é necessário conhecer tudo acerca da área de memória que chamamos «espaço de trabalho» e da forma como o programa a utiliza. Vejamos agora a figura 12.2, que mostra claramente o que acontece. Cada quadrado do *écran*, pelo menos os pretos, corresponde a um quadrado do espaço de trabalho, de modo que o computador tem, na realidade, uma cópia do tabuleiro numa outra parte da memória. Como se vê, além de haver um *byte* para cada quadrado, existem também *bytes* representando quadrados «fora das margens» — quadrados ilegais. Destinam-se a impedir que as peças saiam dos limites do tabuleiro. Nestas circunstâncias, a rotina acima dada não tem muita utilidade, porque se não preenche os quadrados vazios, também não os apaga, de modo que temos de a ajudar um pouco. Empregando RUN 10Ø, preenchemos todos os *bytes* entre 68ØØ e 6836 com FF. Feito isto, corremos o programa (RUN 7ØØ) e interrompemo-lo introduzindo STOP. Se, a seguir, listarmos a área de memória entre 68ØØ e 6836, veremos que corresponde exactamente ao diagrama da figura 12.2 com FF representando os quadrados ilegais, Ø7 as peças brancas, Ø2 para as peças pretas (ou vermelhas) e ØØ para os espaços vazios. Esta demonstração deve provar que a rotina acima dada está a cumprir a sua função. Vamos tentar acompanhar o seu funcionamento e ver o que acontece. Um ponto importante a notar é que a posição das peças é dada através da zona dos atributos, e não do *display file*. Cada quadrado que se possa jogar no *display file* contém um carácter *gráfico A*, de modo que seria inteiramente inútil

procurá-los aí; é preferível olhar para a zona dos atributos e assim distinguir as diferentes peças e os quadrados vazios apenas pelas cores que contêm.

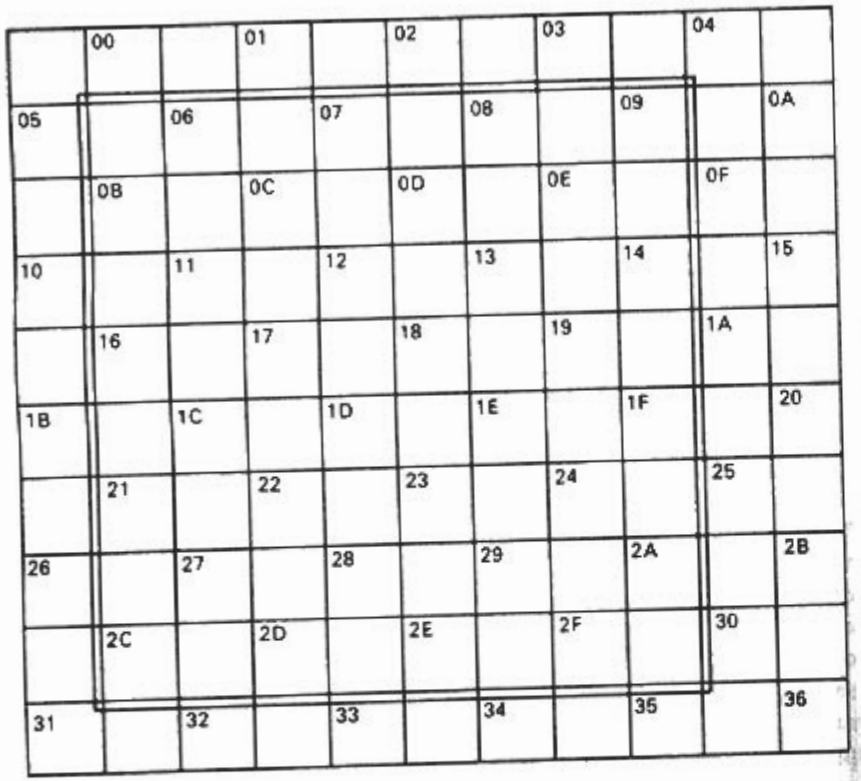


Figura 12.2

Há no entanto um ponto invulgar nesta rotina: as instruções INC BC, que são necessárias porque LDI não só carrega de (HL) a (DE), incrementando ambos os pares de registos, mas decrementando também BC, pelo que as instruções INC BC vêm logo a seguir para restaurar o seu valor inicial: o que se vai fazer a seguir parece muito estranho. É isto: PRINT AT 1,11; TAB Ø;. Tem por fim apagar a segunda linha do *écran* a partir do décimo segundo carácter para a

frente. Mas porquê? Já o veremos. Uma vez que devemos fazer isto mesmo no início da parte principal do nosso código máquina, inserimos o seguinte a partir do endereço 69Ø1:

```
AF          BLANK   XOR A
323C5C              LD (TVFLAG),A      Escreve na parte superior
                                       do écran.
3E16                LD A, «controle AT»
D7                  RST 1Ø
3EØ1                LD A,Ø1
D7                  RST 1Ø
3EØB                LD A,ØB
D7                  RST1Ø              PRINT AT 1,11;
3E17                LD A, «controle TAB»
D7                  RST 1Ø
AF                  XOR A
D7                  RST 1Ø
AF                  XOR A
D7                  RST 1Ø             PRINT TAB Ø;
```

Segue-se mais um refinamento. Recordemos que a parte de código máquina do fim do programa, que introduzimos até agora e que prepara GO TO 71Ø se A$, não começa com STOP. A parte do programa que vamos escrever em seguida depende de HL apontar o início do texto de A$ e, portanto, se quisermos que tudo isto funcione, precisamos de salvaguardar HL. Para o fazer, escrevemos o que se segue. É útil verificar se os endereços estão correctos.

```
RUN 2ØØ/694D/E5//      ("/" significa «enter». E5 é o código
                        hex para PUSH HL).
RUN 2ØØ/6954/E1//      (E1 é o código hex para POP HL).
```

E agora estamos prontos para a próxima secção, que se destina a verificar a velocidade das entradas: 1.º que a *string* tem pelo menos três caracteres de comprimento; 2.º que tanto o primeiro como o segundo caracteres são ambos dígitos entre um e oito. Paralelamente, será calculado o número do quadrado, segundo a convenção da figura 12.2. Esta rotina segue-se imediatamente à que verifica se A$=STOP e assim torna-se no último código máquina do programa. Insere-se o seguinte a partir do endereço 695A:

```
7B          VALID_CH LD A,E          A: = comprimento da
                                     string A$.
FE03                 CP 03
381A                 JR C,INVALID    Erro se LEN A$ for
                                     menor que três.
7E                   LD A,(HL)       A: = primeiro carácter
                                     da string.
FE31                 CP "1"
3815                 JR C,INVALID
FE39                 CP "9"
3011                 JR NC,INVALID   Erro se o c. não for um
                                     dig. entre «1» e «8».
47                   LD B,A
87                   ADD A,A         Multiplica o código do
                                     carácter por dois.
87                   ADD A,A         ... por quatro.
80                   ADD A,B         ... por cinco.
87                   ADD A,A         ... por dez.
80                   ADD A,B         ... por onze.
47                   LD B,A
23                   INC HL          Aponta para o segundo
                                     carácter.
7E                   LD A,(HL)       A: = segundo carácter.
FE31                 CP "1"
3804                 JR C,INVALID
FE39                 CP "9"
3817                 JR C,VALID      Erro se não for um dígito
                                     entre «1» e «8».
CD3768    INVALID    CALL S_PRINT    Escreve mensagem.
16010B               DEFM AT 1,11d
1106                 DEFM fundo amarelo

454E545241444120     DEFM ENTRADA
494C4547414C         DEFM ILEGAL
00                   DEFB 00         Fim de mensagem.
C9                   RET
80          VALID    ADD A,B
CB3F                 SRL A
C6E0                 ADD A,E0
C9                   RET
```


Decerto se observou, ao correr o programa, que qualquer entrada cujos primeiros dois dígitos sejam ambos números de «1» a «8» é aceite e quaisquer outros não, aparecendo no *écran* a mensagem INVALID ENTRY. Percebe-se agora o porquê de PRINT AT 1,11; TAB Ø; no início da rotina. A mensagem anterior tem de ser sempre apagada. O código construído no registo A é o número do quadrado introduzido. Veja-se como ele é calculado. Consiste no código de carácter do primeiro dígito multiplicado por onze (porque a largura do tabuleiro no espaço de trabalho é de onze quadrados) mais o código de carácter do segundo dígito. Este número é então dividido por dois (porque só os quadrados pretos têm números) e finalmente é adicionada a constante EØ, que acontece ser o número certo para converter «11» em Ø6, o número correspondente na área de trabalho.

A rotina seguinte é uma outra verificação — desta vez, se o quadrado é de facto um quadrado negro. Começa logo a seguir à instrução SRL A no fim da rotina anterior. Empregamos WRITE (RUN 1ØØ) para fazer entrar o seguinte, com ou sem RUN 9ØØ. Deve ser introduzido a partir de 6993:

```
381C                 JR C,SQ_BLACK
CD3768               CALL S_PRINT
16010B               DEFM AT 1,11d
1106                 DEFM fundo amarelo
4553534120           DEFM ESSA
43415341204520       DEFM CASA E
4252414E4341         DEFM BRANCA
00                   DEFB 00
C9                   RET
C6E0        SQ_BLACK ADD A,E0
C9                   RET
```

Se corrermos agora o programa, veremos ambas as verificações em acção. Experimente o leitor quebrar uma das regras para ver o que acontece. Segue-se a verificação de que o quadrado indicado contém, realmente, uma das suas peças, de modo que não possa fazer batota movendo uma peça do computador em vez de uma sua. Escreva-a a partir do endereço 69B3:

```
E5                      PUSH HL         Empilha ponteiro para o
                                        segundo cr. de A$.
6F                      LD L,A
2668                    LD H,68         HL aponta p. o quadrado
                                        do tabuleiro cópia, no
                                        espaço de trabalho
7E                      LD A,(HL)       A: = conteúdo deste
                                        quadrado.
E67F                    AND 7F
201F                    JR NZ,PIECE     Salta se o quadrado
                                        estiver ocupado.
CD3768                  CALL S_PRINT
16010B                  DEFM AT 1,11d
1106                    DEFM fundo amarelo
4553534120434153412Ø    DEFM ESSA CASA
4553544120              DEFM ESTA
56415A4941              DEFM VAZIA
ØØ                      DEFB ØØ
E1                      POP HL
C9                      RET
FE02        PIECE       CP 02
281A                    JR Z,PIECE_OK   Salta se uma peça do
                                        homem for escolhida.
CD3768                  CALL S_PRINT
16010B                  DEFM AT 1,11d
1107                    DEFM fundo branco
412Ø5Ø45                DEFM A PEÇA
2Ø4527204D494E4841      DEFM É MINHA
ØØ                      DEFB ØØ
E1                      POP HL
C9                      RET
E1          PIECE_OK    POP HL
C9                      RET
```

Feito o que havia a fazer no que respeita ao quadrado de partida, chegou o momento de pensarmos nas direcções de movimento. O código seguinte vai para 69F9 (apagando as últimas instruções, POP HL e RET).

```
1D                      DEC E
1D                      DEC E           E: = número de
                                        movimentos a fazer.
```



```
1D                      DEC E
2001                    JR NZ,NOT_ZERO
1D          LOOP_P1     DEC E
E3          NOT_ZERO    EX (SP),HL      HL: = aponta para o
                                        byte anterior ao da
                                        próxima direcção de
                                        movimento.
23                      INC HL          HL: = aponta para a
                                        nova direcção.
7E                      LD A,(HL)       A: = dir. do movimento.
E6DF                    AND DF          Converte em maiúscula.
FE41                    CP "A"
3804                    JR C,INV_DIR
FE45                    CP "E"
381C                    JR C,VAL_DIR    Erro se a dir. não for
                                        «A», «B», «C» ou «D».
CD3768      INV_DIR     CALL S_PRINT
16010B                  DEFM AT 1,11d
1106                    DEFM fundo amarelo
2Ø444952454343414F2Ø    DEFM espaço DIRECÇÃO

494C4547414C2Ø          DEFM ILEGAL espaço
ØØ                      DEFB ØØ
E1                      POP HL
C9                      RET
E1          VAL_DIR     POP HL
C9                      RET
```

Observe-se o tratamento curioso dado ao registo E no início dessa rotina. E começa, como já dissemos, com o comprimento da *string* A$. Ora, o número de movimentos introduzido é simplesmente este comprimento menos dois, e por isso antes de mais são-lhe subtraídas duas unidades. No entanto, acontece em seguida que E é decrementado uma outra vez e, mais ainda, se E contivesse ØØ, o seu valor mudaria para FF. Isto significa, na prática, que para movimentos e saltos normais E conterá FF, mas para saltos múltiplos E conterá um menos que o número de saltos a fazer. Isto destina-se a verificar se o jogador não está a fazer batota. Se, por exemplo, decidisse fazer um movimento múltiplo no qual nem todos os movimentos fossem saltos, detectá-lo-íamos com muita facilidade.

A direcção é agora «A», «B», «C» ou «D». É óbvio que não é esta a forma final na qual a desejamos. O que queremos saber é o deslocamento do quadrado inicial ao quadrado final. Por exemplo, suponhamos que a direcção do movimento era «C» — neste caso a deslocação seria Ø6, já que somando Ø6 ao número de qualquer quadrado (figura 12.2) obteremos o número do quadrado resultante do movimento na direcção C. Os deslocamentos de que necessitamos são: «A»=FA, «B»=FB, «C»=Ø6 e «D»=Ø5.

Há ainda uma outra verificação a fazer nas direcções: saber se uma peça é movida para a frente se para trás. Neste jogo, só as damas podem andar para trás. Nesta versão do jogo, as damas são representadas por caracteres a piscar — isto é, caracteres com o *bit* sete do seu *byte* de atributos *set*. A rotina seguinte faz aquilo que queremos. Escrevemo-la a partir do endereço 6A28:

```
1606              LD D,06       D conterá a deslocação
3D                DEC A         A: = 40,41,42 ou 43
CB3F              SRL A         A: = 20 ou 21
3001              JR NC,DIS__1  Salta se a direcção for «A»
                                ou «C».
15                DEC D         D: = Ø6 para «A» ou «C»,
                                Ø5 para «B» ou «D».
FE20    DIS__1    CP 20
7A                LD A,D
2002              JR NZ,DIS__2  Salta se a direcção for «C»
                                ou «D».
ED44              NEG           A: = desl. correcto.
57      DIS__2    LD D,A        D: = desl. correcto.
E3                EX (SP),HL    HL: = aponta p. quad. no
                                espaço de trabalho.
E680              AND 80        A: = 80 p. um movimento
                                p. a frente ou ØØ se for p.
                                trás.
B6                OR (HL)
FE82              CP 82         Verifica se existe um
                                movimento p. trás ilegal.
2819              JR Z,DIS__OK
CD3768  B__MOVE   CALL S__PRINT
16010B            DEFM AT 1,11d
1107              DEFM fundo branco
2050415241 20     DEFM espaço PARA
```



```
5452412753203F20  DEFM TRÁS? espaço
ØØ                DEFB ØØ
E1                POP HL
C9                RET
E1      DIS__OK   POP HL
C9                RET
```

Pouco a pouco, estamos quase a ficar sem coisas para eliminar. A rotina seguinte esperamos que seja evidente. Escrevemo-la a partir de 6A59:

```
4E                LD C,(HL)     C: = peça humana.
3600              LD (HL),ØØ    Apaga a peça da
                                posição inicial.
7D                LD A,L
82                ADD A,D
6F                LD L,A        HL: = aponta o
                                quadrado de destino.
7E                LD A,(HL)     A: = conteúdo do
                                quadrado de destino.
FEFF              CP FF
2019              JR NZ,ON__BOARD
CD3768  OFF__BOARD CALL S__PRINT

16010B            DEFM at 1,11d
1106              DEFM fundo amarelo
2053414920        DEFM espaço SAI
444F20            DEFM DO espaço
4A4F474F2120      DEFM JOGO! espaço
ØØ                DEFB ØØ
E1                POP HL
C9                RET
E67F    ON__BOARD AND 7F        Ignora o status
                                da dama.
FE02              CP Ø2         Procura peça humana.
2019              JR NZ,M__1
CD3768  SQ__BL    CALL S__PRINT
16010B            DEFM at 1,11d
1107              DEFM fundo branco
4341534120        DEFM CASA
```

```
424C4F515545414441 DEFM BLOQUEADA
00                 DEFB 00
E1                 POP HL
C9                 RET
E1          M__1   POP HL
C9                 RET
```

A rotina seguinte completa a secção de movimentos, transferindo a peça para o lugar onde desejamos movê-la, embora, neste momento, esta nova posição seja guardada na cópia do tabuleiro. A rotina faz ainda duas verificações finais: que cada um dos passos numa sequência de saltos múltiplos é na verdade um salto e não uma simples deslocação; e que uma peça que tenha acabado de atingir o grau de dama não pode fazer um movimento para trás na mesma jogada em que chegou a dama. Este segmento praticamente completa DAMAS — PRIMEIRA PARTE. Deve ser escrito a partir do endereço 6A9C:

```
A7                 AND A
2024               JR NZ,JUMP          Movimento ou salto?
7B                 LD A,E
3C                 INC A
282C               JR Z, CONTINUE
CD3768             CALL S__PRINT
16010B             DEFM at 1,11d
1401               DEFM inverse 1
4E414F20504F444520 DEFM NÃO PODE
53414C5441522Ø     DEFM SALTAR
41474F5241         DEFM AGORA
00                 DEFB 00
E1                 POP HL
C9                 RET
3600        JUMP   LD (HL),00          Apaga a peça do comp.
7D                 LD A,L
82                 ADD A,D
6F                 LD L,A              HL: = quad. de destino.
7E                 LD A,(HL)           A: = conteúdo do
                                       quadrado de destino.
3C                 INC A
2898               JR Z,OFF__BOARD     Erro se quad. fora do tab.
```



```
3D                 DEC A
20B4               JR NZ,SQ__BL        Erro se quad. bloqueado.
71          CONTINUE LD (HL),C         Coloca a peça do jogador
                                       humano no novo quad.
1C                 INC E
2804               JR Z,EP__1
1D                 DEC E
C2FE69             JP NZ,LOOP__P1
E1          EP__1  POP HL
C9                 RET
```

E chegamos agora ao fim da primeira parte. Esta secção encarrega-se da promoção a dama das peças chegadas à última linha e em seguida (aquilo que já esperávamos) termina, escrevendo o tabuleiro actualizado. Escrevemos o seguinte a partir do endereço 6AD7:

```
E1                 POP HL              «Equilibra» a pilha
210668             LD HL,BOARD__2
                   +6                  HL aponta p. o prim.
                                       quad. do tabuleiro.
0604               LD B,04
7E          KING   LD A,(HL)           A: = conteúdo do
                                       quadrado.
FE02               CP 02
2002               JR NZ,M__2          Se existir uma peça
3682               LD (HL),82          simples na última fila
                                       transforme-a em dama.
23          M__2   INC HL              Aponta p. o pr. quadrado.
10F6               DJNZ KING           Repete para todos os
                                       quads. da última fila.
212258             LD HL,ATTRS+22      HL aponta p. o prim.
                                       quad. preto do écran (na
                                       zona dos atributos).
110668             LD DE,BOARD__2
                   +6                  DE aponta p. o prim.
                                       quadrado no tab.-cópia.
0E04               LD C,04
0604        PRL__1 LD B,04
1A          PRL__2 LD A,(DE)           A: = conteúdo do
                                       próximo quadrado.
```

```
13                  INC DE          DE aponta para o
                                    próximo quadrado.
77                  LD (HL),A       Escreve o próximo
                                    quadrado no tabuleiro.

23                  INC HL
23                  INC HL          HL aponta p. o pr.
10F9                DJNZ PRL__2     quadrado.
13                  INC DE          Salta sobre a «margem»
D5                  PUSH DE
111700              LD DE,0017
19                  ADD HL,DE       HL aponta p. o prim.
                                    quadrado da pr. linha.

D1                  POP DE
0604                LD B,04
1A        PRL__3    LD A,(DE)       A: = conteúdo do próximo
                                    quadrado.
13                  INC DE          Aponta p. o pr. quadrado.
77                  LD (HL),A       Escreve o pr. quadrado no
                                    tabuleiro.

23                  INC HL
23                  INC HL          Aponta p. o pr. quadrado.
10F9                DJNZ PRL__3
13                  INC DE
13                  INC DE          Salta sobre dois
                                    marcadores de margem.

D5                  PUSH DE
111900              LD DE,0019
19                  ADD HL,DE       HL aponta p. o prim.
                                    quadrado da pr. linha.

D1                  POP DE
0D                  DEC C
20DC                JR NZ,PRL__1
C9                  RET
```

Na segunda parte das damas veremos o problema de produzir as jogadas do computador e também, é claro, verificar se o jogo acabou ou não. Esta verificação é muito fácil: só precisamos determinar se as peças que restaram no tabuleiro são ou não da mesma cor. De qualquer modo, vamos fazer uma pausa por agora e descansar de todo este esforço mental, dedicando-nos a qualquer coisa mais criativa.

CAPÍTULO TREZE

Um toque de cultura

Será possível obtermos música a partir do *Spectrum?* Todos conhecemos, é claro, a instrução BEEP do BASIC, mas será possível fazer melhor? A resposta é sim.

Como sabemos, BEEP permite tocar uma dada nota durante um certo tempo. Pode-se alterar quer a nota quer a duração «com antecedência», alterando o programa, mas não conseguimos fazer nada em «tempo real» — no próprio instante que está a acontecer. Em código máquina é inteiramente diferente. Em BASIC, ao tocar uma música, uma vez que a rotina de emissão de uma nota comece, o controle manter-se-á nela até que a nota acabe. Então e só então se fará uma nova leitura do teclado.

É possível escrever um programa BASIC que faça BEEP por uns instantes e depois INKEY$ para decidir sobre a nova nota a tocar, mas isto ainda não é controle em tempo real. Em código máquina podemos escrever a nossa própria rotina, para substituir BEEP, fazendo a leitura do teclado enquanto a nota estiver a tocar e agindo de acordo com as nossas intenções.

Chamaremos PROGRAMA DE CATY ao programa seguinte, dedicado a uma pessoa que acredita que os computadores devem ser «artísticos» e não apenas capazes de atacar invasores espaciais. Os endereços dados nesta listagem partem do princípio de que o código máquina será escrito no endereço E800 mas, se quisermos, guardamos o programa em qualquer ponto de RAM que nos apeteça acima de 8000, desde que não esqueçamos de mudar as referências aos endereços absolutos NOTES e SOUND. Lembramos que nos países anglo-saxónicos a notação musical, diferente da nossa, é feita por meio de letras: dó=C, ré=D, mi=E, fá=F, sol=G, lá=A e si=B. O sinal # significa sustenido ou seja meio tom acima da nota[1].

PROGRAMA DE CATY

9E93464B	NOTES	G	G#	G+	F#+	Tecla B	H	Y 6
0050A9B4		—	F+	F#	F	Tecla 5	T	G V
8A813D42		A	A#	A+	G#+	Tecla N	J	U 7
5C5600C1		D#+	E+	—	E	Tecla 4	R	F C
78003539		B	—	B+	A#+	Tecla M	K	I 8
6962CFDD		C#+	D+	D#	D	Tecla 3	E	D X
70003200		C+	—	C++	—	Tecla *sym*	L	Ø 9
0070ECFD		—	C+	C#	C	Tecla 2	W	S Z
00000000		—	—	—	—	Tecla *spc*	*ent*	P Ø
00000000		—	—	—	—	Tecla 1	Q	A *cap*

```
00        SOUND    NOP            Esta sub-rotina provoca
                                  um atraso de duração
                                  determinada.
00                 NOP
00                 NOP
10FB               DJNZ SOUND
D3FE               OUT(FE),A      Produz um impulso (um
                                  elemento de som).
C9                 RET
```

Chama-se a rotina neste ponto:

```
3A485C    START    LD A,(BORDCR)  A: = atributo da
                                  moldura.
1F                 RRA
1F                 RRA
1F                 RRA
E607               AND 07         A: = cor de fundo do
                                  atributo (cor da moldura
                                  actual).
F610               OR 10          Set «bit som» (bit 4).
4F                 LD C,A
F3                 DI
C5        LOOP     PUSH BC
CD8E02             CALL KEY_SCAN  DE: = leitura do teclado.
C1                 POP BC
212027             LD HL,2720     HL: = cód. de leitura de
                                  caps shift space.
```



```
A7                 AND A
ED52               SBC HL,DE
281B               JR Z,EXIT      Termina se caps shift
                                  space for premido.
7B                 LD A,E         A; = leitura do teclado
                                  (ignorando os shifts).
3C                 INC A
28EF               JR Z,LOOP      Teclas não premidas:
                                  salta.
AF                 XOR A
57                 LD D,A         DE: = leitura do teclado
                                  (ignorando os shifts).
2100E8             LD HL,NOTES
19                 ADD HL,DE      HL aponta p. o byte nec.
                                  na tabela NOTES.
46                 LD B,(HL)      B: = byte necessário
                                  para a nota.
B8                 CP B           Compara B com zero.
28E5               JR Z,LOOP      Salta se não houver nota
                                  nessa tecla.
79                 LD A,C         A:=cor de moldura
                                  com o bit quatro set.
C5                 PUSH BC
CD28E8             CALL SOUND     Prim. meio ciclo da nota.
C1                 POP BC
E607               AND 07         Reset o «bit do som»
                                  (bit quatro).
CD28E8             CALL SOUND     Segundo meio ciclo da
                                  nota.
18D8               JR LOOP
FB        EXIT     EI
C9                 RET
```

Agora necessitamos apenas de mais uma instrução para começar a correr o programa. A instrução RANDOMIZE USR 59440, endereço da etiqueta SART (em hex E830). Podemos (e devemos) apagar agora todo o BASIC escrevendo NEW e em seguida, o seguinte programa BASIC.

```
1 RANDOMIZE USR 59440
2 STOP
9999 LOAD "" CODE
```

que pode ser guardado empregando SAVE «MUSIC» LINE 9999 e depois SAVE «MUSIC» CODE 59392,102.

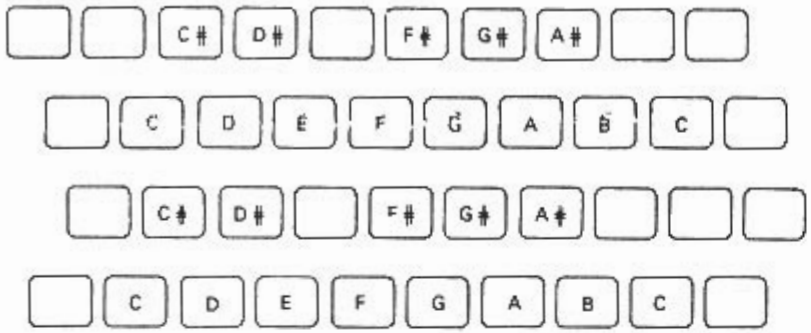


Figura 13.1

Dispomos agora de duas oitavas; o teclado da figura 13.1 mostra onde se encontram as notas. Podemos agora tocar um número razoável de melodias. Se tivermos um desses pequenos adaptadores com uma ficha *jack* de medida adequada ao *Spectrum* de um lado e, do outro, uma que se aplique à nossa aparelhagem estereofónica, o som do *Spectrum* melhorará muito. É indiferente ligar esta ficha ao *Spectrum* em EAR ou em MIC. O manual *Sinclair* diz que se pode fazer isto (para BEEP) mas não salienta muito esse ponto. No entanto, o som do *Spectrum* através de amplificador é bastante bom. Deve-se experimentar.

As teclas individuais podem ser afinadas. A listagem do programa indica também qual o *byte* correspondente a cada nota (no bloco de dados chamado NOTES). Para afinar a nota emitida por uma tecla, basta mudar o valor desse *byte* correspondente. Um zero na tabela significa que a tecla correspondente, quando carregada, só produzirá silêncio. Quanto maior for o *byte*, mais grave será a nota e quanto menor o *byte*, mais aguda será.

Uma vez que o programa esteja a correr, qualquer tecla programada com uma nota irá tocar essa nota até que retiremos o dedo. Mas não se podem tocar duas notas simultaneamente. Note-se, já agora, que a tecla *symbol shift* tem uma nota associada, C (dó).

Nunca se poderia detectar esta nota em BASIC como o fazemos em código máquina. Para interromper o programa, é necessário premir *caps shift* juntamente com *space*.

O modo como o programa realmente funciona é, em princípio, bastante simples. A instrução de código máquina OUT (FE),A (e já agora também a instrução BASIC OUT 254,A) é a mais importante. Os *bits* dois, um e zero do *byte* em A devem conter a cor de BORDER, mas é o *bit* quatro que tem a função mais interessante. Este é o «*bit* de som» e quando se faz um OUT, origina um som. Na verdade, o som só é produzido quando o *bit* quatro muda; por isso, querendo produzir uma nota contínua, é preciso mudar o *bit* quatro de um para zero e de zero para um novamente entre OUTs consecutivos. Da velocidade com que se fizer isto depende a frequência (altura) da nota.

A música no *Spectrum* é um assunto fascinante e é possível escrever um programa que leia informação não do teclado mas de uma comprida *string* de *bytes* contendo o valor e a duração de cada nota de uma melodia. (Estude o leitor isto sozinho, porque a única maneira de aprender realmente é por meio de experimentação.) A instrução DI no programa, que levanta alguma dificuldade de interpretação, destina-se a fazer com que a velocidade da operação (eliminando a interrupção que, de outra forma, ocorreria cinquenta vezes por segundo, alterando assim a duração da execução de SOUND) seja um pouco mais precisa e bastante mais rápida. É útil omiti-la, para ver o que acontece.

Vamos deixar agora a música e voltar a atenção para um assunto um pouco diferente: desenhos...

DESENHOS

Eis outro programa que depende da capacidade artística do operador humano. O problema é que, por causa do espaço que o *Spectrum* emprega para guardar na memória o *display file*, este programa só interessa a quem tenha 48 K.

O programa guarda na memória até cinco imagens de *écran* completas, diferentes, e «circula» através delas, expondo no *écran* uma de cada vez, o tempo que se quiser. Um «desenho» é qualquer imagem que se crie no *écran* — pelo que se podem usar todas as

cores, gráficos de alta resolução, tudo o que nos apetecer. Pode-se utilizar todo o conjunto de caracteres, incluindo os definíveis pelo utilizador PLOT, DRAW e CIRCLE. De facto, qualquer imagem que se consiga criar no *écran* do *Spectrum*.

Agora que estamos preparados, escrevemos o código máquina seguinte a partir do endereço EF05:

A primeira coisa a fazer é reservar memória para guardar estas imagens. Para isto escreve-se CLEAR 26623, que libertará toda a memória do endereço 68ØØ para cima.

```
                ORG EF05
210040  STORE   LD HL,D_FILE     Aponta para o início do
                                 écran.
110068          LD DE,PICTURE1   Aponta p. o end. onde
                                 guardar a primeira
                                 imagem.
01001B          LD BC,1B00       Número de bytes no
                                 écran.
EDB0            LDIR             Copia a imagem.
3A485C          LD A,(BORDCR)
1F              RRA
1F              RRA
1F              RRA              A: = cor BORDER.
12              LD (DE),A        Guarda a cor
C9              RET              BORDER.
```

Agora escrevemos um programa — BASIC ou não — que produz a imagem no *écran*. Pode-se fazer NEW porque o código máquina está acima de (RAMTOP) e portanto não será apagado. Fazer NEW não é, claro está, necessário — pode-se sempre juntar mais BASIC a HEXLD 3 (digamos, à linha 1ØØØ) ou mesmo utilizá-lo para escrever código máquina. A última linha do programa deve ser RANDOMIZE USR 61189.

É útil saber que, fazendo-se POKE 23659,Ø (como instrução de um programa — não como um comando directo) se pode a partir daí escrever em todas as vinte e quatro linhas de *écran*. Até mesmo PRINT AT 23,Ø;"X" funciona! No entanto, se o fizermos, é absolutamente necessário fazer POKE 23659,2 antes de o programa voltar ao modo de comando (por qualquer motivo: ou por haver mais instruções, ou por fazer BREAK, por execução de uma

instrução STOP, ou por ocorrer qualquer erro que origine uma mensagem de erro) ou o sistema fará *crash*.

Depois de fazer tudo isto, muda-se a instrução LD DE,PICTURE1 para LD DE,PICTURE2, faz-se um novo desenho (e RANDOMIZE USR 61189), etc.

Os endereços de que necessitamos são:

PICTURE1	68ØØ
PICTURE2	83Ø1
PICTURE3	9E02
PICTURE4	B9Ø3
PICTURE5	D4Ø4

Vamos escrever RANDOMIZE FN R("FFF6","68ØØ") para fazer a variável BEGIN de HEXLD 3 igual a 68ØØ — o primeiro endereço para SAVE — e depois fazer RUN 4ØØ — o que deve gravar tudo, o que demorará o seu tempo.

Pela primeira vez neste livro vamos utilizar PAUSE. A instrução CALL PAUSE_BC reproduz exactamente a instrução PAUSE BASIC — suspendendo a execução durante BC *frames* (cada um durante 20 ms, havendo portanto 50 *frames* por segundo) ou até que seja premida uma tecla. Fazemos entrar este programa em código máquina a partir do endereço EF18:

```
0605    PICTURES  LD B,number of pic-
                  tures.
210068            LD HL,PICTURE1
C5      NEXT_PIC  PUSH BC
01001B            LD BC,1B00
110040            LD DE,D_FILE     DE aponta para o
                                   começo do écran.
EDB0              LDIR             Copia a pr. imagem.
7E                LD A,(HL)        A: = cor BORDER.
D3FE              OUT (FE),A       Muda a cor BORDER.
23                INC HL           Aponta p. a pr. imagem.
E5                PUSH HL
016400            LD BC, length of
                  pause.
CD3D1F            CALL PAUSE_BC    Pausa durante o nº
                                   requerido de frames.
```

```
E1              POP HL
C1              POP BC
10E8            DJNZ NEXT_PIC
C9              RET
```

Decerto que o leitor reparou na nova instrução OUT (FE),A. Esta instrução irá mudar a cor da moldura para o valor de A. Se A for maior que sete, então só os três *bits* mais à direita serão utilizados. Já agora, a instrução OUT (C),r fará o mesmo desde que C contenha FE. Para mudar a cor da moldura permanentemente para o valor contido em A, é necessário empregar as seguintes instruções:

```
D3FE            OUT(FE),A       Muda imediatamente a
                                cor de BORDER.
17              RLA
17              RLA
17              RLA             Multiplica por oito.
E638            AND 38          Ignora os bits 7 e 6 e
                                define cor preta.
CB6F            BIT 5,A         Esta cor tem uma
                                componente verde?
2002            JR Z,N_BORD     Nesse caso, salta.
F607            OR 07           Define cor branca.
32485C  N_BORD  LD (BORDCR),A   Guarda a nova cor
                                BORDER.
```

LIFE

No último programa deste capítulo as coisas alteram-se um pouco. Nós, humanos, tivemos o monopólio da arte durante muito tempo — chegou a vez dos computadores...

Este programa chama-se LIFE — representa um ciclo de vida, nascimento/crescimento/morte, de uma colónia de células vivendo numa grelha quadrada. Produz resultados fascinantes. Perante os nossos olhos aparece um padrão em constante evolução — começando ao acaso e acabando por vezes com a morte da colónia de células, por vezes com uma estrutura celular fixa e imóvel que atingiu o seu equilíbrio e ainda, por vezes, com uma sequência

continuamente repetida, chamada de equilíbrio dinâmico. É uma coisa espantosa de se ver.

LIFE foi inventado em 1970 por John Conway, da Universidade de Cambridge, e é surpreendente que a Tate Gallery ainda não tenha adquirido uma cópia dele. Apesar de tratar, na realidade, do crescimento de células que seguem regras matemáticas rígidas, torna-se, na prática, um algoritmo gerador de imagens bem conseguido.

O princípio de LIFE é muito simples. Uma grelha — geralmente quadrada — está coberta com células em cerca de um quarto dos seus quadrados. Neste programa essas posições são escolhidas completamente ao acaso. Esta configuração da grelha chama-se «geração zero».

Gerações sucessivas são construídas com base num princípio facilmente compreensível. Cada quadrado da grelha está cercado por oito quadrados. Cada um destes quadrados circundantes ou contém outra célula ou está vazio. Cada célula que tem precisamente ou duas ou três células vizinhas atingirá a próxima geração, todas as outras irão morrer. Uma nova célula, no entanto, irá nascer em qualquer espaço vazio que tenha exactamente três células vizinhas. Com estas regras bastante simples, é surpreendente que o programa produza resultados tão elegantes.

Nesta versão de LIFE, a nossa grelha tem dezasseis quadrados por dezasseis porque, é claro, dezasseis é um número muito fácil para trabalhar em hexadecimal. Mais ainda, a nossa grelha está construída, estranhamente, num espaço curvo contínuo, de modo que todos os quadrados do bordo esquerdo estão ligados aos do bordo direito e vice-versa e também todos os quadrados no topo superior estão em contacto com os correspondentes quadrados no topo inferior e vice-versa.

O BASIC auxiliar é:

```
10 RAND USR START
20 RAND USR NEXT_GEN
30 GO TO 20
9999 LOAD "LIFE" CODE
```

Que deve ser guardado utilizando SAVE "LIFE" LINE 9999. START e NEXT_GEN são endereços no programa em código

máquina. Nesta listagem partimos do princípio de que o primeiro endereço utilizado é 6800 e que o endereço da etiqueta START é 6908 hex. Podemos facilmente alterá-los a nosso gosto.

```
                    ORG 6800
          DUMP      DEFS 100h       Um espaço de trabalho
                                    usado p. calcular a pr.
                                    geração.
EF010110  TABLE     DEFB EF 01 01 10 Bytes representando
                                    as deslocações para os
                                    quadrados vizinhos.
10F FFFF0           DEFB 10 FF FF F0
AF        START     XOR A
323C5C              LD (TVFLAG),A   Selecciona PRINT no
                                    cimo do écran
328F5C              LD (ATTR_T),A   Temporariamente, faz
                                    PAPER e INK ambos
                                    pretos.
0E10                LD C,10
0610      BLACK_1   LD B,10
3E4F      BLACK_2   LD A,"O"
D7                  RST 10          Escreve a pr. célula.
10FB                DJNZ BLACK_2
3E0D                LD A,"enter"
D7                  RST 10          Fim da fila.
0D                  DEC C
20F3                JR NZ,BLACK_1
210058              LD HL,ATTRS     HL aponta para a zona
                                    dos atributos.
0E10                LD C,10
0610      CELL_1    LD B,10
E5        CELL_2    PUSH HL
2A765C              LD HL,(SEED)    HL: = variável de sist.
                                    SEED (dos n.º aleatórios).
54                  LD D,H
5D                  LD E,L          DE: = variável de sist.
                                    SEED (dos n.º aleatórios).
29                  ADD HL,HL
29                  ADD HL,HL
19                  ADD HL,DE
```



```
29                  ADD HL,HL
29                  ADD HL,HL
29                  ADD HL,HL
19                  ADD HL,DE       HL: = novo número
                                    aleatório.
22765C              LD (SEED),HL
7C                  LD A,H
FEC4                CP C4
3804                JR C,BLACK      Decide se coloca ou
                                    não a célula.
3E07                LD A,07
1801                JR C_1
AF        BLACK     XOR A
E1        C_1       POP HL          HL: = aponta o byte
                                    de atributos da célula.
77                  LD (HL),A       Escreve célula se existe.
23                  INC HL          Aponta a pr. posição
                                    na zona dos atributos.
10E1                DJNZ CELL_2
111000              LD DE,0010
19                  ADD HL, DE      Aponta a prim. célula
                                    da linha seguinte.
0D                  DEC C
20D8                JR NZ,CELL_1
C9                  RET
210058    NEXT_GEN  LD HL,ATTRS     HL aponta para a zona
                                    dos atributos.
110068              LD DE,DUMP      DE aponta p. a área útil.
E5                  PUSH HL         Empilha a constante
                                    5800.
D5                  PUSH DE         Empilha o endereço de
                                    DUMP.
011000              LD BC,0010      Número de quadrados
                                    em cada fila.
79                  LD A,C          Número de filas.
C5        COPY_G    PUSH BC
EDB0                LDIR            Copia uma fila para a
                                    zona de trabalho.
C1                  POP BC          BC: = 0010.
09                  ADD HL,BC       HL aponta a pr. fila da
                                    zona dos atributos.
```

```
3D                      DEC A
20F8                    JR NZ,COPY__G     Copia toda a grelha na
                                          área de trabalho.
E1                      POP HL            HL: = endr. de salto.
D1                      POP DE            DE: = 5800.
D5                      PUSH DE
E5                      PUSH HL
0E00        NXT__CELL   LD C,00           C conta o número de
                                          células vizinhas.
110069                  LD DE, TABLE      DE: = aponta a tabela
                                          de deslocamentos.
0608                    LD B,08
1A          NXT__DIS    LD A,(DE)         A: = próximo
                                          deslocamento.
13                      INC DE            Aponta para a pr.
                                          posição na tabela.
85                      ADD A,L
6F                      LD L,A            HL: = aponta o pr.
                                          quadrado vizinho.
7E                      LD A,(HL)         A: = conteúdo deste
                                          quadrado.
A7                      AND A             Este quadrado tem
                                          uma célula?
2801                    JR Z,COUNTED
0C                      INC C             Então, incrementa o cont.
10F5        COUNTED     DJNZ NXT__DIS     Repete para todos os
                                          vizinhos.
E1                      POP HL            HL: = aponta p. o
                                          quad. corrente em DUMP.
79                      LD A,C            A: = número de
                                          células vizinhas.
FE02                    CP 02
380F                    JR C,NO__CELL     Célula morre com uma
                                          ou nenhuma vizinha.
FE04                    CP 04
300B                    JR NC,NO__CELL    Célula morre com quatro
                                          ou mais vizinhas.
FE03                    CP 03
2803                    JR Z,CELL         Com três vizinhas, uma
                                          nova célula é criada.
7E                      LD A,(HL)         A: = conteúdo
                                          presente do quad.
```



```
1805                    JR PUT
3E07        CELL        LD A,07           07 significa «preto em
                                          branco» como atributo.
1801                    JR PUT
AF          NO__CELL    XOR A             00 significa «preto em
                                          preto» como atributo.
E3          PUT         EX (SP),HL        HL aponta para a zona
                                          dos atributos.
77                      LD (HL),A         Escreve conteúdo do quad.
23                      INC HL            Aponta o pr. quadrado.
7D                      LD A,L
E61F                    AND 1F
FE10                    CP 10
2004                    JR NZ,ROW
111000                  LD DE,0010
19                      ADD HL,DE         HL aponta p. o início
                                          da pr. fila se necessário.
E3          ROW         EX (SP),HL        HL aponta p. DUMP.
23                      INC HL            Aponta p. o pr. quadrado.
E5                      PUSH HL
7D                      LD A,L
A7                      AND A
20C3                    JR NZ,NXT__CELL   Repete até considerar
                                          todas as células.
E1                      POP HL
E1                      POP HL            «Equilibra» a pilha.
C9                      RET               Retorna ao BASIC.
```

Se foram utilizados os endereços da listagem, START é 26888 e NEXT__GEN é 26956.

A primeira coisa a fazer é escrever RANDOMIZE para colocar um número aleatório na variável de sistema SEED, de que o programa irá necessitar. Pode-se agora fazer RUN.

Um ponto interessante neste programa é que ele é capaz de produzir os seus próprios números aleatórios. A parte a seguir à etiqueta CELL__2 encarrega-se disso. Deve-se estudar a forma como isto se faz e, querendo, utilizaremos o mesmo princípio nos nossos programas.

LIFE produz uma geração zero construída ao acaso quase instantaneamente, e as sucessivas gerações à espantosa velocidade

de quinze por segundo! Pode-se abrandar esta velocidade juntando uma instrução PAUSE do BASIC. Uma outra maneira é juntar uma linha dentro do ciclo — por exemplo, LET X=Ø/:LET X= X + 1:/ PRINT AT 17,Ø "Geração";X — que tem a vantagem adicional de nos indicar o número de gerações já apresentadas.

Por fim, deve-se notar o modo pelo qual este programa, contrariamente a outros programas LIFE, calcula cada nova geração inteiramente baseada na anterior. Não calcula a primeira fila e depois a segunda com base na primeira fila já alterada. A segunda fila é determinada inteiramente tendo em conta a primeira fila anterior à alteração. É para guardar a geração anterior que serve o bloco de memória chamado DUMP na listagem em código máquina. Cada nova geração é então correctamente construída.

Existem muitos outros programas geradores de imagens, alguns muito mais simples, mas nenhum com a elegância matemática de LIFE. Um bom desafio para os novos programadores é escrever uma versão de LIFE com vinte e quatro por vinte e quatro quadrados ou mesmo com vinte e quatro por trinta e dois quadrados.

Pode-se utilizar RST 1Ø nas duas linhas da parte inferior do *écran* desde que se carregue a variável de sistema DF_SZ com zero, no início e com dois logo que a impressão esteja feita.

A maior grelha de células que se pode conseguir no *Spectrum* é de 256d por 192d, empregando *pixels* brancos para células e *pixels* pretos para espaços vazios, mas isso seria um programa bastante complicado. Quem se sentir realmente entusiasmado, pode tentar esta obra monumental. A decisão depende de cada um.

CAPÍTULO CATORZE

Damas – segunda parte

Este capítulo trata da rotina que escolhe qual o movimento que o computador deve fazer, depois da nossa jogada.

Deve-se seguir muito cuidadosamente o que vamos dizer em seguida. É um resumo sistemático do modo como essa jogada é escolhida.

Inspeccionamos o tabuleiro, um quadrado preto de cada vez, e sempre que encontramos uma peça do computador devemos meditar. Para cada movimento possível, atribuímos um valor numérico que é tanto maior quanto melhor esse movimento nos parece em relação aos outros. Segue-se então que para seleccionar um movimento, só precisamos escolher aquele ao qual tivermos atribuído maior valor.

É claro que esta ideia não permitirá que o computador planeie com avanço, pois só pensa uma jogada de cada vez. Para construir esta lista de jogadas e correspondentes valores numéricos não necessitamos de guardar cada um dos movimentos. Uma vez localizada uma jogada possível e calculada a sua pontuação, acontece o seguinte:

Se a pontuação for inferior às listas, é ignorada.

Se a pontuação for igual às da lista, a jogada é adicionada à lista.

Se a pontuação for mais alta que as da lista, então a lista é abolida, sendo inciada uma nova, com a jogada encontrada.

Deste modo a lista é sempre muito pequena. Quando se chega à decisão final, o computador apenas tem de escolher um destes movimentos, que ficaram na lista, ao acaso. A próxima questão é — onde é guardada esta lista?

A resposta é: na pilha. Isto simplifica as coisas mas obriga a que seja necessário manter um registo do sítio onde começa a lista. Vamos guardá-la no endereço 5CBØ (a variável de sistema SPARE2 — dois *bytes* não utilizados). A esta quantidade vamos chamar L_BASE (a base da lista). É sempre perfeitamente seguro guardar coisas nesta variável de sistema já que, como o manual diz, ela não é utilizada pela ROM.

A parte destinada a tomar as decisões sobre a jogada do computador começa no endereço 6AE7. A primeira instrução é LD (L__BASE),SP. O começo da lista fica assim preservado. Podemos agora fazer com a pilha o que quisermos, desde que nos lembremos de restaurar o seu valor antes de retornar a BASIC. A segunda e a terceira instruções "empilham" o número zero na nossa lista, o que irá indicar ao programa que ela se encontra, de momento, vazia.

O ciclo de verificação é como se segue. Observem-se as duas novas variáveis empregadas: L__COUNT (que guarda o número de entradas na lista) e P__COUNT (que guarda o número de peças que foram analisadas). Ambas são guardadas na zona de memória correspondente à variável de sistema MEMBOT, geralmente as memória do calculador de vírgula flutuante do *Spectrum*. Mais uma vez, um bom sítio para guardar números, perfeitamente seguro para utilizar em qualquer programa de código máquina que não exija o uso de números em vírgula flutuante (nem a impressão de caracteres gráficos, que usa mem0 e mem1 — ver Capítulo Dezoito). A rotina, e de facto todas as rotinas subsequentes apresentadas neste capítulo, é para ser *inserida*, porque a rotina que imprime o tabuleiro deve ser a última. O endereço 6AE7 no qual esta rotina deve ser inserida coloca-a, de facto, imediatamente antes da impressão do tabuleiro.

ED73 B05C	BD__SCAN.	LD (L__BASE),SP	Guarda o início da lista.
210000		LD HL,0000.	
E5		PUSH HL	Inicializa a lista.
22925C		LD (P__COUNT), HL	Inicializa o cont. do n.º de peças e o cont. do n.º de entradas na lista.
210668		LD HL,BOARD __2+6	HL aponta p. o prim. quad. preto do tabuleiro.
7E	NXT	LD A,(HL)__CHK.	A: = conteúdo do teclado.
E67F		AND 7F	Ignora o *status* da dama.
FE07		CP 07	Encontrada uma peça do computador?
CC5200		CALL Z,EVALUATE	(Linha temporária.)
23		INC HL	Aponta p. o pr. quadrado.
7D		LD A,L.	
FE30		CP 30.	

20F2	JR NZ,NXT__CHK	Repete para todos os quadrados do tabuleiro.
ED7BB 05C	LD SP,(L__BASE)	«Esvazia» a lista para evitar *crash*.

Como se vê, esta parte é muito clara, mas chama uma sub-rotina que ainda não escrevemos — EVALUATE — que tem a função de calcular todos os movimentos possíveis a partir desse quadro. Como ainda não a escrevemos, modificamos o hex dessa instrução para CALL Z,0052, endereço da ROM no qual se encontra uma instrução RET — fazendo assim com que a instrução seja temporariamente sem acção, servindo apenas para nos lembrar o que se deveria passar neste ponto. A instrução final encontra-se provisoriamente no programa, evitando o *crash*. Não é difícil compreender como, carregando SP com (L__BASE), se elimina a necessidade de fazer POP de tudo o que se encontra na pilha antes de retornar. Carregando SP leva-se a máquina a «pensar» que a pilha não foi alterada desde a primeira instrução da rotina.

Repare-se também que a instrução LD (P__COUNT),HL carregará tanto P__COUNT como L__COUNT, já que são variáveis de um *byte* e HL contém dois *bytes*. Isto, claro, só funciona quando L__COUNT e P__COUNT estão guardadas em endereços consecutivos.

Precisamos pensar agora qual a forma que desejamos que a lista tome. Vejamos o problema em sentido inverso. Qual a forma que gostaríamos que a lista tivesse para podermos examinar a pilha com mais facilidade?

O primeiro item da pilha deve ser o número de passos da jogada — isto é, um para um só movimento/salto, dois para um salto duplo, três para um salto triplo e assim por diante. O segundo item deve ser o valor numérico que foi atribuído aos items da pilha — a prioridade, como vamos chamar-lhe. Seguindo estes elementos de informação deve vir a própria lista, começando no quadrado a partir do qual nos vamos deslocar, seguido de uma sequência de uma ou mais direcções nas quais o movimento correspondente deverá ser efectuado. Imediatamente a seguir, o segundo item da lista, da mesma forma, depois o terceiro e assim por diante...

Vê-se que cada elemento de que necessitarmos na pilha só precisará de ocupar um *byte*. O número de passos não pode ser

maior que FF. Escolheremos a prioridade como quisermos — podemos sempre torná-la de um só *byte*, por exemplo. O quadrado inicial pode ser guardado «empilhando» apenas a parte baixa do seu endereço em BOARD__2. As direcções em que podemos mover guardam-se do mesmo modo que anteriormente — 05, 06, FA ou FB para mais ou menos cinco ou seis. Para o programa adquirir o máximo de eficiência, justifica-se que usemos uma forma tão condensada de guardar informação. Imaginemos que a lista está agora completa e vamos seleccionar um item daí. A pilha apresenta-se como no diagrama, figura 14.1(A). Se empregarmos a instrução POP BC, B conterá a prioridade e C o número de passos. A prioridade é agora uma informação desnecessária, já que só precisámos dela para construir a lista. C, contudo, é muito importante. No caso do diagrama da figura 14.1(A), C seria um, mas noutros casos pode não acontecer isso. A pilha apresentar-se-á, neste caso, como na figura 14.2(B).

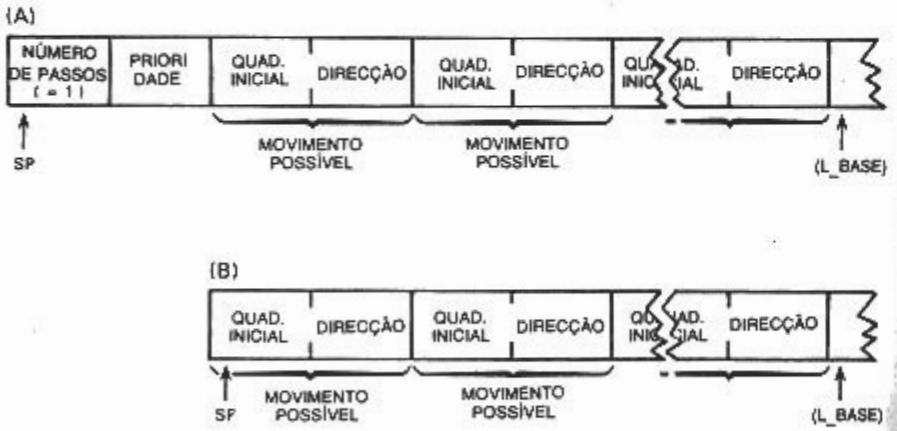


Figura 14.1

Se, em vez disto, existissem dois passos para cada movimento da lista e não um, a pilha seria como na figura 14.2.

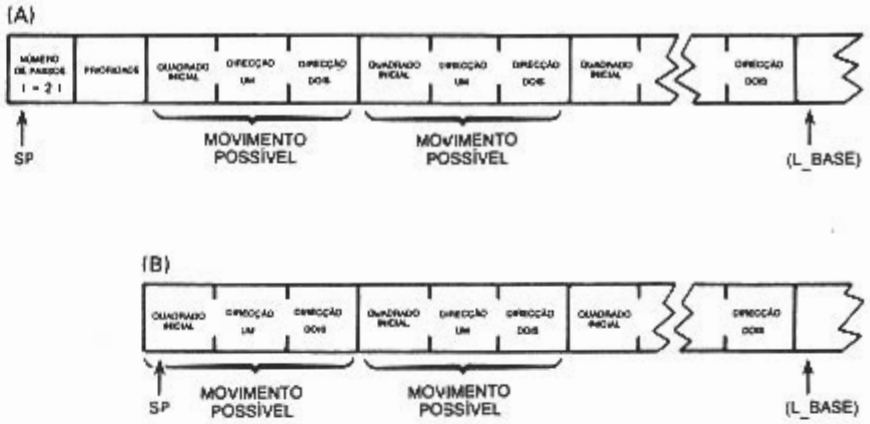


Figura 14.2

Necessitamos nesse caso de remover um número, ao acaso, de items da pilha, de modo que o movimento que finalmente escolhermos se encontre no topo desta. A rotina seguinte encarregar-se-á disso mesmo e deve ser inserida no endereço 6BØ3.

```
C1          CHOOSE  POP BC          C: = n.º de passos numa
                                    seq. de movimentos.
3A935C              LD A,(L__COUNT) A: = número de escolhas
                                    possíveis.
A7                  AND A
2812                JR Z,CHOICE     Salta se não houver
                                    movimentos a escolher.
5F                  LD E,A
3A785C              LD A,(FRAMES)   A: = números aleatórios
                        low         entre ØØ e FF.
93          RANDOM  SUB  E
3ØFD                JR NC,RANDOM
83                  ADD A,E         A: = número aleatório
                                    entre ØØ e E-1.
2808                JR Z,CHOICE     Se zero não faz nada.
41          NSQ__OFF LD B,C         B: = n.º de passos na
                                    sequência de movimentos.
33                  INC SP          Apaga o «quadrado
                                    inicial» da pilha.
```

```
33          NDR__OFF    INC SP              Apaga «próxima
                                            direcção» da pilha.
10FD                    DJNZ NDR__OFF       Apaga toda a seq. de
                                            movimentos.
3D                      DEC A               .
20F8                    JR NZ,NSQ__OFF      Apaga «A» itens.
            CHOICE      $.
```

O movimento seleccionado encontra-se agora no topo da pilha. Vamos ver como ela se encontra agora. A figura 14.3 mostra-nos isso.

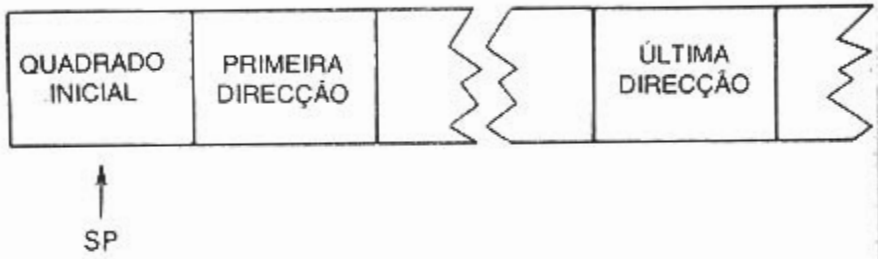


Figura 14.3

A sequência para encontrar o quadrado inicial é DEC SP/POP AF/LD L,A/LD H,68. Como se vê, o «quadrado inicial» é a parte baixa do endereço, sendo 68 a parte alta. É muito importante que o ponteiro da pilha (SP) nunca aponte para nenhum endereço superior ao do último item da pilha. No diagrama da figura 14.3 isto significa que SP não pode ir mais para a direita da imagem que o item do topo (isto é, onde se encontra, no diagrama). Podemos deslocá-lo para a esquerda (com DEC SP ou PUSH) sem perigo, mas não para a direita — nem mesmo temporariamente — excepto se empregarmos a instrução POP. Isto passa-se assim porque sempre que o processador recebe uma interrupção (o que acontece 50 vezes por segundo) a pilha é utilizada para guardar o endereço de retorno e os registos (além de ser utilizada pela própria rotina de interrupção). No retorno da interrupção embora o SP e os registos sejam restaurados nos mesmos valores que anteriormente, os *bytes*

na pilha imediatamente abaixo de SP estarão modificados, sendo que o que se encontra à esquerda da posição imediata de SP será assim apagado. Uma interrupção pode acontecer entre duas quaisquer instruções. É claro que nada nos impede de fazer um pouco de batota, empregando DI (e de facto, se fizéssemos isso, o nosso programa correria um pouco mais depressa) mas isto não é considerado como uma boa técnica de programação. Não pergunte porquê, porque não sabemos![1]

De qualquer modo: é melhor seguirmos a opinião profissional e sermos cuidadosos com o ponteiro da pilha.

DEC SP/POP AF carrega o *byte* no topo da pilha no registo A sem pôr a lista em perigo. Agora, é necessário apenas que o movimento seja escrito e executado. Insere-se este novo fragmento a partir do endereço 6B1C. Note-se que desta vez não existem verificações sobre o cumprimento das regras porque partimos do princípio de que o computador não pode fazer batota.

```
CD3768              CALL S__PRINT
16030B              DEFM AT 3,11d
4555204A4F474F20    DEFM EU JOGO espaço
00                  DEFB 00
3B                  DEC SP
F1                  POP AF          A: = quadrado inicial.
F5                  PUSH AF
87                  ADD A,A         Multiplica por dois para
                                    tomar em conta os
                                    quadrados brancos.
3C                  INC A           Toma em conta que o
                                    primeiro quadrado
                                    preto é 12 e não 11.
11FF0B              LD DE,0BFF      D: = «largura» do
                                    tabuleiro BOARD_2
                                    (veja a figura 12.2).
                                    E: = −1.
1C          MOD     INC E           E conta as filas.
92                  SUB D
30FC                JR NC,MOD       E: = primeira coord. do
                                    quadrado.
```

```
82                  ADD A,D          A: = segunda coord.
                                     do quadrado dado.
F5                  PUSH AF
7B                  LD A,E
C630                ADD A,30         Converte em carácter
                                     ASCII.
D7                  RST 10           Escreve a primeira coord.
F1                  POP AF           A: = segunda coord.
C630                ADD A,30         Converte em carácter
                                     ASCII.
D7                  RST 10           Escreve o segundo díg.
F1                  POP AF           A: = número do
                                     quadrado (figura 12.2).
6F                  LD L,A
2668                LD H,68          HL: = aponta o
                                     quadrado inicial em
                                     BOARD_2.
41                  LD B,C           B: = número de
                                     passos do movimento.
4E      NXT__STEP   LD C,(HL)        C: = peça do comp.
3600                LD (HL),00       Apaga peça do quad.
3B                  DEC SP.
F1                  POP AF           A: = próxima direcção:
                                     FA, FB, 06 ou 05.
F5                  PUSH AF          FA para direcção «A».
1647                LD D,47          FB p. direcção «B», etc.
CB7F                BIT 7,A.
2004                JR NZ,DIR__CHR   Salta se a direcção for
                                     FA ou FB.
1649                LD D,49.
ED44                NEG              A: = FA para direcção
                                     «C», FB para direcção
                                     «D».
82      DIR__CHR    ADD A,D          A: = 41, 42, 43 ou 44.
D7                  RST 10           Escreve a direcção.
F1                  POP AF           A: = direcção FA, FB,
                                     06 ou 05.
57                  LD D,A.
85                  ADD A,L.
6F                  LD L,A           HL: = quad. de destino.
7E                  LD A,(HL)        A: = conteúdo do quad.
A7                  AND A.
2805                JR Z,SQUARE      É um mov. ou um salto?
```



```
3600                LD (HL),00       Apaga a peça «tomada».
7D                  LD A,L.
82                  ADD A,D.
6F                  LD L,A           HL: = novo quadrado
                                     de destino.
71      SQUARE      LD (HL),C        Coloca a peça do
                                     comp. na nova posição.
10DE                DJNZ NXT__STEP
        M__DONE     $.
```

Agora faremos o que quisermos menos tentar correr deste já o programa, porque ele tem agora um pequeno *bug* (erro) (sobre o significado exacto da palavra *«bug»*, ter em conta a seguinte citação: *We call them bugs, because to call them mistakes would be psychologically unacceptable*) que é necessário corrigir. Na secção com a etiqueta CHOOSE (a penúltima) o controle salta para a etiqueta CHOICE se (L__ COUNT) for zero. (Escrevemo-lo assim porque quando o introduzimos o seu verdadeiro destino ainda não existia.) Em vez disso devemos fazer dar um salto à etiqueta M__DONE da última secção. Vamos introduzir o seguinte para remediar este facto:

```
RUN 100/6B09/5E
```

Como as coisas se encontram neste momento, não podemos tentar esta rotina, ou qualquer outra neste capítulo que venha aqui ter, porque a sub-rotina EVALUATE não existe. Sugerimos que se escreva uma «falsa» sub-rotina EVALUATE que, se quisermos realizar qualquer outra acção, se limite a construir na pilha uma lista com o formato conveniente, com valores prefixados por nós. Coloca-se esta sub-rotina «a fingir» a partir do endereço 7CBA (16 K) ou FCBA (48 K) empregando WRITE (RUN 100). É um lugar apropriado: é na realidade o endereço do gráfico «definido pelo utilizador» B. (Lembremo-nos que HEXLD 3 guarda também os caracteres gráficos «definidos pelo utilizador».) O gráfico A já foi usado (uma peça de damas) mas a partir de «B» está tudo livre.

```
C1      EVALUATE    POP BC           BC: = endereço de
        __MOCK                       retorno da sub-rotina.
111106              LD DE,0611
D5                  PUSH DE          Empilha o mov. 32C.
```

```
15              DEC D
D5              PUSH DE          Empilha o mov. 32D.
1C              INC E
D5              PUSH DE          Empilha o mov. 34D.
14              INC D
D5              PUSH DE          Empilha o mov. 34C.
1C              INC E
D5              PUSH DE          Empilha o mov. 36C.
15              DEC D
D5              PUSH DE          Empilha o mov. 36D.
1C              INC E
D5              PUSH DE          Empilha o mov. 38D.
110101          LD DE,0101
D5              PUSH DE          Empilha a prioridade e o
                                 número de passos por
                                 movimento.
110C07          LD DE,070C
ED53925C        LD (P__COUNT),DE (P_COUNT) = 0D (doze
                                 peças no tabuleiro)
                                 (L_COUNT) = 07 (sete
                                 movimentos à escolha).

C5              PUSH BC          «Empilhar» o endr. de
C9              RET              retorno.
```

Agora vamos escrever RANDOMIZE FN R ("FFFE"I,"6BA8") (48 K) ou RANDOMIZE FN R ("7FFE","6BA8") (16 K) para restabelecer o valor de LIMIT.

Fazendo RUN 100/6AFB/BA7C, muda-se o endereço na instrução CALL Z, EVALUATE; deste modo vemos o programa a funcionar em toda a sua glória.

Note-se que só a primeira jogada do computador está agora programada correctamente, de modo que só se pode fazer uma única jogada. A verdadeira sub-rotina EVALUATE é algo que iremos ver na terceira parte de damas. Por enquanto, temos ainda de nos preocupar com a verificação de fim de jogo.

A rotina seguinte é dada sem mais explicações. Mas devemos compreender como funciona. Vamos escrevê-la a partir do endereço 6BA7 (no fim do programa):


```
3A925C GAME-OVER LD A,(P__COUNT)  A: = número de peças
A7               AND A            do computador.
2036             JR NZ,GO__2
CD3768           CALL S__PRINT
16020B           DEFM AT 2,11d
1107             DEFM fundo branco
1201             DEFM «flash» activo
204D5549544F     DEFM espaço MUITOS PARABÉNS espaço—
53205041524 1    espaço
42454E53202D20
16030B           DEFM AT 3,11d
47414E484F5520   DEFM GANHOU
4F20             DEFM O
4A4F474F212121
202020           DEFM JOGO!!! três espaços
00               DEFB 00
CF               RST 08
FF               DEFB FF          Retorno ao modo
                                  de comando BASIC.

210668   GO__2   LD HL,BOARD__2
                 +6               HL aponta o prim. quad.
062A             LD B,2A
7E       GO__LOOP LD, A,(HL)       A: = conteúdo do quad.
23               INC HL           Aponta o pr. quadrado.
E67F             AND 7F           Ignora o status da dama.
FE02             CP 02            É uma peça humana?
C8               RET Z            Se sim, retorna ao BASIC.
10F7             DJNZ GO__LOOP
CD3768           CALL S__PRINT
16020B           DEFM AT 2,11d
1107             DEFM fundo branco
1201             DEFM «flash» activo
20415A41522E2E2E DEFM espaço AZAR...
2020             DEFM dois espaços
2D20             DEFM espaço
16030B           DEFMAT 3,11d
56454E4349       DEFM VENCI
204F20           DEFM espaço O espaço
4A4F474F2120     DEFM JOGO! espaço
00               DEFB 00
CF               RST 08
FF               DEFB FF          Retorna ao modo
                                  de comando BASIC.
```

Acabámos a primeira parte com uma rotina que verificava se alguma das nossas peças se tornava dama. Vamos terminar esta parte com uma rotina que faz o mesmo para as peças do computador. A rotina deve ser inserida imediatamente antes da rotina que escreve o tabuleiro actualizado — no endereço 6B6C:

```
212C68          LD HL,BOARD__2+2C  Aponta p. o prim.
                                   quadrado da última fila.
0604            LD B,04
7E      W__KING LD A,(HL)          A: = conteúdo do quad.
FE07            CP 07              É uma peça do
                                   computador?
2002            JR NZ,WK__2
3687            LD (HL),87         Muda a peça p. dama.
23      WK__2   INC HL
10F6            DJNZ W__KING
```

No próximo capítulo vamos ver alguns jogos completos (e curtos) escritos com a intenção de demonstrar o que o código máquina pode fazer em termos de velocidade e, comparativamente ao BASIC, em muito menos *bytes*.

CAPÍTULO QUINZE

Jogos gráficos

ESPIRAIS

Neste jogo gráfico rápido e em tempo real, somos colocados no princípio de uma espiral quadrada e temos de chegar ao fim dela num mínimo de tempo. A pontuação está constantemente à vista — começa em 99900 e decresce continuamente, mas é impossível fazer batota, interrompendo quando se tem uma alta pontuação, porque o programa não o permite. De vez em quando, a pontuação atingirá zero antes de chegarmos ao fim da espiral. Se isso acontecer, é porque, obviamente, precisamos praticar mais.

É fascinante observar este jogo — ver a pontuação diminuir tem um efeito surpreendente. Podemos tornar o jogo tão difícil quanto quisermos, bastando para isso alterar o valor inicial da temporização, contido em BC. Neste caso demos-lhe o valor de 0800, mas podemos usar 1000 para um jogo mais lento, 0400 para um jogo mais rápido e assim por diante.

Existe no entanto uma dificuldade — se batermos numa parede não saltamos para trás, na realidade ficamos presos a ela, e a única maneira de sair dessa situação é inverter o sentido. Isto pode tornar-se bastante complicado.

Bem, leitor, boa sorte nesta corrida; mantenha um registo de pontuações altas (sem batota) e veja se consegue dominar o jogo.

As teclas movem-no do seguinte modo: «I» para cima; «J» para a esquerda, «K» para a direita e «M» para baixo. Estas teclas formam uma espécie de cruz no teclado, de modo que é muito fácil lembrar que tecla move para que lado. É preferível utilizar estas teclas como teclas de direcção em vez das teclas de cursor «5», «6», «7» e «8» pela razão de que é por vezes muito difícil lembrar-se qual a tecla para cima e qual a para baixo, se o «6» se o «7».

Segue-se a listagem do programa, que está escrita supondo-o guardado a partir do endereço 6800. Pode, no entanto, ser recolocado em qualquer outro endereço, desde que todos os endereços

absolutos referentes ao programa que nele aparecem sejam também alterados. O programa deve ser chamado a partir da etiqueta START.

```
E0FF    SPIRAL   DEFW 1111 1111 1110 0000 b   Estes dados
20A0             DEFW 1010 0000 0010 0000 b   representam o
A0AF             DEFW 1010 1111 1010 0000 b   formato da
A0A8             DEFW 1010 1000 1010 0000     espiral
A0AA             DEFW 1010 1010 1010 0000 b   quadrada, com
A0AE             DEFW 1010 1110 1010 0000 b   os uns
A0A0             DEFW 1010 0000 1010 0000 b   representando
A0BF             DEFW 1011 1111 1010 0000 b   paredes e os
2080             DEFW 1000 0000 0010 0000 b   zeros, espaços.
E0FF             DEFW 1111 1111 1110 0000 b
0D      SCORE_I  DEFM enter                   Estes dados
4F5320           DEFM OS                      representam a
5345555320       DEFM SEUS                    mensagem a
504F4E544F53     DEFM PONTOS                  ser escrita com
3939393030       DEFM 99900                   a pontuação no
                                              momento.

160B0F SCORE_C   DEFM AT 11d,15d;              Estes bytes
303030           DEFM 000                     serão usados p.
                                              dar a pontuação
                                              no momento.
```

Chame aqui:

```
AF       START   XOR A
323C5C           LD (TVFLAG),A        Escreve na parte
                                      superior do écran.
060A             LD B,0A
210068           LD HL,SPIRAL         Aponta p. dados da
5E       S_LOOP  LD E,(HL)            espiral.
23               INC HL
56               LD D,(HL)            DE: = próxima word de
                                      dados.
23               INC HL
EB               EX DE,HL             HL: = próxima word de
                                      dados.
29       P_LOOP  ADD HL,HL
3806             JR C,WALL
FD36553F         LD (ATTR_T),3F       Cores escolhidas:
                                      branco em branco.
1804             JR PR_SQ
FD365500 WALL    LD (ATTR_T),00       Selecciona preto em
                                      preto.
3E2B     PR_SQ   LD A,"A+"
D7               RST 10               Escreve.
7C               LD A,H
B5               OR L
20EC             JR NZ,R_LOOP         Escreve fila completa.
3E0D             LD A,"enter"
D7               RST 10               Prepara p. a pr. fila.
EB               EX DE,HL             HL: = aponta para a pr.
                                      word de dados.
10E1             DJNZ S_LOOP          Escreve estr. completa.
3E38             LD A,38              A é a selecção de cores
                                      preto em branco.
322158           LD (ATTRS+21),A      Escreve "+" na posição
                                      de partida.
FD365530         LD (ATTR_T),30       Atributos para preto em
                                      amarelo.
211468           LD HL,SCORE_I        HL aponta p. os bytes
                                      da pontuação inicial.
0615             LD B,15
7E       SC_LOOP LD A,(HL)
23               INC HL
D7               RST 10               Escreve pontuação inicial.
10FB             DJNZ SC_LOOP
213939           LD HL,3939
222C68           LD (SCORE_C+3),HL
222D68           LD (SCORE_C+4),HL    Pontuação de momento
                                      = 999(00).
212158           LD HL,ATTRS+21       HL aponta para a
                                      posição corrente.
22925C           LD (POSITION),HL     Guarda posição
210000           LD HL,0000           corrente.
22945C           LD (LAST_MOVE),HL
212E68   LOOP    LD HL,SCORE_C+5      HL aponta o díg. das
                                      centenas da pontuação.
7E       DECIMAL LD A,(HL)
FE0F             CP 0F
```

```
2008            JR NZ,POSITIVE
0603            LD B,03
23      RESET   INC H_
3630            LD (HL),"0"
10FB            DJNZ RESET
C9              RET
3D      POSITIVE DEC A          Decrementa pontuação.
FE2F            CP 2F
2005            JR NZ,OK
3639            LD (HL),"9"
2B              DEC HL
18E9            JR DECIMAL
77      OK      LD (HL),A
212968          LD HL,SCORE__C  HL aponta p. os bytes
                                da pontuação corrente.
0606            LD B,06
7E      CS__LOOP LD A,(HL)
23              INC HL
D7              RST 10          Escreve pontuação
10FB            DJNZ CS__LOOP   corrente.
010008          LD BC,0800      Um atraso temporizado.
                                Alterar o valor inicial de
                                BC muda a velocidade
                                do jogo.

0B      DELAY   DEC BC
78              LD A,B
B1              OR C
20FB            JR NZ,DELAY
CD8E02          CALL KEY__SCAN  Lê o teclado.
7B              LD A,E          A: = tecla que foi
                                premida.
FE09            CP 09           Vf. se foi a tecla «J».
2811            JR Z,LEFT
FE10            CP 10           Vf. se foi a tecla «M».
2812            JR Z,DOWN
FE11            CP 11           Vf. se foi a tecla «K»
2813            JR Z,RIGHT
FE12            CP12            Vf. se foi a tecla «I».
20BF            JR NZ,LOOP
11E0FF  UP      LD DE,FFE0
180D            JR MOVE
11FFFF  LEFT    LD DE,FFFF
```



```
1808            JR MOVE
112000  DOWN    LD DE,0020
1803            JR MOVE
110100  RIGHT   LD DE,0001
2A945C  MOVE    LD HL,(LAST__MOVE) O jogador está preso na
                                   parede?
7C              LD A,H
B5              OR L
2805            JR Z,MOVE__OK
19              ADD HL,DE       Se sim, o jogador está a
                                voltar atrás?
7C              LD A,H
B5              OR L
20A1            JR NZ,LOOP
2A925C MOVE__OK LD HL,(POSITION)
7E              LD A,(HL)
EE07            XOR 07          Redefine o quad. com cor
77              LD (HL),A       preta ou branca conforme
                                necessário.
19              ADD HL,DE       Encontra nova posição.
7E              LD A,(HL)
EE07            XOR 07          Desenha a cruz em preto
77              LD (HL),A       ou branco.
22925C          LD (POSITION),HL Guarda a nova posição.
210000          LD HL,0000
FE38            CP 38           Atingimos uma parede?
2802            JR Z,FINISH     Se não, salta.
62              LD H,D
6B              LD L,E
22945C  FINISH  LD (LAST__MOVE),HL
2A925C          LD HL,(POSITION)
118558          LD DE,ATTRS+85
ED52            SBC HL,DE       Verifica se o quadrado
C8              RET Z           final foi ou não alcançado.
C37F68          JP LOOP
```

Note-se que a antepenúltima instrução SBC HL,DE depende do facto de ser zero o *flag carry*, proveniente da instrução CP 38. É, claro está, essencial que o *flag carry* seja zero antes de uma instrução SBC se quisermos obter uma verdadeira subtracção.

BREAKOUT

Nesta versão de BREAKOUT movemos uma raquete para a esquerda com a tecla *caps shift* e para a direita com a tecla de espaço. Emprega em muitos pontos as mesmas técnicas que temos empregado até agora — preenchendo o *écran* com o mesmo carácter mas escondendo-o por meio de uma inteligente manipulação da zona dos atributos. Neste jogo, no entanto, precisamos de dois símbolos — um para a raqueta, outro para a bola — e precisamos ainda de mostrar tijolos de cores diferentes e a pontuação corrente. Neste programa o que se passa é que todo o *écran* está coberto com símbolos «bola» que são em seguida «escondidos» fazendo o fundo e a cor da tinta da mesma cor em cada quadrado, branco para um quadrado vazio, preto para uma parede, ou qualquer outra cor para um tijolo.

Cada tijolo de cor diferente dará um número de pontos diferente — um para azul, dois para vermelho e assim por diante. A pontuação de cada cor é o algarismo da tecla acima da qual o nome dessa cor aparece.

BREAKOUT é um jogo com quatro partes: 1.º inicialização de um novo jogo; 2.º recomeço do jogo com uma nova bola; 3.º deslocação da bola; e 4.º deslocação da raqueta se necessário. Vamos ver agora cada um destes passos pormenorizadamente.

Primeiro, a parte de inicialização, que envolve a) imprimir o tabuleiro de jogo, b) definir a posição inicial da bola e c) definir a velocidade inicial do jogo e o número de vidas de que se dispõe, etc. Primeiro, vamos ver como imprimir o tabuleiro.

Temos uma sub-rotina cuja função é imprimir uma fila de tijolos em duas cores diferentes:

```
                ORG 6800
0608    BRICKS  LD B,08
77      BR_LOOP LD (HL),A
23              INC HL
77              LD (HL),A       Escreve o tijolo na
                                primeira cor.
23              INC HL
C609            ADD A,09        Muda a cor.
77              LD (HL),A
23              INC HL
77              LD (HL),A       Escreve o tijolo na
                                segunda cor.
23              INC HL
D609            SUB 09          Muda para a cor
                                anterior.
10F2            DJNZ BR_LOOP
C9              RET
```

Seguidamente temos alguns dados para utilização do programa:

```
16001A  SCORE   DEFM AT 0,26d;  Dados p. a rotina PRINT
                                SCORE mais à frente.
1007            DEFM tinta branca
1100            DEFM fundo preto
0000000000      DEFS 05         Espaço para guardar a
                                pontuação corrente.
```

E uma última sub-rotina, chamada AT_15_A e que simula PRINT AT 15h,A;

```
F5      AT_15_A PUSH AF
F5              PUSH AF
3E16            LD A, «Controle» AT
D7              RST 10
3E15            LD A,15         Primeira coordenada AT.
D7              RST 10
F1              POP AF          Segunda coordenada AT.
D7              RST 10
F1              POP AF
C9              RET
```

A partir daqui começa o próprio programa BREAKOUT e o código necessário para preparar o tabuleiro:

```
AF      START   XOR A           A: = zero, CARRY reset.
F5              PUSH AF         Empilha flags.
323C5C          LD (TVFLAG),A   PRINT na parte superior
                                do écran.
```

```
326B5C          LD (DF__SZ),A     Aumenta a parte sup. do
                                  écran p. 24 linhas.
01DF02          LD BC,02DF
3E0D            LD A,"enter"
D7              RST 10            Escreve a partir do início
                                  da segunda linha.
3E90    SET__UP LD A, «A-gráfico»
D7              RST 10            PRINT símbolo «bola».
0B              DEC BC
78              LD A,B
B1              OR C
20F8            JR NZ,SET__UP     Preenche todo o écran.
212158          LD HL,ATTRS+21    HL aponta p. o prim.
                                  quadrado branco.
363F            LD (HL),3F        Cores branco em branco.

112258          LD DE,ATTRS+22    Aponta para o segundo
                                  quadrado branco.
01DE02          LD BC,02DE
EDB0            LDIR              Torna brancos todos os
                                  quadrados necessários.
218158          LD HL,ATTRS+58    HL aponta p. o prim. tijolo.
3E09            LD A,09
CD0068          CALL BRICKS       Escreve fila de tijolos
                                  azuis e vermelhos.
3E2D            LD A,2D
CD0068          CALL BRICKS       Escreve uma fila de tijolos
                                  cyan e amarelos.
3E1B            LD A,1B
CD0068          CALL BRICKS       Escreve fila de tijolos
                                  magenta e verdes.
3E09            LD A,09
CD0068          CALL BRICKS       Escreve fila de tijolos
                                  azuis e vermelhos.

AF              XOR A
210058          LD HL,ATTRS       Aponta o canto superior
                                  esquerdo da parede.
77              LD (HL),A         Faz este quad. preto.
110158          LD DE,ATTRS+01    Aponta o segundo
                                  quadrado da parede.
011F00          LD BC,001F
C5              PUSH BC
```



```
EDB0            LDIR              Torna preta o resto da
                                  parede superior.
D1              POP DE            DE: = 001F
0617            LD B,17
23      SIDES   INC HL            Aponta o pr. quad. da
                                  parede esquerda.
77              LD (HL),A         Faz este quad. preto.
19              ADD HL,DE         Aponta o pr. quad. da
                                  parede direita.
77              LD (HL),A         Faz este quad. preto.
10FA            DJNZ SIDES
211868          LD HL,SCORE+07    Aponta para o espaço
                                  reservado SCORE.
0604            LD B,04
3620    SC__LOOP LD (HL), «espaço» Preenche com espaços.

23              INC HL
10FB            DJNZ SC__LOOP
3630            LD (HL),"0"       Pontuação igual a zero.
2E11            LD L,11           HL aponta p. a etiqueta
                                  SCORE.
060C            LD B,0C
7E      PR__SC  LD A,(HL)
23              INC HL
D7              RST 10            Escreve a pontuação no
                                  écran.
10FB            DJNZ PR__SC
```

Observe-se como neste jogo a pontuação está guardada independentemente do *écran*, sendo, além disso, guardada em decimal, dígito por dígito. Isto porque é mais fácil ler um carácter simples de uma posição de memória fixa do que decifrar um padrão de *pixels* de oito posições fixas de memória diferentes. Por, fim, para preparar a posição da bola, velocidade e número de vidas, o procedimento é:

```
21605A          LD HL,ATTRS+0260
22B05C          LD (BALL__INIT),HL  Guarda a posição de
                                    partida da bola.
21000B          LD HL,0B00
```

```
22AE5C          LD (SPEED),HL     Guarda a velocidade
                                  inicial de jogo.
FD363102        LD (DF__SZ),02    Restaura a variável de
                                  sistema p. evitar crash.
3E09            LD A,09
32A85C          LD (LIVES),A      Guarda o número de
                                  vidas mais um.
```

Isto é, na verdade, toda a inicialização de que precisamos. Faltam várias coisas — por exemplo, apesar de definida a posição da bola, ela não é escrita. A raqueta não é sequer mencionada! A razão por que isto acontece é que a raqueta é desenhada de cada vez que o jogo recomeça e a bola também. Então porquê preocuparmo-nos com a posição inicial? Bem, nesta versão a bola começa numa posição ligeiramente diferente de cada vez. Isto assegura a possibilidade de apagar todos os tijolos.

A variável SPEED determina a velocidade do jogo; controla um ciclo de atraso — necessário para abrandar o andamento, uma vez que o código máquina é muito rápido. Em resumo, quanto maior o número mais lento é o jogo e quanto mais pequeno o número mais rápido é o jogo. A velocidade, no entanto, não é constante ao longo do jogo, ao contrário de ESPIRAIS. Neste jogo a velocidade aumenta à medida que o tempo avança!

A segunda secção do jogo realiza as seguintes tarefas: a) muda a posição inicial da bola e coloca-a na nova posição; b) faz com que a direcção inicial do movimento seja para cima e para a direita; c) escreve a raqueta e ao mesmo tempo apaga qualquer símbolo de raqueta anterior que pudesse existir; d) decrementa o número de vidas restantes e termina se este número atinge zero; e e) dá ao jogador a hipótese de recuperar da última secção, proporcionando um intervalo de tempo entre o desaparecimento de uma bola e o aparecimento da seguinte. Observe-se, nesta secção, como se descreve a raqueta e se apaga a anterior.

```
3AA85C  RESTART LD A,(LIVES)
3D              DEC A
2002            JR NZ,L__OK
F1              POP AF            «Equilibra» a pilha.
C9              RET               Retorna ao BASIC se não
                                  houver mais vidas.
```



```
32A85C  L__OK   LD (LIVES),A
2AB05C          LD HL,(BALL__INIT)
23              INC HL            Muda a posição de início
                                  da bola.
22B05C          LD (BALL__INIT),HL
22AC5C          LD (BALL__POS),HL Define a posição actual.
3639            LD (HL),39        Escreve a bola (azul em
                                  branco).
21E1FF          LD HL,FFE1
22AA5C          LD (DIRECTION),HL Direcção inicial: para cima
                                  à direita.
3E03            LD A,03
32A95C          LD (BAT__POS),A   Centro da raqueta na
                                  coluna três.
3E01            LD A,01
CD1D68          CALL AT__15__A    Print AT 15h,1;.
FD365507        LD (ATTR__T),07   Cores: branco em preto.
0605            LD B,05
3E8C  BAT__LOOP LD A, «gráfico-shift 3»
D7              RST 10
10FB            DJNZ BAT__LOOP    Escreve o símbolo da
                                  raqueta.
FD36553F        LD (ATTR__T),3F   Cores: branco em branco.
0619            LD B,19
3E90    ERASE   LD A, «A-gráfico»
D7              RST 10            Apaga o símbolo da
                                  raqueta anterior.
10FB            DJNZ ERASE
0604            LD B,04
210000  RDL__1  LD HL,0000
2B      RDL__2  DEC HL
7C              LD A,H
B5              OR L
20FB            JR NZ,RDL__2      Entra um longo ciclo de
10F6            DJNZ RDL__1       espera p. o jogador
                                  recuperar p. a pr. bola.
```

Para escrever a bola temos de passar primeiro por um ciclo de temporização (controlado por SPEED — a velocidade de jogo) e depois apagar a posição anterior da bola. É examinado o conteúdo do quadrado seguinte na direcção que a bola leva e, se aí se encontrar um símbolo de raqueta, inverte-se a direcção mas a bola

não se move; de outro modo a bola mover-se-á na direcção dada, não sendo escrita na nova posição.

São então verificadas uma série de condições. Se o conteúdo da nova posição da bola (note-se que esta não foi ainda escrita nessa posição) for um tijolo colorido, a pontuação aumenta, a componente vertical da direcção da bola inverte-se (embora a componente horizontal se mantenha inalterada) e a velocidade de jogo aumenta.

Se o quadrado seguinte acima ou abaixo (dependendo da direcção que a bola leva) for uma parede ou uma raqueta, a componente horizontal da direcção da bola é invertida. Se o quadrado seguinte à esquerda ou à direita (dependendo do sentido de deslocamento da bola) for uma parede ou uma raqueta, inverte-se a componente horizontal da velocidade.

Estes passos asseguram que o quadrado seguinte na direcção da bola se encontre vazio e que a bola fará sempre ricochete nas paredes e na raqueta com os ângulos correctos. A parte do programa responsável por isto é a que segue:

```
2AAE5C  LOOP      LD HL,(SPEED)
2B      DELAY     DEC HL          Ciclo de atraso curto que
7C                LD A,H          controla a velocidade do
                                  jogo.
B5                OR L
20FB              JR NZ,DELAY
F1                POP AF          Recupera o flag carry.
3F                CCF             Complementa-o.
F5                PUSH AF         Torna a guardá-lo.
DA9369            JP C,MOVE_BAT   A bola só se move cada
                                  duas passagens; assim a
                                  raqueta move-se duas
                                  vezes mais depressa que
                                  a bola.

2AAC5C            LD HL,(BALL_POS) HL aponta para a bola.
363F              LD (HL),3F      Apaga a posição anterior
                                  da bola.
ED5BAA5C          LD DE,(DIRECTION)
19      NEW_POS   ADD HL,DE       Calcula a nova posição
                                  da bola.
22AC5C            LD (BALL_POS),HL Guarda esta posição.
```



```
7E                LD A,(HL)
FE07              CP 07           A raqueta está nesta
                                  nova posição?
200D              JR NZ,SQ_FREE   Salta se não estiver.
7B                LD A,E
ED44              NEG
5F                LD E,A
7A                LD A,D
2F                CPL             Troca o sinal de DE
                                  (direcção de deslocamento).
57                LD D,A
ED53AA5C          LD (DIRECTION),DE Guarda a nova direcção.
18EA              JR NEW_POS
3639    SQ_FREE   LD (HL),39       Escreve aí a bola (azul
                                  em branco).
FE3F              CP 3F           Antes, estava um tijolo
                                  nessa posição?
282A              JR Z,V_CH       Salta se não.
E607              AND 07          A: = pontuação por
                                  atingir o tijolo.
2AAE5C            LD HL,(SPEED)
01F0FF            LD BC,FFF0      BC: = -10.
09                ADD HL,BC
22AE5C            LD (SPEED),HL   Aumenta ritmo de jogo.
211C68            LD HL,SCORE+0B  HL aponta p. o últ. díg.
                                  da pontuação.
86                ADD A,(HL)      A: = díg. das novas unds.
FE3A    CARRY     CP 3A           O novo dígito é maior
                                  que nove?
380A              JR C,INC        Salta se não for.
D60A              SUB 0A          Calcula o díg. correcto.
77                LD (HL),A       Guarda este dígito.
2B                DEC HL          Aponta pr. díg. à esquerda.
7E                LD A,(HL)       A: = próximo dígito.
F610              OR 10           Muda de «espaço» p.
                                  «zero» se necessário.
3C                INC A           Adiciona o carry do
                                  último dígito.
18F2              JR CARRY
77      INC       LD (HL),A       Guarda o dígito.
2E11              LD L,11         HL aponta para a
                                  etiqueta SCORE.
```

```
060C              LD B,0C
7E      PR__SC__2 LD A,(HL)
23                INC HL
D7                RST 10        Escreve a pontuação.
10FB              DJNZ PR__SC__2
180F              JR VERT
D5      V__CH     PUSH DE       Empilha a direcção.
7A                LD A,D
E6C0              AND C0
F620              OR 20
5F                LD D,A        DE: = componente
                                vertical de
                                direcção.
2AAC5C            LD HL,(BALL__POS HL: = posição corrente
                                da bola.
19                ADD HL,DE
7E                LD A,(HL)     A: = byte de atributos
                                do quad. computado.
E6F8              AND F8        Ignora a cor da tinta.
D1                POP DE        DE: = dir. verdadeira.
200B              JR NZ,H__CH   Salta a menos que o
7B      VERT      LD A,E        fundo seja preto (isto é,
                                raqueta ou parede).
EEC0              XOR C0
5F                LD E,A
7A                LD A,D
2F                CPL
57                LD D,A        Inverte a componente
ED53AA5C          LD (DIRECTION),DE vertical de direcção.
D5      H__CH     PUSH DE       Empilha a direcção.
CB1B              RR E
CB1B              RR E
9F                SBC A,A       A: = FF (p. a esquerda)
                                ou 00 (p. a direita).
57                LD D,A
F601              OR 01         A: = FF (p. a esquerda)
                                ou 01 (p. a direita).
5F                LD E,A        DE: = componente
                                horizontal de direcção.
2AAC5C            LD HL,(BALL__POS HL: = posição corrente
                                da bola.
19                ADD HL,DE
```



```
7E                LD A,(HL)     A: = conteúdo do
                                quadrado calculado.
E6F8              AND F8        Ignora a cor da tinta.
D1                POP DE        DE: = dir. verdadeira.
2008              JR NZ,L__CH   Salta a menos que o
                                quad. seja parede ou
                                raqueta.
7B                LD A,E
EE3E              XOR 3E
5F                LD E,A        Inverte a componente
ED53AA5C          LD (DIRECTION),DE horizontal de direcção.
2AAC5C  L__CH     LD HL,(BALL__POS) HL: = posição corrente
                                da bola.
11E05A            LD DE,ATTRS+02E0
A7                AND A
ED52              SBC HL,DE
3806              JR C,MOVE__BAT
19                ADD HL,DE     Se a bola está na fila de
363F              LD (HL),3F    baixo do écran, apaga-a
C3A468            JP RESTART    e recomeça com nova
                                bola.
```

Um ponto interessante a observar é o modo como a pontuação é aumentada.

Compare-se o mecanismo utilizado aqui com o empregado em ESPIRAIS para diminuir a pontuação. Existem uma ou duas diferenças entre este e os outros. Em primeiro lugar, é claro, a pontuação de BREAKOUT aumenta em vez de diminuir e, em segundo, a pontuação pode aumentar qualquer quantidade (inferior a dez) de uma só vez.

Para se compreender a parte de mudança de direcção o melhor é experimentar o seu efeito. Note-se que as únicas direcções empregadas são FFDF (para cima à esquerda), FFE1 (para cima à direita), 001F (para baixo à esquerda) e 0021 (para baixo à direita). Todos estes casos serão convertidos conforme o necessário pelas rotinas acima dadas.

Para movimentar a raqueta, antes de mais lê-se o teclado. Se mais do que uma tecla for premida de uma só vez, a raqueta mantém-se estacionária. Se *caps shift* for premido, a raqueta move-se para a esquerda e se *space* for premido, a raqueta move-se para a direita.

Note-se que a raqueta só se moverá se o quadrado seguinte no sentido do momento estiver livre. Se estiver bloqueado ou por uma parede ou por uma bola (o que acontecerá em ocasiões muito raras mas mesmo assim de considerar), a raqueta não se move.

Vamos estudar esta parte final do programa e observar a maneira como a raqueta se move. Recordemos que a variável BAT__POS guarda o número da coluna do centro da raqueta.

```
CD8E02  MOVE__BAT
                CAL_ KEY__SCAN
14              INC D
2042            JR NZ,CYCLE     Raqueta imóvel se
                                premir mais de uma tecla.
7B              LD A,E          A: = cód. da tecla
                                premida (FF se nenhuma).
FE20            CP 20           Tecla premida foi
2808            JR Z.RIGHT      space?
FE27            CP 27           Foi caps shift?
2039            JR NZ,CYCLE
1601    LEFT    LD D,01
1802            JR M__BAT__1
16FF    RIGHT   LD D,FF
3AA95C  M__BAT__1 LD A,(BAT__POS) A: = n.º de coluna do
                                centro da raqueta.
92              SUB D
92              SUB D
92              SUB D           A: = n.º de coluna do pr.
                                quadrado na linha.
F5              PUSH AF
C6A0            ADD A,A0
6F              LD L,A
265A            LD H,5A         HL: = end. deste quad.
                                na zona dos atributos.
7E              LD A,(HL)
FE3F            CP 3F           Este quad. está vazio?
2803            JR Z,M__BAT__2  Salta se estiver.
F1              POP AF          «Equilibra» a pilha.
181F            JR CYCLE        A raqueta não se move.
F1      M__BAT__2 POP AF
CD1D68          CALL AT__15__A  Mova a posição de print
                                p. este quadrado.
```



```
FD365507        LD (ATTR__T),07 Cores: branco em preto.
F5              PUSH AF
3E8C            LD A, «gráfico shift 3»
                3'
D7              RST 10          Aumenta a raqueta na
                                direcção necessária.
F1              POP AF
82              ADD A,D
82              ADD A,D
32A95C          LD (BAT__POS),A Guarda a nova posição
                                da raqueta.
82              ADD A,D
82              ADD A,D
82              ADD A,D
CD1D68          CALL AT__15__A  Move a posição print p.
                                outro extremo da raqueta.
FD36553F        LD (ATTR__T),3F Cores: branco em
                                branco.
3E90            LD A, «A gráfico»
D7              RST 10          Apaga o anterior
                                extremo da raqueta.
C3ED68  CYCLE   JP LOOP         Repete tudo.
```

CAPÍTULO DEZASSEIS

Damas – terceira parte

A parte da história que já conhecemos é... Era uma vez um ser humano que introduziu uma jogada num *ZX Spectrum*. O computador verificava essa jogada para se certificar de que não havia batota e, se houvesse, condenava o pobre humano a um duro castigo (que seria ter de escrever o movimento todo de novo). O movimento foi aceite. O computador começou a procurar no tabuleiro por peças para mover. De repente, depara-se-lhe miraculosamente com a sub-rotina EVALUATE. Para onde é que seguimos a partir daqui? O bom senso diz-nos que a primeira parte é bastante clara. Serve para responder à pergunta «Podemos mover a peça?» e se não, para voltar à parte anterior do programa. Se pudermos mover a peça... deixaremos este problema para mais tarde.

Este é o início da rotina EVALUATE. Como já dissemos a sua função é decidir se um movimento é ou não possível. Vamos escrever a sub-rotina a partir do endereço 6C1A:

```
E5      EVALUATE PUSH HL          Empilha o endr. que
                                  estamos a considerar.
FD3458           INC (P__COUNT)   Conta o número de
                                  peças do computador
                                  no tabuleiro.
0604             LD B,04
7E               LD A,(HL)        A: = 07 p. uma peça, 87
                                  p. uma dama.
17               RLA
3802             JR C,EV__LOOP    Salta se for dama.
05               DEC B
05               DEC B            B: = número de
                                  direcções permitidas.
48      EV__     LD C,B           C: = número da pr.
        LOOP                      direcção a considerar.
0D               DEC C            C: = 00, 01,02 ou 03.
```



```
CB39             SRL C            C: = 00 (sem CARRY),
                                  00 (CARRY), 01 (sem
                                  CARRY) ou 01 (CARRY).
3E05             LD A,05
3001             JR NC,EV__1      Salta se o movimento for
                                  um ou três.
3C               INC A            A: = 05, 06, 05 ou 06.
0D      EV__1    DEC C            C: = FF, FF, 00 ou 00.
2002             JR NZ,EV__2      Salta se o movimento for
                                  um ou dois.
ED44             NEG              A: = 05, 06, FB ou FA.
4F      EV__2    LD C,A           C: = direcção do
                                  deslocamento.
85               ADD A,L
6F               LD L,A           HL: = quad. de destino.
7E               LD A,(HL)        A: = conteúdo desse
                                  quadrado.
E67F             AND 7F           Ignora o status da dama.
2811             JR Z,MV__FOUND   Salta com quad. vazio.
FE02             CP 02
2007             JR NZ,EV__3      Salta, a menos que o
                                  quad. tenha peça
                                  humana.
7D               LD A,L
81               ADD A,C
6F               LD L,A           HL: = novo quadrado de
                                  destino.
7E               LD A,(HL)        A: = conteúdo desse
                                  quadrado.
A7               AND A
2806             JR Z,MV__FOUND   Com quad. vazio, salta.
E1      EV__3    POP HL           HL: = quadrado original
                                  considerado.
E5               PUSH HL
10DB    EV__DJ   DJNZ EV__LOOP    Repete para todas as
                                  direcções permitidas.
E1               POP HL           Restaura valor original de HL
C9               RET              Fim da sub-rotina.
        MV__FOUND $
```

Eis como funciona: B representa o «número da direcção» a considerar — que é ou um (direcção D), dois (direcção C), três

(direcção B), ou quatro (direcção A). Note-se que, no caso de uma dama, B terá de percorrer sucessivamente todos estes valores, mas, se for uma peça normal, B só tomará os valores dois e um, porque as peças não podem andar para trás. A direcção do movimento é calculada a partir de B no registo A e guardada, para o que for necessário, no registo C. A partir daí tudo se torna evidente. Considera-se possível um movimento se, 1.º, o quadrado seguinte nesta direcção estiver vazio ou, 2.º, se o quadrado seguinte nesta direcção estiver ocupado por uma dama ou peça normal do jogador humano mas o quadrado a seguir, nessa direcção, estiver vazio. Se for encontrado um movimento válido, o controle salta para a etiqueta MV__FOUND, parte do programa que ainda temos de escrever; de outro modo o ciclo será repetido até se terem experimentado todas as direcções.

A parte que se segue é a mais interessante porque tem a função de atribuir um valor numérico — uma prioridade — a esta escolha de jogada (quanto mais alto o número, melhor a jogada) e manipular a lista correspondente.

A melhor maneira de o conseguir é fazer uma cópia do tabuleiro (mais uma cópia!) de modo a manipularmos BOARD__2 tanto quanto quisermos sem perigo de o destruir totalmente. Vamos arranjar um pouco de espaço para esta cópia — que será chamada BOARD__3. Introduzimos 42 (decimal) espaços a partir do endereço 6C1A de modo que a rotina EVALUATE comece agora no endereço 6C44. Estes espaços serão utilizados para guardar a nossa cópia. O que se segue é, portanto, a primeira parte da rotina MV__FOUND, que devemos escrever a partir de 6C77:

```
210668    MV__FOUND   LD HL,BOARD__2
                      +6          HL aponta p. o prim.
                                  quadrado jogável.
111A6C                LD DE,BOARD__3  DE aponta p. o prim.
                                  espaço reservado.
C5                    PUSH BC
012A00                LD BC,002A
EDB0                  LDIR        Copia o tabuleiro.
```

E a seguir, para efectuar o movimento «experimental» (escrevendo a partir de 6C83):


```
C1                    POP BC      C: = direcção de
                                  deslocação.
E1                    POP HL      HL: = quadrado inicial
                                  do movimento.
E5                    PUSH HL
C5                    PUSH BC
1E00                  LD E,00     Pontuação inicial
                                  começa em zero.
46                    LD B,(HL)   B: = peça do comp.
73                    LD (HL),E   Apaga a peça do comp.
                                  do quad. corrente.
7D                    LD A,L
81                    ADD A,C
6F                    LD L,A      HL: = quad. de destino.
7E                    LD A,(HL)   A: = conteúdo deste
                                  quadrado.
A7                    AND A
280D                  JR Z,MV__1  Salta excepto se for
                                  tomada uma peça.
73                    LD (HL),E   Apaga a peça tomada.
1E81                  LD E,81
17                    RLA         CARRY: = se for peça,
                                  reset, se dama, set.
CB13                  RL E        E: = 02 (peça) ou 03
                                  (dama).
CB13                  RL E        E: = 05 (peça) ou 07
                                  (dama).
CB 13                 RL E        E: = 0A (peça) ou 0E
                                  (dama).
7D                    LD A,L
81                    ADD A,C
6F                    LD L,A      HL: = novo quadrado de
                                  destino.
70          MV__1     LD (HL),B   Coloca peça na nova
                                  posição.
```

Uma sub-rotina agora — sim, é isso mesmo, uma sub-rotina dentro de outra sub-rotina. A função desta é determinar se uma peça do computador está ou não em perigo de ser tomada, de uma direcção dada. A sub-rotina requer que HL aponte para o quadrado em questão e C contenha a direcção 05 ou 06. A sub-rotina deve

ser inserida (já que precisamos de saber o seu endereço absoluto para a chamarmos) no endereço 6C44.

```
E5          DANGER  PUSH HL
7D                  LD A,L
91                  SUB C
47                  LD B,A
7D                  LD A,L
81                  ADD A,C
6F                  LD L,A          HL aponta o próximo
                                    quadrado abaixo.
7E                  LD A,(HL)
E67F                AND 7F          Ignora o status da dama.
2006                JR NZ,DN__1     Salta com quad. não vazio.
68                  LD L,B          HL aponta o próximo
                                    quadrado acima.
7E                  LD A,(HL)
FE82                CP 82           Este quad. contém dama
                                    do jogador humano?
280C                JR Z,DN__3      Se assim for, salta.
FE02        DN__1   CP 02           Este quad. contém peça
                                    ou dama humana?
2005                JR NZ,DN__2     Se não, salta.
68                  LD L,B          HL aponta o próximo
                                    quadrado acima.
7E                  LD A,(HL)
A7                  AND A
2803                JR Z,DN__3      Salta com quad. vazio.
E1          DN__2   POP HL         Restaura valor original de HL
A7                  AND A           Flag carry reset.
C9                  RET
E1          DN__3   POP HL         Restaura original.
37                  SCF             Flag carry set, indicando
                                    perigo.
C9                  RET
```

O endereço do início da rotina EVALUATE será agora 6C65, por causa desta inserção. Em seguida vamos debruçar-nos sobre o código máquina que utiliza esta sub-rotina. Observemos a figura 16.1. Ela atribui um valor a cada um dos quadrados de modo que alguns são considerados mais importantes que outros. A rotina

seguinte percorre o tabuleiro, casa por casa, e cada vez que encontra uma peça do computador ajusta a «pontuação até o momento» (registo E) tendo em conta o valor atribuído à casa, e também se a peça está em risco de ser tomada. Escrevemos esta rotina a partir do endereço 6CC1.

	06 0		07 0		08 0		09 0
0B 0		0C 1		0D 0		0E 1	
	11 0		12 0		13 0		14 0
16 1		17 2		18 2		19 0	
	1C 0		1D 2		1E 2		1F 1
21 0		22 0		23 0		24 0	
	27 0		28 0		29 0		2A 0
2C 0		2D 0		2E 0		2F 0	

Figura 16.1

```
210668              LD HL,BOARD__2   HL aponta p. o prim.
                    +06              quadrado jogável.
7E      DN__LOOP    LD A,(HL)        A: = conteúdo do pr.
                                     quad. a considerar.
E67F                AND 7F           Ignora o status da dama.
FE07                CP 07            Peça do computador?
203B                JR NZ,DN__NS     Salta se não for.
7D                  LD A,L
D60C                SUB 0C
281B                JR Z,P__1        O quadrado 0C tem
                                     valor 01.
```

```
D602          SUB 02
2817          JR Z,P__1          O quadrado 0E tem valor 01.

D608          SUB 08
2813          JR Z,P__1          O quadrado 16 tem valor 01.

3D            DEC A
280F          JR Z,P__2          O quadrado 17 tem valor 02.

3D            DEC A
280C          JR Z,P__2          O quadrado 18 tem valor 02.

D605          SUB 05
2808          JR Z,P__2          O quadrado ID tem valor 02.

3D            DEC A
2805          JR Z,P__2          O quadrado IE tem valor 02.

3D            DEC A
2803          JR NZ,P__1         O quadrado IF tem valor 01.

1802          JR P__0            Todos os outros quadrados têm valor 00.

1C      P__2  INC E
1C      P__1  INC E
7E      P__0  LD A,(HL)          A: = peça do comp.
17            RLA
1681          LD D,81
CB12          RL D               D: = 02 (peça) ou 03 (dama).
CB12          RL D               D: = 05 (peça) ou 07 (dama).
CB12          RL D               D: = 0A (peça) ou 0E (dama).

0E05          LD C,05
CD446C        CALL DANGER        Verifica se há perigo na direcção 05
3806          JR C,DN__FOUND     Peça em perigo: salta.
0C            INC C
CD446C        CALL DANGER        Verifica se há perigo na direcção 06.
3003          JR NC,DN__NS       Peça em segurança: salta.
```



```
7B      DN__  LD A,E             A: = pontuação actual.
        FOUND
92            SUB D
5F            LD E,A             Diminui a pontuação conforme o necessário.

7D      DN__NS LD A,L
23            INC HL             HL aponta p. o pr. quadrado a considerar.
FE2F          CP 2F              Já considerámos todo o tabuleiro?
20B8          JR NZ,DN__LOOP     Salta se ainda não.
```

A próxima alteração que temos de fazer à pontuação é atribuir um bónus de sete pontos a qualquer peça que atinja a última fila e se transforme em dama. Este código máquina, que deve ser escrito a partir do endereço 6D0C, encarregar-se-á disso.

```
0604          LD B,04
2B      KCH__LP DEC HL           Aponta p. o pr. quadrado da últ. fila.
7E            LD A,(HL)
FE07          CP 07              Existe uma peça do comp. (ainda não dama) na última fila?
2004          JR NZ,KCH__N       Salta se não.
7B            LD A,E
C607          ADD A,07
5F            LD E,A             Se sim, aumenta a
10F4    KCH__N DJNZ KCH__LP      pontuação.
```

E para restaurar o tabuleiro original (escreve-se a partir de 6D1A):

```
D5            PUSH DE
211A6C        LD HL,BOARD__3
110668        LD DE,BOARD__2+6
012A00        LD BC,002A
EDB0          LDIR
D1            POP DE             E: = pontuação.
```

```
7B          LD A,E      A: = pontuação.
C680        ADD A,80    A: = prioridade atribuída
                        ao mov.
57          LD D,A      D: = prioridade atribuída
                        ao mov.
1E01        LD E,01     E: = número de passos
                        no movimento.
```

Notemos que se somam 80 à pontuação final para evitar que a prioridade seja negativa. Isto destina-se apenas a que a nossa aritmética seja mais fácil. E por fim, para ajustar a lista como queremos, um pedaço de código; vamos escrevê-lo a partir do endereço 6D2D:

```
D9          EXX
C1          POP BC      B: = n.º de direcção.
                        C: = direcção de
                        deslocamento.
E1          POP HL      HL: = endr. do quad.
                        inicial do movimento.
D1          POP DE      DE: = endereço de
                        retorno da sub-rotina.
D9          EXX
E1          POP HL      HL: = prioridade dos
                        itens na lista.
A7          AND A
ED52        SBC HL,DE
19          ADD HL,DE
280A        JR Z,L__ADD Acrescenta a lista se a
                        prioridade for a mesma.
3014        JR NC,L__OK Rejeita o mov. se não
                        for bastante bom.
ED7BB05C    LD SP,(L__BASE) Anula a lista anterior.
FD365900    LD (L__COUNT),00 Número de itens na lista
                        igual a zero.
D9      L__ADD EXX
61          LD H,C
E5          PUSH HL     Empilha o quadrado de
                        início e a direcção.
2668        LD H,68     HL: = endr. do quad. de
                        onde se inicia o mov.
```



```
D9          EXX
D5          PUSH DE     Empilha o número de
                        passos e a prioridade.
FD3459      INC (L__COUNT) Conta o número de itens
                        na lista.
1801        JR L__EXIT
E5      L__OK PUSH HL   Empilha o anterior n.º de
                        passos e a prioridade.
215827  L__EXIT LD HL,2758
D9          EXX         HL': = 2758 para evitar
                        crash.
                        B: = n.º de direcção.
                        C: = direcção de
                        deslocação.
                        DE: = endereço de
                        retorno da sub-rotina.
                        HL: = quad. de onde se
                        inicia o movimento.
D5          PUSH DE     Torna a empilhar o endr.
                        de retorno da sub-rotina.
E5          PUSH HL     Torna a empilhar o
                        endereço inicial.
C3946C      JP EV__DJ   Retoma ciclo de
                        verificação.
```

Para alterar o endereço a que se refere a instrução CALL Z, EVALUATE, fazemos RUN 100,6AFB/656C. O programa de damas está agora completo, ou pelo menos tão completo quanto pretendemos deixá-lo. Da maneira em que as coisas se encontram, a instrução RUN 700 iniciará um jogo que decorrerá de uma forma muito razoável, mas note-se o seguinte:

1.º Não é obrigatório tomar peças susceptíveis de serem tomadas, portanto uma peça não será retirada do tabuleiro só porque se recusa a tomar outra peça;

2.º O programa, no seu estado actual, admite outros melhoramentos — o que foi deliberadamente deixado para os jogadores. Apesar de o jogador poder efectuar saltos múltiplos, e apesar de o computador jogar bastante bem, a máquina ainda não é suficientemente «esperta» para realizar saltos múltiplos por si própria. O computador dá os seus saltos quando acha conveniente, mas só um salto em cada jogada, mesmo onde é possível saltar três ou quatro

peças de uma só vez. É este, portanto, o desafio que pomos ao leitor. Até aqui, foi-lhe apresentado o trabalho todo feito; agora é a sua vez. O princípio é o mesmo que para fazer saltos simples mas o procedimento exacto necessita de algumas alterações, porque há mais do que uma direcção a guardar durante a avaliação da prioridade (deve guardá-las em RAM em vez de o fazer nos registos) e mais do que uma direcção a «empilhar» na lista.

3.º Não é de momento possível oferecer um empate. A situação de empate (em que um jogador ainda tem peças mas não as pode movimentar) não é verificada ou permitida tanto para o computador como para o humano. No entanto, adaptar o programa para que tal aconteça é bastante simples — deve verificar se a jogada introduzida tem uma forma especial que signifique «empate» (para o humano) e ver simplesmente a variável L__COUNT (para o computador).

Agora só nos resta fazer um pequeno trabalho de «arrumação»: RUN 8ØØ/3E1Ø/3EAF/7CBA (16 K ; aqueles que utilizam 48 K devem mudar o último endereço para FCBA) irá restaurar toda a zona dos caracteres gráficos, de gráfico B a gráfico U, deixando o gráfico A inalterado. RUN 1ØØ/694F/14ØØ muda a instrução implícita «GO TO 71Ø» no código máquina para um «GO TO 2Ø» implícito. Agora vamos escrever o seguinte:

```
NEW
1Ø RANDOMISE USR 26706
2Ø INPUT LINE A$
3Ø RANDOMISE USR 26881
4Ø LOAD "" CODE
5Ø LOAD "" CODE USR "A"
SAVE «Damas» LINE 4Ø
SAVE "Dr mc" CODE 26624,137Ø
SAVE "Dr gr" CODE USR "A",8
```

Temos, é claro, a faculdade de alterar o esquema de cores. Só necessitamos agora, para começar o jogo, de fazer RUN. É claro que se quisermos melhorar DAMAS, como foi sugerido, precisaremos de HEXLD 3. Neste caso, deve-se deixar esta parte do BASIC do programa de DAMAS como estava, a partir da linha 7ØØ. A decisão, no entanto, é problema de cada um!

CAPÍTULO DEZASSETE

Desassemblar a ROM

Existem três "níveis" nos quais podemos desassemblar, cada um deles um pouco mais sofisticado que o anterior. Os primeiros dois níveis não são muito satisfatórios, mas são, em contrapartida, muito fáceis de programar.

O primeiro nível, já o alcançámos — a rotina USR H__LIST em HEXLD 3, que vimos anteriormente, faz isso precisamente. Isto é, dado um endereço como 15F2, produzirá uma saída como esta:

```
15F2 D9
15F3 E5
15F4 2A *
15F5 51 Q
15F6 5C \
```

e assim por diante. Isto não é verdadeiramente desassemblar, apesar de podermos, é claro, procurar o significado destes *bytes* nas tabelas do fim do livro, mas esta tarefa é muito demorada, sendo ainda muito provável que nos percamos a meio caminho. O segundo «nível» não é muito melhor, mas é também muito fácil de programar. Estamos a falar de uma saída como esta:

```
15F2 D9
15F3 E5
15F4 2A515C
15F7 5E
....
```

e assim por diante. Como se vê, cada instrução aparece sob a forma de uma sequência com o número certo de *bytes*. Isto produz uma listagem de aspecto muito agradável, havendo poucas ou nenhumas hipóteses de nos perdermos ao procurar estes *bytes* nas

tabelas. O nosso objectivo, no entanto, é alcançar o terceiro nível. O que queremos é uma saída como esta:

```
15F2 EXX
15F3 PUSH HL
15F4 LD HL,(5C51)
15F7 LD E,(HL)
....
```

e assim por diante. É talvez bastante fácil de programar; só temos de fazer com que o computador procure as palavras apropriadas numa tabela em vez de sermos nós a fazê-lo. No entanto seria necessário muito espaço só para guardar a tabela (cerca de quatro ou cinco K). O método que vamos descrever permite-nos condensar este programa em menos de um K e um quarto, mas consegui-lo é bastante difícil. Existe na realidade um quarto nível de desassemblagem, que nem tentaremos tocar, mas que interessa conhecer. Imaginemos uma saída deste género:

```
D9          PRINT_A  EXX
E5                   PUSH HL
2A515C               LD HL,(CURCHL)
5E                   LD E,(HL)
....
```

Como dissemos, não vamos sequer tocar neste assunto neste nível, só precisamos de mais uma coisa, uma outra tabela, contendo desta vez todas as etiquetas empregadas. Voltando um pouco atrás, vamos agora tratar de coisas mais simples. Comecemos por considerar uma versão um pouco melhorada de H_LIST que atinja o segundo nível de desassemblagem e calcule o comprimento de cada instrução antes de escrever os *bytes* correspondentes. Só necessitamos de uma tabela que contenha dois elementos de informação para cada *byte*. São eles: 1.º, o número de *bytes* de cada instrução que comece com este *byte;* e 2.º, o número de *bytes* de cada instrução que comece com DD ou FD seguido deste *byte*. Como se sabe, surge alguma confusão com as instruções que começam por CD ou ED, mas na realidade não necessitamos de mais nenhuma tabela ou qualquer outra coisa especialmente destinada a estas instruções desde que nos lembremos das seguintes regras:

— todas as instruções que comecem com CB têm dois *bytes* de comprimento;
— todas as instruções que comecem com DDCB ou FDCB têm quatro *bytes* de comprimento;
— todas as instruções que comecem com ED têm dois *bytes* de comprimento, excepto LD BC, (pq), LD DE, (pq), LD SP, (pq), LD (pq), BC, LD (pq), DE e LD (pq), SP. O *byte* imediatamente a seguir a ED para estas seis instruções é, respectivamente, 4B, 5B, 7B, 43, 53 ou 73. Em binário estes seis números têm a forma Ø1 ** *Ø11.

Pode estranhar-se a aparente irregularidade na sequência de *bytes*, onde parecem faltar 6B e 63. Na realidade as instruções de código ED6B **** e ED63 **** existem, embora raramente sejam empregadas com este código: são, respectivamente, LD HL, (pq) e LD (pq), HL que normalmente são codificadas como 2A **** e 22 ****, que só ocupam três *bytes* e é mais rápido, além de compatível com os processadores 8Ø8Ø/8Ø85.

Mais nenhuma instrução tem esta forma.

Não há instruções que comecem com DDED ou FDED.

Assim, precisamos de uma tabela com um pequeno número de informação relativa a cada *byte*. Primeiro, as instruções que não comecem com DD, ED ou FD só podem ter um, dois ou três *bytes* de comprimento. Isto significa que, para guardar a informação necessária sobre o comprimento, só precisamos de dois *bits*. Segundo, as instruções que comecem com DD ou FD só podem ter dois, três ou quatro *bytes* de comprimento, ou ignorando DD ou FD, um, dois ou três *bytes*. Mais uma vez só precisamos de dois *bits*, para guardar o comprimento. Isto dá um total de quatro *bits* e podemos assim representar os comprimentos apropriados para cada *byte* por um só dígito hexadecimal ou *nibble*, como também se costuma dizer (um *nibble* é o mesmo que um *byte*, mas mais pequeno: tem quatro *bits* em vez de oito). Um *nibble* pode ser escrito num só dígito hexadecimal, ao contrário de um *byte*, que precisa de dois. O nosso programa fará então uso da seguinte tabela, chamada LENS e que, neste caso, se encontra a partir do endereço 68ØØ. Pode-se, claro está, guardá-lo em qualquer outro sítio da RAM, desde que todos os elementos da tabela tenham os endereços com a mesma parte alta.

```
LENS DEFB   5F 55 55 A5 55 55 55 A5
            AF 55 55 A5 A5 55 55 A5
            AF F5 55 A5 A5 F5 55 A5
            AF F5 99 E5 A5 F5 55 A5
            55 55 55 95 55 55 55 95
            55 55 55 95 55 55 55 95
            55 55 55 95 55 55 55 95
            99 99 99 59 55 55 55 95
            55 55 55 95 55 55 55 95
            55 55 55 95 55 55 55 95
            55 55 55 95 55 55 55 95
            55 55 55 95 55 55 55 95
            55 FF F5 A5 55 FE FF A5
            55 FA F5 A5 55 FA F5 A5
            55 F5 F5 A5 55 F5 FA A5
            55 F5 F5 A5 55 F5 F5 A5
```

Como se vê, são dezasseis filas, com dezasseis dígitos hex por fila. Às sequências de *bytes* começando por DD ou FD que não representem um *opcode* válido, como DDØØ, foi atribuído um número de *bytes* como se DD ou FD não estivessem lá.

O programa seguinte irá fazer uma «desassemblagem» produzindo, para cada instrução, uma sequência de *bytes* com o comprimento correcto. Parte-se do princípio de que a tabela LENS existe (e se encontra em 68ØØ) e também que dispomos de uma sub-rotina chamada H__PRINT para escrever o conteúdo do registo A em hexadecimal sem alterar nenhum dos outros registos. Esta é, na verdade, a sub-rotina utilizada anteriormente no livro, fazendo parte de H__LIST em HEXLD 3. Chamaremos a este novo programa H__LIST__2 e esta versão começa a partir do endereço 688Ø.

```
AF          H__LIST__2  XOR A
323C5C                  LD (TVFLAG),A       Dirige a impressão p. a
                                            parte sup. do écran.
2AF8FF (or 2AF87F)      LD HL,(ADDRESS)     HL: = endr. a partir do
                                            qual se inicia a listagem.
3EØD        NEXT        LD A,"enter"
D7                      RST 1Ø              Escreve a partir do início
                                            de nova linha.
```



```
CD7DFF (or CD7D7F)      CALL PR__HL         Escreve HL em hex.
ØE00                    LD C,ØØ             C é utilizado como flag
                                            que nos indica se a
                                            instrução começa com
                                            DD ou FD ou não.
7E                      LD A,(HL)           A: = início da próxima
                                            instrução.
FEDD                    CP DD
2804                    JR Z,DDFD
FEFD                    CP FD               A instrução começa
2006                    JR NZ,NORM          como DD ou FD?
            DDFD        $
CDDØFF (or CDDØ7F)      CALL H__PRINT       Escreve «DD» ou «FD».
23                      INC HL              Aponta o próximo byte.
ØC                      INC C               Muda o flag C conforme
                                            necessário.
7E                      LD A,(HL)           A: = próximo byte.
FEED        NORM        CP ED
2011                    JR NZ,SIMPLE        Byte começa por
                                            «ED»?
CDDØFF (or CDDØ7F)      CALL H__PRINT       Escreve o byte «ED».
23                      INC HL              Aponta p. o pr. byte
7E                      LD A,(HL)
E6C7                    AND C7              Este byte é da forma
                                            01***Ø11?
FE43                    CP 43
Ø601                    LD B,Ø1
201E                    JR
                        NZ,NXT__BYT
Ø603                    LD B,Ø3
181A                    JR NXT__BYT
E5          SIMPLE      PUSH HL
CB3F                    SRL A               Divide A por dois.
F5                      PUSH AF             Empilha o flag carry.
C6ØØ                    ADD A,LENS low
6F                      LD L,A
2668                    LD H,LENS high      HL aponta para o byte
                                            apropriado na tabela.
F1                      POP AF
7E                      LD A,(HL)           A: = byte da tabela.
3804                    JR C,NIBBLE         Usa o flag carry para
1F                      RRA                 decidir se vai utilizar o
```

```
1F                  RRA          primeiro ou o segundo
1F                  RRA          dígito hex.
1F                  RRA
ØD        NIBBLE    DEC C        Usa C p. decidir qual
                                 dos dois bytes usar.

2002                JR NZ,OK
1F                  RRA
1F                  RRA
E603      OK        AND Ø3       Guarda este número em
47                  LD B,A       B p. usar como count.
E1                  POP HL       Recupera o endereço do
                                 byte a ser
                                 «desassemblado».
7E        NXT_BYT   LD A,(HL)
23                  INC HL
CDDØFF (or CDDØ7F)  CALL H_PRINT Escreva pr. byte em hex.
1ØF9                DJNZ NXT_BYT
18B1                JR NEXT
```

Chegamos agora ao terceiro nível — e agora é verdadeiramente desassemblar. No entanto, não explicaremos o funcionamento pormenorizado de cada parte do programa; é conveniente cada qual descobri-lo por si próprio. No entanto, daremos a listagem do programa e o algoritmo pelo qual ele funciona.

DESASSEMBLAR

Eis pois o algoritmo que permitirá desassemblar qualquer código máquina em *assembler*. Isto é — mudar, por exemplo, 69 para LD L,C ou FDCB2A76 para BIT 6, (IY+2A). Uma maneira de o fazer seria guardar na memória uma tabela muito grande, tal como as incluídas nos apêndices deste livro; mas, se esta solução é perfeitamente aceitável, falta-lhe, no entanto, a elegância de um programa de computador bem concebido. Só a informação da tabela iria ocupar cerca de 4 K! Este algoritmo possibilita-nos escrever o nosso programa em código máquina num espaço significativamente menor — inferior a um K e um quarto, na realidade.

Neste algoritmo empregamos as seguintes convenções:

r(Ø) significa "B", r(1) significa "C", r(2) significa "D", r(3) significa "E", r(4) significa "H", r(5) significa "L", r(6) significa "X" e r(7) significa "A".

s(Ø) significa "BC", s(1) significa "DE", s(2) significa "Y" e s(3) significa "SP".

q(Ø) significa "BC", q(1) significa "DE", q(2) significa "Y" e q(3) significa "AF".

n(Ø) significa "Ø", n(1) significa "1", n(2) significa "2", n(3) significa "3", n(4) significa "4", n(5) significa "5", n(6) significa "6" e n(7) significa "7".

c(Ø) significa "NZ", c(1) significa "Z", c(2) significa "NC", c(3) significa "C", c(4) significa "PO", c(5) significa "PE", c(6) significa "P" e c(7) significa "M".

x(Ø) significa "ADD A,", x(1) significa "ADC A,", x(2) significa "SUB *espaço*", x(3) significa "SBC A,", X(4) significa "AND *espaço*", x(5) significa "XOR *espaço*", X(6) significa "OR *espaço*" e x(7) significa "CP *espaço*".

Definem-se duas variáveis chamadas CLASS e INDEX e fazem-se inicialmente iguais a zero.

Vamos escrever o *byte* que está a ser desassemblado em binário e dividi-lo em três partes: F, G e H. F consiste dos *bits* 7 e 6, G dos *bits* 5, 4 e 3 e H dos *bits* 2, 1 e Ø. Por exemplo, para desassemblar o *byte* 69 (binário Ø1ØØØ1Ø1) teríamos de parti-lo, assim, em três partes: Ø1/ØØ1/Ø1/. Neste caso F vale um, G vale cinco e H, um.

Em seguida dividimos G em duas partes, J e K, sendo J constituído pelos *bits* 2 e 1 e K apenas pelo *bit* Ø. Se G for 1Ø1 binário como acima, dividimo-lo do seguinte modo: 1Ø/1. Neste caso J seria igual a dois e K igual a um.

Reservemos uma área de memória que chamaremos DIS (o princípio da RAM livre é um bom lugar), para conter uma *string* de comprimento desconhecido. Para isto só temos de fazer com que o primeiro *byte* de DIS conte o número de *bytes* da *string* e começar o texto da *string* a partir do segundo *byte* de DIS (só necessita de um *byte*, já que DIS nunca terá um comprimento superior a FF

caracteres). DIS deve ser, inicialmente, uma *string* vazia, isto é, sem conter qualquer texto.

O algoritmo começa a partir daqui:

1) Se o *byte* a desassemblar é CB então...
   seja CLASS igual a um, e
   se INDEX for igual a zero, prosseguimos a partir do próximo *byte*,
   se INDEX for não zero, interpretemos os próximos dois *bytes* em ordem inversa.
2) Se o *byte* é ED, façamos CLASS igual a dois e continuemos a partir do *byte* seguinte.
3) Se o *byte* é DD, façamos INDEX igual a um e continuemos a partir do próximo *byte*.
4) Se o *byte* é FD, façamos INDEX igual a dois e continuemos a partir do próximo *byte*.

Se CLASS for igual a zero aplicamos o seguinte:

Se o byte é 76, a instrução completamente desassemblada é HALT.

Se F é igual a zero,...

Se H é igual a zero,...

Se G é maior que 3 fazemos DIS igual a JR c(G−4), v.

Se G é menor que 4 escolhemos o item n.º (G+1) desta lista: NOP/EX AF, AF'/DJNZ v/JR v.

Se H é igual a um,...

Se K é zero, fazemos DIS igual a LD s(J),w.

Se K é um fazemos DIS igual a ADD y,s(J).

Se H é igual a dois, fazemos DIS igual a LD seguido do item n.º (G+1) desta lista:
(BC),A/A,(BC)/(DE),A/A,(DE)/(w),y/y,(w)/(w), A/A,(w).

Se H é igual a três, ...

Se K é igual a zero, fazemos DIS igual a INC s(J).

Se K é igual a um, fazemos DIS igual a DEC s(J).

Se H é igual a quatro, fazemos DIS igual a INC r(G).

Se H é igual a cinco, fazemos DIS igual a DEC r(G).

Se H é igual a seis, fazemos DIS igual a LD r(G), v.

Se H é igual a sete, escolhemos o item n.º (G+1) desta lista:
RLCA/RRCA/RLA/RRA/DAA/CPL/SCF/CCF.

Se F é igual a um, fazemos DIS igual a LD r(G),r(H).

Se F é igual a dois, fazemos DIS igual a x(G) seguido imediatamente de r(H).

Se F é igual a três, ...

Se H é igual a zero, fazemos DIS igual a RET c(G).

Se H é igual a um, ...

Se K é igual a zero, fazemos DIS igual a POP q(J).

Se K é igual a um, fazemos DIS igual ao item n.º (J+1) desta lista:
RET/EXX/JP (y)/LD SP,y.

Se H é igual a dois, fazemos DIS igual a JP c(G),w.

Se H é igual a três escolhemos o item n.º (G+1) desta lista:
JP w/*/OUT (v),A/IN A,(v)/EX (SP),y/EX DE,HL/DI/EI.

Se H é igual a quatro, fazemos DIS igual a CALL c(G), w.

Se H é igual a cinco, ...

Se K é zero, fazemos DIS igual a PUSH q(J).

Se K é um, fazemos DIS igual a CALL w.

Se H é igual a seis fazemos DIS igual a x(G) seguido de v.

Se H é igual a sete, fazemos DIS igual a RST seguido do item n.º (G+1) desta lista:
Ø0/Ø8 v/1Ø/18/2Ø/28/3Ø/38.
Porque no *Spectrum,* RST Ø8 utiliza o *byte* seguinte.

—

Se CLASS for igual a um, aplica-se o seguinte:

Se F é igual a zero, escolhemos o item n.º (G+1) desta lista:
RLC/RRC/RR/SLA/SRA/*/SRL e seguido de r(H).

Se F é igual a um, fazemos DIS igual a BIT n(G),r(H).

Se F é igual a dois, fazemos DIS igual a RES n(G),r(H).

Se F é igual a três, fazemos DIS igual a SET n(G),r(H).

Se CLASS for igual a dois, aplica-se o seguinte:

F nunca pode ser igual a zero.
Se F é igual a um...

Se H é igual a zero, fazemos DIS igual a IN r(G),(C).

Se H é igual a um, fazemos DIS igual a OUT (C),r(G).

Se H é igual a dois,...

Se K é igual a zero, fazemos DIS igual a SBC HL,s(J).
Se K é igual a um, fazemos DIS igual a ADC HL,s(J).

Se H é igual a três,...
Se K é igual a zero, fazemos DIS igual a LD (w),s(J).
Se K é igual a um, fazemos DIS igual a LD s(J),(w).
Se H é igual a quatro, fazemos DIS igual a NEG.

Se H é igual a cinco,...
Se K é igual a zero, fazemos DIS igual a RETN
Se K é igual a um, fazemos DIS igual a RETI

Se H é igual a seis, escolhemos o item n.º (G+1) desta lista:
IM O*IM 1/IM 2.

Se H é igual a sete, escolhemos o item n.º (G+1) desta lista:
LD I,A/LD R,A/LD A,I/LD A,R/RRD/RLD/*/*.
Se F é igual a dois escolheos o item n.º (H+1) desta lista.
LD/cp/in/ot seguido do item n.º (G-3) desta lista:
I/D/IR/DR.
F não pode ser igual a três.

Para achar o resultado final

Substituímos todos os x por:
(HL) se INDEX for igual a zero.
(IX+v) se INDEX for igual a um.
(IY+v) se INDEX for igual a dois.

Substituímos todos os y por:
HL se INDEX for igual a zero.
IX se INDEX for igual a um.
IY se INDEX for igual a dois.

Substituímos todos os v pelo *byte* que vai ser desassemblado, a seguir, em hex.

Substituímos todos os w pelos dois *bytes* que vão ser desassemblados, a seguir em hex e em ordem inversa.

A *string* DIS contém agora a instrução correctamente desassemblada que vai ser escrita no *écran*.

Veremos mais adiante um programa em linguagem máquina que desassembla código empregando este algoritmo. Este programa ocupará menos de um K e um quarto da memória. Apesar de isto parecer surpreendente, devemos acrescentar que este programa, embora possível, é muito complicado e recorre a um número razoável de técnicas de programação completamente novas.

O programa é constituído por oito sub-rotinas diferentes, ligadas por uma sub-rotina MASTER que chama cada uma das outras na ordem necessária. Isto consegue-se da maneira a seguir descrita.

Em alguma parte do programa deve encontrar-se uma tabela chamada SUBRTS com oito endereços — os das oito sub-rotinas que executam o programa. O par de registos HL' (note bem: HL') apontará uma sequência de dados que a sub-rotina principal MASTER interpreta como a ordem pela qual as outras sub-rotinas devem ser chamadas. Os dados desta sequência terminam por um item com o *bit* sete *set*. Os *bits* dois, um e zero indicam exactamente qual a sub-rotina a chamar. Assim, por exemplo, qualquer *byte* da forma binária ***** 11Ø irá chamar a sub-rotina seis e qualquer *byte* da forma binária de *****Ø11 irá chamar a sub-rotina três. Um *byte* da forma binária 1 *** *Ø11 não só irá chamar a sub-rotina três como informar também a sub-rotina MASTER de que esta é última sub-rotina a chamar.

Assim, qualquer item de dados nesta sequência tem este aspecto: Ø *** * nnn, para todos menos o último item, que será 1 *** * nnn (a parte escrita como nnn significa o número apropriado — de zero a sete, como foi descrito). Algumas destas sub-rotinas exigirão, também, outros dados (que nunca precisarão ocupar mais de três *bits*). Se representarmos esta informação adicional em binário como «ddd», podemos poupar espaço, guardando-a entre alguns dos *bits*, até agora não utilizados, do *byte* de controle, de modo que ele fique com este aspecto: Ø * dd dnnn para todos os *bytes* excepto o último,

que será 1 * dd dnnn. Ao *bit* seis, que ainda não foi utilizado, atribuímos um sentido à parte — se for um, isto significa que deve ser colocada uma vírgula no fim de DIS, depois de ter sido executada a sub-rotina corrente.

Atendendo à pouca clareza do raciocínio que nos guiou, e para o tornar mais claro, vamos supor que HL' indica um endereço no qual está guardada a sequência de *bytes* Ø4 44 25 85. Isto significa, antes do mais, que a sub-rotina quatro deve ser chamada com a informação adicional ØØØb, em seguida a sub-rotina quatro deve ser chamada de novo (também com a informação ØØØb), mas desta vez com a indicação de que, depois de executada, deve ser acrescentada uma vírgula ao fim de DIS, depois a sub-rotina cinco deve ser chamada mais uma vez, mas agora com ØØb.

A sub-rotina MASTER que realiza tudo isto é a seguinte:

```
D9      MASTER  EXX
7E              LD A,(HL)       Lê o byte de dados e
23              INC HL          incrementa o ponteiro.
D9              EXX
5F              LD E,A          E: = este byte (os bits 7
                                a 3 serão necs. depois).
E607            AND Ø7          A: = n.º da sub-rotina a
                                ser chamada.
17              RLA             Multiplica por dois.
4F              LD C,A
0600            LD B,ØØ         BC: = número da
                                sub-rotina vezes dois.
21????          LD HL,RETURN    HL: = endr. de retorno
                                das sub-rotinas «menores».
E5              PUSH HL         Empilha este endr.
21????          LD HL,SUBRTS
09              ADD HL.BC       HL aponta p. o endr. da
                                sub-rotina a chamar.
4E              LD C,(HL)
23              INC HL
46              LD B,(HL)       BC: = endr. a chamar.
C5              PUSH BC
C9              RET             Chama esta sub-rotina.
```


Podemos aprender muita coisa se estudarmos esta sub-rotina MASTER. Como se chama a sub-rotina apropriada (escolhida de entre oito)? Primeiro, colocamos na pilha o endereço da etiqueta RETURN. Isto significa que se cada uma das sub-rotinas terminar com RET, o controle nessa altura irá saltar para a etiqueta RETURN. Estando o endereço de retorno na pilha, basta agora saltar para o endereço da rotina requerida. É o que faz a segunda parte de MASTER.

O endereço da rotina a chamar é colocado no topo da pilha, por cima do endereço da etiqueta RETURN. Executa-se então uma instrução RET. RET tem o efeito de retirar o endereço no topo da pilha, efectuando um salto para esse endereço. O valor agora no topo da pilha é o endereço da etiqueta RETURN, o que significa que, quando a sub-rotina terminar, com RET, o controle retornará correctamente. Tudo isto é necessário porque não existe a instrução CALL (BC): no BASIC do *Spectrum*, a instrução GO SUB variável é permitida, mas não em código máquina. Outra maneira de obtermos o mesmo resultado que com PUSH BC/RET é (este é mais rápido) LD H,B/LD L,C/JP (HL).

Todos os registos alternativos (excepto A') têm, neste programa, uma utilização específica. Eis as suas funções:

BC' guarda o endereço do próximo *byte* a ser desassemblado.
D' guarda a variável INDEX.
E' guarda a variável CLASS.
HL' aponta para o *byte* com as informações sobre a próxima sub-rotina.

O *byte* a desassemblar é lido e guardado no registo D por meio da sequência EXX/LD A,(BC)/INC BC/EXX/LD D,A. A partir daqui, as quantidades às quais demos os nomes de F, G e H podem ser calculadas. Em alguma parte deste programa encontra-se uma tabela chamada DATADS contendo doze endereços diferentes que correspondem aos casos CLASS=Ø/F=Ø,1,2 ou 3, CLASS=1/F=Ø1,2 ou 3 e CLASS=2/F=Ø,1,2 ou 3. O valor inicial de HL' é simplesmente lido a partir desta tabela.

HL' também pode ser alterado pelas próprias sub-rotinas. Por exemplo, a função da sub-rotina zero é redefinir HL' como igual ao endereço número (H+1) numa tabela de oito endereços. Não confundir este último H com o registo H.

Serão também necessárias várias outras sub-rotinas juntamente com as oito de controle. Uma dessas rotinas de grande importância é uma sub-rotina para acrescentar um só carácter ao fim da *string* DIS, estabelecendo a convenção de que a *string* começa no endereço (STKEND) (o início da RAM livre) e que o primeiro *byte* é o comprimento da *string*. Esta pode ser inicializada pela sequência LD HL,(STKEND)/LD(HL),ØØ. Para juntar um carácter (guardado no registo A) a sub-rotina requerida é:

F5	CHR	PUSH AF	Guarda os registos A, B,
C5		PUSH BC	C, H e L para não serem
E5		PUSH HL	corrompidos por esta sub-rotina.
E67F		AND 7F	Deixa de parte o *bit* sete.
2A655C		LD HL,(STKEND)	HL aponta a *string* DIS.
4E		LD C,(HL)	C: = cm. anterior de DIS.
ØC		INC C	C: = novo cm. de DIS.
71		LD (HL),C	Guarda novo cm. de DIS.
ØGØØ		LD B,ØØ	BC: = cm. de DIS.
Ø9		ADD HL,BC	HL aponta p. o novo últ. *byte* de DIS.
77		LD (HL),A	Guarda o cr. em DIS.
E1		POP HL	Restaura todos os
C1		POP BC	registos.
F1		POP AF	
C9		RET	Fim da sub-rotina.

A figura 17.2 é um diagrama que mostra mais ou menos o que se passa. As oito sub-rotinas necessárias para este programa de desassemblar são as que se seguem:

SUB-ROTINA ZERO – SPLIT

Esta é a sub-rotina chamada pelo *byte* ØØ. Se for alguma vez utilizada, será então o *byte* ØØ o primeiro da sequência. A seguir ao *byte* ØØ encontram-se oito novos endereços. Situados nestes endereços encontram-se oito diferentes sequências de dados. Entre estes endereços a sub-rotina selecciona o de ordem (H+1) e lê os dados subsequentes a partir da nova posição.

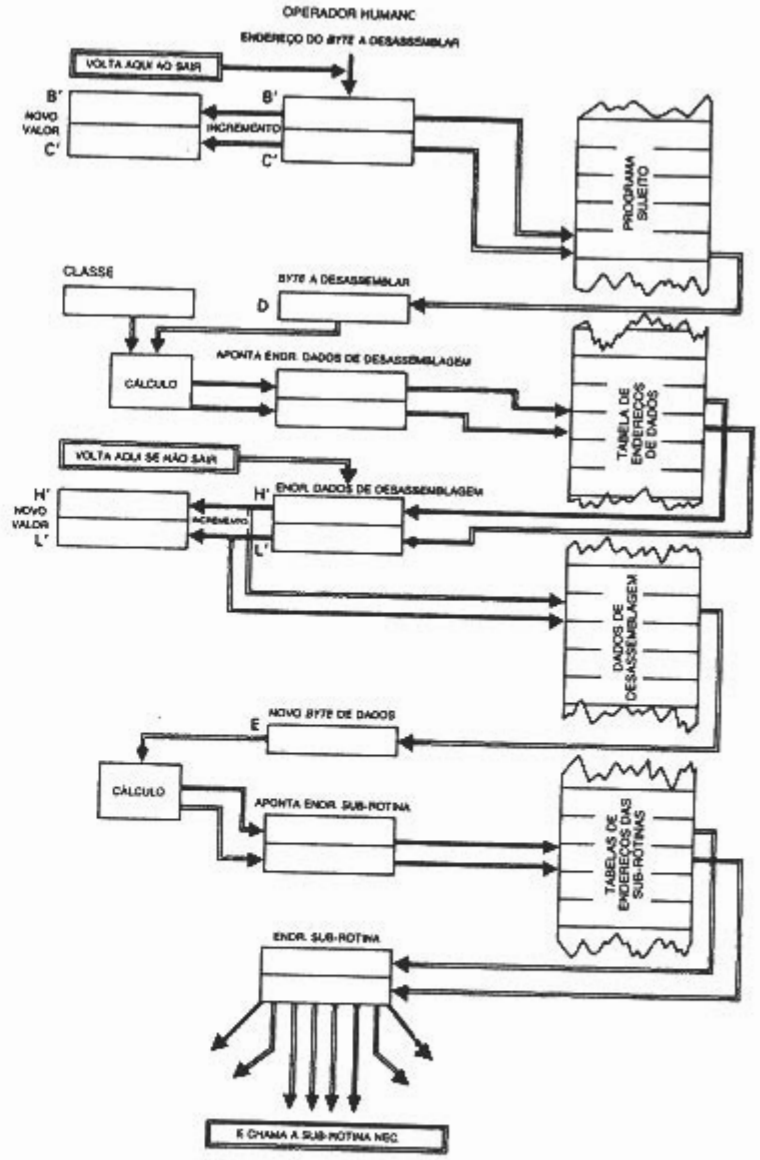


Figura 17.1

SUB ROTINA UM – LITERAL

O *byte* de controle é seguido de uma série de caracteres, tais como «N», «O' e «P» que representam parte ou toda a instrução desassemblada. O último carácter desta *string* deve obrigatoriamente ter o *bit* sete *set*. Esta sub-rotina deve detectar este *bit* e utilizá-lo como uma indicação de fim da string. A informação suplementar contida no *byte* de controle (isto é, os *bits* 5, 4 e 3) deve ser ou ØØØ ou ØØ1. Se for ØØ1, significa que a *string* dada deve ser seguida de um espaço. Talvez fosse preferível, em vez de um espaço, acrescentar o número necessário para que os operandos das várias instruções comecem todos na mesma coluna. Como os *opcodes* mais curtos têm duas letras e os mais longos, quatro, isto significa que o número de espaços a acrescentar seria um, dois ou três. (Ou estes números mais um número fixo de espaços.) O número exacto de espaços a acrescentar poderia ser determinado facilmente com base no comprimento de DIS (contido no seu primeiro *byte*): seria igual a cinco menos esse comprimento. A *string* que é especificada por este meio deve ser acrescentada ao fim de DIS.

SUB-ROTINA DOIS – LIST-G

A sua função é seleccionar o item número (G+1) da lista seguinte. O dado suplementar para esta sub-rotina (*bits*, 5, 4 e 3) deve ser Ø11 se houver quatro items na lista, ou 111 se houver oito items na lista. Por exemplo, o *byte* de controle 3A (binário Ø/Ø111/Ø1Ø) significa «chame a sub-rotina dois para seleccionar o item da ordem (G+1) da seguinte lista de oito». A lista pode, por exemplo, ser RLCA/RRCA/RLA/RRA/DAA/CPL/SCF/CCF. O último *byte* de cada palavra deve ter o *bit* sete *set* para que a sub-rotina saiba onde termina uma palavra e começa outra. Assim, no exemplo dado, se G fosse cinco, a palavra CPL seria acrescentada à *string* DIS. O próximo *byte* a interpretar pela sub-rotina MASTER deverá ser o primeiro *byte* a seguir ao «F» de «cc».

SUB-ROTINA TRÊS – LIST-H

Semelhante à sub-rotina dois, mas utilizando H em vez de G.

SUB-ROTINA QUATRO – SELECT-G

A sua função depende dos *bits* 5, 4 e 3 do *byte* de controle:

ØØØ significa seleccione r(G).
ØØ1 significa seleccione s(G).
Ø1Ø significa seleccione q(G).
Ø11 significa seleccione n(G).
1ØØ significa seleccione c(G).
1Ø1 não é utilizado.
11Ø significa seleccione x(G).
111 não é utilizado.

O item seleccionado deve ser colocado no fim de DIS.

SUB-ROTINA CINCO – SELECT-H

Como na rotina quatro mas utilizando H em vez de G.

SUB-ROTINA SEIS – SKIP

Torna zero o *bit* cinco de E (o *byte* a desassemblar). Se o valor anterior do *bit* cinco era um, salte sobre n *bytes* de dados. O número «n» é o valor do *byte* imediatamente a seguir. Se o *bit* cinco era anteriormente zero, este *byte* imediatamente a seguir e que serve apenas para especificar n é ignorado e o *byte* que se lhe segue é interpretado como o próximo *byte* de controle.

SUB-ROTINA SETE – KSKIP

Substituímos o *bit* três de E pelo *bit* quatro; substituímos o *bit* quatro pelo *bit* cinco e fazemos o *bit* cinco igual a zero. O efeito disto foi o mesmo que LET G=J. Então, se o valor anterior do *bit* três era um, saltamos sobre n *bytes*, tal como na sub-rotina seis. Esta sub-rotina pode ser interpretada como IF K=Ø THEN.../IF K=1 THEN...

O PROGRAMA COMO UM TODO

O programa completo para desassemblar consiste na inicialização das variáveis CLASS e INDEX, atribuir a BC' o valor do endereço

inicial a desassemblar (introduzido pelo operador humano), encontrar o endereço HL' a partir das tabelas e entrar então na rotina MASTER. Ao sair desta, deve-se substituir todos os x e y e depois todos os v e w como foi estabelecido anteriormente neste capítulo. O resultado calculado pode agora ser escrito e o próximo *byte* a desassemblar será tratado exactamente do mesmo modo. O programa completo, incluindo todas as sub-rotinas necessárias e todos os dados, ocupa um pouco menos de um K e um quarto da memória.

Por muito bem que este processo tenha sido explicado, tentar interpretar o modo como ele funciona é um desafio. Este programa foi, desta vez, listado deliberadamente sem comentários, de modo a que quem quiser tirar todo o benefício da listagem terá de interpretá-la por si próprio.

Só para provar que o programa realmente funciona ele foi, na verdade, empregado para produzir uma listagem de si próprio. Pode-se ver, portanto, uma amostra do seu funcionamento. O BASIC que deve acompanhar o código máquina consiste apenas em fazer POKE no endereço (ADDRESS) — (7FF8) nesta listagem — do endereço a partir do qual se quer desassemblar e em seguida chamar o código máquina no endereço 69B5 (27061d).

Talvez seja até boa ideia acrescentar este programa permanentemente a HEXLD 3. De qualquer modo o trabalho está todo feito — e é necessário procurar compreendê-lo...

Boa sorte.

6800 TYPES	Dados para as sub-rotinas quatro e cinco.
6854 SUBRTS	Endereços das oito sub-rotinas de controle.
6864 DATADS	Endereços das sequências de dados.
687C REPLACE	Sub-rotina para substituir x, y, v e w se nec.
68B4 CHR	Sub-rotina para acrescentar um carácter à *string* DIS.
68C7 HP__BC	Escreve BC em hex.
68CC HP__A	Escreve A em hex.
68D5 HP__AL	Escreve o segundo dígito de A em hex.
68DF PRINT__A	Escreve um carácter ASCII no *écran*.
68E9 INS	Insere novo *byte* do programa sujeito em DIS.
6903 RETURN	Endr. de retorno das sub-rotinas de controle.

6911 DECODE	Descodifica DIS e escreve o resultado final.
69B5 START	Chama a partir deste endereço.
69BF RESTART	Recomeça p. desassemblar a pr. instrução.
69C9 MAIN	Rotina principal.
6A33 MASTER	Selecciona e chama a rotina de controle.
6A4B SPLIT	Sub-rotina Zero.
6A5C LITERAL	Sub-rotina Um.
6A6E FIND	Selecciona item n.° (A+1) da seguinte lista.
6A72 SELECT	Selecciona item n.° (A+1) de qualquer lista.
6A7E LIST-G	Sub-rotina Dois.
6A84 LIST-H	Sub-rotina Três.
6AA1 SELECT-G	Sub-rotina Quatro.
6AA7 SELECT-H	Sub-rotina Cinco.
6ABE KSKIP	Sub-rotina Sete.
6ACE SKIP	Sub-rotina Seis.
6AE2 DATA	Tds. dados req. p. as sub-rotinas de controle.

Os seguintes endereços contêm dados em vez de código máquina:

6800 – 687B
6925 – 6929
6932 – 6938
693E – 6944
6950 – 6952
695B – 695D
6963 – 6965
6975 – 6977
6997 – 699B
69F1 – 69F4
69E2 – 6CE6

```
687C  POP  IX
687E  LD  DE,(5C65)
6882  AND  A
6883  SBC  HL,DE
6885  LD  A,(DE)
6886  SUB  L
6887  INC  A
6888  PUSH  AF
```

```
6889 LD A,(DE)
688A LD L,A
688B ADD A,(IX+00)
688E DEC A
688F LD (DE),A
6890 ADD HL,DE
6891 LD E,(IX+0C)
6894 LD D,00
6896 PUSH HL
6897 ADD HL,DE
6898 DEC HL
6899 EX DE,HL
689A POP HL
6898 POP AF
689C PUSH BC
689D LD C,A
689E LD B,00
68A0 LDDR
68A2 EX DE,HL
68A3 INC DE
68A4 LD C,(IX+00)
68A7 PUSH IX
68A9 POP HL
68AA INC HL
68AB LD B,00
68AD LDIR
68AF EX DE,HL
68B0 DEC HL
68B1 POP BC
68B2 PUSH DE
68B3 RET
68B4 PUSH AF
68B5 PUSH BC
68B6 PUSH HL
68B7 AND 7F
68B9 LD HL,(5C68)
68BC LD C,(HL)
68BD INC C
68BE LD (HL),C
68BF LD B,00
68C1 ADD HL,BC
68C2 LD (HL),A
68C3 POP HL
68C4 POP BC
68C5 POP AF
68C6 RET
68C7 LD A,B
68C8 CALL 68CC
68CB LD A,C
68CC PUSH AF
68CD RRA
68CE RRA
```



```
68CF RRA
68D0 RRA
68D1 CALL 68D5
68D4 POP AF
68D5 AND 0F
68D7 OR 30
68D9 CP 3A
68DB JR C,02
68DD ADD A,07
68DF PUSH BC
68E0 PUSH DE
68E1 PUSH HL
68E2 EXX
68E3 RST 10
68E4 EXX
68E5 POP HL
68E6 POP DE
68E7 POP BC
68E8 RET
68E9 EXX
68EA LD A,(BC)
68EB INC BC
68EC EXX
68ED PUSH AF
68EE CALL 68F6
68F1 POP AF
68F2 RRA
68F3 RRA
68F4 RRA
68F5 RRA
68F6 AND 0F
68F8 OR 30
68FA CP 3A
68FC JR C,02
68FE ADD A,07
6900 LD (HL),A
6901 DEC HL
6902 RET
6903 BIT 6,E
6905 JR Z,0B
6907 LD A,20
6909 CALL 6884
690C BIT 7,E
690E JP Z,6A33
6911 EXX
6912 LD A,D
6913 EXX
6914 LD C,A
6915 LD HL,(5C65)
6918 LD B,(HL)
6919 INC HL
691A LD A,(HL)
```

```
691B AND A
691C JR NZ,27
691E LD A,C
691F AND A
6920 JR NZ,0A
6922 CALL 687C

692A JR 1E
692C DEC H
692D JR NZ,0C
692F CALL 687C

.+39 JR 0A
6938 CALL 687C

6945 LD A,(HL)
6946 DEC A
6947 JR NZ,1D
6949 LD A,C
694A AND A
694B JR NZ,08
6940 CALL 687C

6953 JR 11
6955 DEC D
6956 JR NZ,05
6958 CALL 687C

695E JR 05
6960 CALL 687C

6966 DJNZ B1
6968 LD HL,(5C65)
6968 LD B,(HL)
696C INC HL
696D LD A,(HL)
696E SUB 02
6970 JR NZ,1F
6972 CALL 687C

6978 EXX
6979 LD A,E
697A RRA
697B SBC A,A
697C AND D
697D EXX
697E PUSH AF
697F JR Z,04
6981 EXX
6982 DEC BC
```



```
6983 DEC BC
6984 EXX
6985 CALL 68E9
6988 POP AF
6989 JR Z,19
698B EXX
698C INC BC
698D NOP
698E EXX
698F JR 13
6991 DEC A
6992 JR NZ,12
6994 CALL 687C

699C CALL 68E9
699F CALL 68E9
69A2 INC HL
69A3 INC HL
69A4 INC HL
69A5 INC HL
69A6 DJNZ C4
69A8 LD HL,(5C65)
69AB LD B,(HL)
69AC INC HL
69AD LD A,(HL)
69AE CALL 68DF
69B1 DJNZ F9
69B3 JR 0A
69B5 XOR A
69B6 LD (5C3C),A
69B9 EXX
69BA LD BC,(7FF8)
69BE EXX
69BF LD HL,(5C65)
69C2 LD (HL),00
69C4 EXX
69C5 LD DE,0000
69C8 EXX
69C9 LD A,0D
69CB CALL 68DF
69CE EXX
69CF CALL 68C7
69D2 LD A,20
69D4 CALL 68DF
69D7 LD A,(BC)
69D8 INC BC
69D9 PUSH DE
69DA EXX
69DB POP BC
69DC CP 76
69DE JR NZ,16
69E0 DEC C
```

```
69E1 JR  Z,12
69E3 LD  HL,69F1
69E6 LD  B,04
69E8 LD  A,(HL)
69E9 INC HL
69EA CALL 680F
69ED DJNZ F9
69EF JR  D8

69F5 INC C
69F6 CP  C8
69F8 JR  Z,18
69FA CP  E0
69FC JR  Z,10
69FE CP  D0
6A00 JR  Z,08
6A02 CP  F0
6A04 JR  NZ,1A
6A06 LD  B,02
6A08 JR  11
6A0A LD  B,01
6A0C JR  0D
6A0E LD  C,02
6A10 JR  09
6A12 LD  C,01
6A14 INC  B
6A15 DEC  B
6A16 JR  Z,03
6A18 EXX
6A19 INC  BC
6A1A EXX
6A1B PUSH BC
6A1C EXX
6A1D POP  DE
6A1E JR  B7
6A20 LD  D,A
6A21 AND  C0
6A23 OR  C
6A24 RLCA
6A25 RLCA
6A26 RLCA
6A27 ADD  A,64
6A29 LD  L,A
6A2A LD  H,68
6A2C LD  C,(HL)
6A2D INC  HL
6A2E LD  B,(HL)
6A2F PUSH BC
6A30 EXX
6A31 POP  HL
6A32 EXX
6A33 EXX
```



```
6A34 LD  A,(HL)
6A35 INC  HL
6A36 EXX
6A37 LD  E,A
6A38 AND  07
6A3A RLA
6A3B LD  C,A
6A3C LD  B,00
6A3E LD  HL,6900
6A41 PUSH HL
6A42 LD  HL,6854
6A45 ADD  HL,BC
6A46 LD  C,(HL)
6A47 INC  HL
6A48 LD  B,(HL)
6A49 PUSH BC
6A4A RET
6A4B LD  A,D
6A4C EXX
6A4D AND  07
6A4F RLA
6A50 PUSH DE
6A51 LD  E,A
6A52 LD  D,00
6A54 ADD  HL,DE
6A55 LD  E,(HL)
6A56 INC  HL
6A57 LD  D,(HL)
6A58 EX  DE,HL
6A59 POP  DE
6A5A EXX
6A5B RET
6A5C EXX
6A5D LD  A,(HL)
6A5E INC  HL
6A5F CALL 6854
6A62 RLA
6A63 JR  NC,F8
6A65 EXX
6A66 BIT  3,E
6A68 RET  Z
6A69 LD  A,20
6A6B JP  6854
6A6E EXX
6A6F PUSH HL
6A70 EXX
6A71 POP  HL
6A72 LD  B,A
6A73 INC  B
6A74 DEC  B
6A75 RET  Z
6A76 LD  A,(HL)
```

```
6A77 INC HL
6A78 RLA
6A79 JR NC,F8
6A7B DJNZ F9
6A7D RET
6A7E LD A,D
6A7F RRA
6A80 RRA
6A81 RRA
6A82 JR 01
6A84 LD A,D
6A85 AND 07
6A87 CALL 6A8E
6A8A LD A,(HL)
6A8B INC HL
6A8C CALL 8884
6A8F RLA
6A90 JR NC,F8
6A92 LD A,E
6A93 RRA
6A94 RRA
6A95 RRA
6A96 AND 07
6A98 INC A
6A99 CALL 6A8E
6A9C PUSH HL
6A9D EXX
6A9E POP HL
6A9F EXX
6AA0 RET
6AA1 LD A,D
6AA2 RRA
6AA3 RRA
6AA4 RRA
6AA5 JR 01
6AA7 LD A,D
6AA8 AND 07
6AAA PUSH AF
6AAB LD A,E
6AAC AND 38
6AAE LD L,A
6AAF LD H,60
6AB1 POP AF
6AB2 CALL 6A72
6AB5 LD A,(HL)
6AB6 INC HL
6AB7 CALL 8884
6ABA RLA
6ABB JR NC,F8
6ABD RET
6ABE LD A,D
6ABF RRA
```



```
6AC0 AND 18
6AC2 LD B,A
6AC3 LD A,D
6AC4 RLA
6AC5 RLA
6AC6 AND 20
6AC8 OR B
6AC9 XOR D
6ACA AND 38
6ACC XOR D
6ACD LD D,A
6ACE BIT 5,D
6AD0 JR NZ,04
6AD2 EXX
6AD3 INC HL
6AD4 EXX
6AD5 RET
6AD6 RES 5,D
6AD8 EXX
6AD9 PUSH BC
6ADA LD C,(HL)
6ADB INC HL
6ADC LD B,00
6ADE ADD HL,BC
6ADF POP BC
6AE0 EXX
6AE1 RET
```

```
6CC0  A0  1B  40  C4  43  D0  49  CE
6CD0  4F  D4  BA  A0  A0  A0  A0  C9
6CE0  C4  49  D2  44  C2  09  49  4E
6CF0  C3  C3  84  D3  44  45  C3  84
6D00  00  00  00  00  00  00  00  00
6D10  00  00  00  00  00  00  00  00
```


CAPÍTULO DEZOITO

Aritmética em vírgula flutuante

A aritmética na ROM do *Spectrum* divide-se em duas categorias: inteira e em vírgula flutuante. Ambas têm um ponto em comum — os números nas duas categorias são representados em cinco *bytes*.

A representação em cinco *bytes* dos inteiros é muito fácil de compreender.

Eis uma lista das representações de cinco *bytes* dos inteiros de zero a quatro; facilmente se descobrirá um padrão:

Decimal	*Forma de cinco* bytes
0	00 00 00 00 00
1	00 00 01 00 00
2	00 00 02 00 00
3	00 00 03 00 00
4	00 00 04 00 00

E os negativos?

Decimal	*Forma de cinco* bytes
−1	00 FF FF FF 00
−2	00 FF FE FF 00
−3	00 FF FD FF 00
−4	00 FF FC FF 00

Esperemos que se tenha compreendido como as coisas se passam. Vamos experimentar agora com números maiores.

Decimal	*Forma de cinco* bytes	
1000	00 00 E8 03 00	(Note que 1000d = 03E8h).
−1000	00 FF 18 FC 00	(Note que −1000d = FC18h).

Por outras palavras, a regra é a seguinte: se o número for positivo, entre 0 e 65535d, representado em hexadecimal por mmnn, a sua representação em cinco *bytes* será 00 00 nn mm 00. Se

o número for negativo, entre −1 e −65535d, adiciona-se-lhe 65536d e, sendo o resultado mmnn, em hex, a representação em cinco *bytes* será: Ø0 FF nn mm ØØ.

Os números não inteiros, por outro lado, são um problema totalmente diferente. Números como Ø.5 ou PI como guardá-los em memória? Pensamos que a compreensão do modo pelo qual a representação de cinco *bytes* é construída irá ajudar na compreensão da parte que se segue. Vamos ver como se transformam números decimais em "números *Sinclair*".

Primeiro, é necessário considerar o tamanho de um desses números. O maior que pode ser guardado na forma de cinco *bytes* é 1.7Ø14118E38. Isto é:
17Ø,141,18Ø,ØØØ,ØØØ,ØØØ,ØØØ,Ø2Ø,ØØØ,ØØØ,ØØØ, ØØØ,ØØØ.
O menor número é, claro está, menos essa quantidade. Nenhum número além desse limite pode ser trabalhado pelo *Spectrum*.

Agora que já nos assegurámos de que o número que desejamos converter está dentro desses limites, devemos seguir uma sequência de passos. Vamos dar a este número um nome — X.

O primeiro passo é ignorar o sinal do número (isto é, LET X1=ABS X) e calcular então a quantidade chamada EXPOENTE, pela seguinte fórmula:

LET EXPOENTE=1+INT (LOG×1/LOG 2).

O primeiro *byte* da representação de cinco *bytes* é 8Øh mais este expoente. Demonstrá-lo-emos com alguns exemplos:

Número	*Valor absoluto*	*Expoente*	*Forma de cinco* bytes
X	X1 = ABS X	1 + INT(LOG X1 / LOG 2)	
Ø.375	Ø.375	−1	7F
PI	3.1415927	2	82
−1.5E2Ø	1.5E2Ø	68	C4

O segundo passo é computar uma quantidade chamada MANTISSA, por meio da seguinte fórmula:

LET MANTISSA=X1/2 ↑EXPOENTE

Número	*Expoente*	*Mantissa*
Ø.375	−1	Ø.75
PI	2	Ø.7853982
−1.5E2Ø	68	Ø.5Ø82198

Escondidos nalguma parte desta mantissa estão os últimos quatro *bytes* da representação *Sinclair* e portanto, precisamos de os extrair daí de qualquer modo. A melhor maneira de compreender isto é imaginar uma rotina BASIC que faça o seguinte:

```
LET A = MANTISSA
FOR I = 1 TO 4
LET A = 256*A
O próximo byte da forma de cinco bytes é INT A escrito em hex.
LET A = A - INT A
NEXT I
```

As variáveis A e I que aparecem no programa anterior não chegam a ter existência real — são variáveis mudas, destinadas apenas a ilustrar o que se passa. No entanto o valor final de A será utilizado em breve, de maneira que é melhor não o esquecer ainda. Vejamos como este processo afecta os nossos exemplos:

Número	*Forma de cinco* bytes					*«A» final*
Ø.375	7F	CØ	ØØ	ØØ	2Ø	Ø
PI	82	C9	ØF	DA	A1	Ø.6314368
−1.5E2Ø	C4	82	1A	BØ	D4	Ø.2842112

O próximo passo é chamado "arredondamento" e a sua acção é a seguinte: se o valor final de A for maior ou igual a zero ponto cinco, então deve incrementar, como um todo, os últimos quatro *bytes* da representação de cinco *bytes* (por exemplo ØØ ØØ ØØ 29 incrementa para ØØ ØØ ØØ 2A, 23 29 FF FF incrementa para 23 2A ØØ ØØ e assim por diante). Há no entanto um ponto com o qual se deve ter cuidado. Se os quatro *bytes* começarem como FF FF FF FF, não devem ser "incrementados" para ØØ ØØ ØØ ØØ — em vez disso estes *bytes* devem transformar-se, na realidade, em 8Ø ØØ ØØ ØØ, sendo incrementado o primeiro *byte* da representação de cinco *bytes*. Os nossos exemplos ficam arredondados como se segue:

Número	*Forma de cinco* bytes				
Ø.375	7F	CØ	ØØ	ØØ	ØØ
PI	82	C9	ØF	DA	A2
−1.5E2Ø	C4	82	1A	BØ	D4

O último passo é reintroduzir o sinal do número X inicial. A regra é muito simples: se X é positivo, subtraímos 8Øh ao segundo *byte* e se X é negativo, não fazemos nada.

Número	*Forma de cinco* bytes				
0.375	7F	4Ø	ØØ	ØØ	ØØ
PI	82	49	ØF	DA	A2
−1.5E2Ø	C4	82	1A	BØ	D4

O processo está agora completo. Esta é a forma final da representação de cinco *bytes*. Por razões óbvias, o maior expoente que podemos ter é FF e, assim, o maior número que pode ser guardado é FF 7F FF FF FF. Isto é, como já foi dito, 1.7Ø14118E38. O E na notação significa "com o ponto decimal deslocado (neste caso) 38d posições para a direita". Os números guardados segundo esta representação só são precisos até ao nono algarismo decimal.

Há também uma restrição quanto ao menor número positivo que podemos representar, porque se o primeiro *byte* da forma de cinco *bytes* for ØØ, isto significa que o número é um inteiro (isto é, da forma ØØ ss nn mm ØØ).

Assim, o expoente mais pequeno que podemos ter é Ø1 e o menor número positivo é Ø1 ØØ ØØ ØØ ØØ ou, em decimal, 2.9387359E-39 (zero não é um número positivo). Para nós, isto é Ø.ØØØ, ØØØ, ØØØ, ØØØ, ØØØ, ØØØ, ØØØ, ØØØ, ØØØ, ØØØ, ØØØ, ØØØ, ØØ2, 938, 735, 9 — que, podemos dizer, é verdadeiramente microscópico.

Pode-se verificar tudo isto com o seguinte programa BASIC:

```
1Ø LET A=Ø
2Ø LET B=PEEK 23627+256*PEEK 23628
3Ø FOR I=1 TO 5
4Ø INPUT A$
5Ø PRINT A$;" ";
6Ø POKE B+I, 16*FN K(A$(1))+FN K(A$(2))
7Ø NEXT I
8Ø PRINT ,A
9Ø GO TO 3Ø
1ØØ DEF FN K(X$) = CODE X$-48-(7 AND X$>"9")-(32
AND X$>"F")
```


O programa define uma variável A (a primeira na zona das variáveis) e em seguida substitui o seu valor prévio fazendo POKEs na área das variáveis, um *byte* de cada vez. Se corrermos e introduzirmos *"7F"/"4Ø"/"ØØ"/"ØØ"/ØØ"* deve-se obter o número Ø.375 como resultado. Do mesmo modo, se introduzirmos *"ØØ"/"ØØ"/"Ø3"/"ØØ"/"ØØ"*
ver-se-á escrito o número três. E assim por diante.

Um pormenor interessante é que, se introduzirmos *"ØØ"* como primeiro *byte* e qualquer número ímpar, que não FF, como segundo *byte*, obteremos resultados inesperados — por exemplo, *"ØØ"/"Ø1"/"Ø1"/"ØØ"/"ØØ"* vai resultar, na realidade, em 1Ø23. Uma outra curiosidade é que introduzir *"Ø2"/"D9"/"C7"/"DC"/"EC"* dá o mesmo resultado que *"ØØ"/"FF"/"ØØ"/"ØØ"/"ØØ"* (em teoria isto representa menos zero). Não vale a pena entrar em pânico! Isto na verdade não acontece devido a erro na ROM; aconteceu porque introduzimos uma série de *bytes* sem sentido, que nunca podem surgir de outra maneira. A ROM, na maior parte do tempo, comporta-se com melhor senso que os humanos (apesar de INT −65536 deixar muito a desejar). Deve notar-se que a representação de cinco *bytes* para números decimais permite também representar números inteiros. A representação alternativa para inteiros "curtos", isto é, entre −65535 e +65535, existe apenas para tornar mais rápidas as operações aritméticas com estes inteiros. (O *ZX 81*, por exemplo, não possui esta representação de inteiros "curtos", empregando sempre a mesma forma de cinco *bytes* para todos os números.) Desde que um inteiro esteja fora desses limites, é utilizada a forma geral para representá-lo. O problema que surge frequentemente com o número −65536 deve-se a um erro na concepção da ROM.

Estando fora dos limites para inteiros "curtos", deveria este número ser representado sempre na forma geral, como 91 8Ø ØØ ØØ ØØ; no entanto, algumas vezes a ROM tenta utilizar a forma ØØ FF ØØ ØØ ØØ para o representar, o que, não sendo permitido, conduz a resultados errados.

COMO UTILIZAR OS NÚMEROS DE VÍRGULA FLUTUANTE APROPRIADAMENTE

Um número que exige cinco *bytes* de memória para o guardar, não pode, é claro, ser guardado num registo ou mesmo num par de registos. Podemos, se assim o desejarmos, guardar um tal número em vírgula flutuante num registo *quíntuplo* (como AEDCB, por exemplo) mas se quisermos realizar alguma aritmética, teremos certamente necessidade de guardar mais que um número de cada vez. Assim, a questão é a seguinte: se não podemos guardar estes números em registos, então onde é que os podemos guardar? Resposta: há uma área da RAM reservada para este fim — a pilha do calculador.

A pilha do calculador é uma pilha como aquela com a qual já nos familiarizámos ao longo do livro. Existem apenas uma ou duas diferenças: 1.º ela está guardada numa zona diferente da memória (veja o diagrama da página 165 do manual do *Spectrum*); 2.º a pilha do calculador «cresce» para cima e não para baixo e assim, o endereço do topo da pilha é mais alto do que o do item imediatamente a seguir; 3.º cada item da pilha do calculador ocupa cinco *bytes* e não dois; 4.º ela pode guardar tanto números como cadeias; e 5.º CALL e RET não a alteram de modo algum.

Existem três sub-rotinas diferentes da ROM que são utilizadas para guardar um número na pilha do calculador. A primeira é STACK__A, que toma o inteiro guardado no registo A, converte-o na representação de cinco *bytes*, após o que guarda esses cinco *bytes* no topo da pilha do calculador. A segunda rotina, muito semelhante, é chamada STACK__BC; partindo do inteiro contido no par de registos BC, converte-o na forma de cinco *bytes* e então «empilha» estes cinco *bytes*. A terceira é bastante mais poderosa, já que nos dá total poder sobre os números em vírgula flutuante. Chama-se STACK__AEDCB e a sua função é «empilhar» o conteúdo do «registo quíntuplo» AEDCB na pilha do calculador. Esta última sub-rotina não efectua nenhuma conversão porque parte do princípio que o número já está na forma de cinco *bytes*, com o primeiro *byte* encontrando-se em A, o segundo em E e assim por diante. Os endereços são 2D28 (STACK__A), 2D2B (STACK__BC) e 2AB6 (STACK__AEDCB). Em hex:

```
CD282D          CALL STACK__A
CD2B2D          CALL STACK__BC
```


Incidentalmente, as duas primeiras instruções na rotina STACK__A são LD C,A e LD B,ØØ. Imediatamente a seguir começa a rotina STACK__BC. Esta, por sua vez, carrega A e E ambos com zero, muda BC para D e C (a parte baixa em primeiro lugar), carrega B com zero e salta para STACK__AEDCB. Brilhante. Inversamente, existem sub-rotinas que retiram números da pilha do calculador. São elas FP__TO__A, FP__TO__BC e FP__TO__AEDCB. Estas sub-rotinas são bastante gerais e nunca interrompem o programa com uma mensagem de erro, mesmo se o número no topo da pilha for excessivamente grande para caber no registo designado. Eis então uma descrição do que acontece quando são chamados FP__TO__A ou FP__TO__BC. Suponhamos que o número no topo da linha é chamado Y. Este número é então retirado da pilha, independentemente do facto de caber ou não no registo de destino. Se couber sem problemas, o *flag* zero será *set* e o *flag carry* será *reset*. Se houver qualquer problema, os *flags* dirão qual...

Se o *flag carry* é um, isto significa que o valor absoluto do número (ABS Y) era demasiado grande para caber no registo. Por exemplo, se FP__TO__A fosse usado e o número no topo da pilha fosse 1ØØØd, como este número não pode ser guardado no registo A só por si, isto levaria o *flag carry* a ser um.

Se o *flag* zero está *reset*, isto significa que o número era negativo; no entanto (a não ser que o *flag carry* seja um), o valor absoluto do número (isto é, ABS Y) será guardado no registo designado.

A sub-rotina FP__TO__AEDCB, é claro, nunca irá causar qualquer tipo de problemas, uma vez que qualquer número de cinco *bytes* cabe sempre em cinco registos.

Os endereços são 2DD5 (FP__TO__A),2DA2 (FP__TO__BC) e 2BF1 (FP__TO__AEDCB). Em hex:

```
CDD52D          CALL FP__TO__A
CDA22D          CALL FP__TO__BC
CDF12B          CALL FP__TO__AEDCB
```

A aritmética com números de cinco *bytes* é muito simples. Os endereços das rotinas são:

```
3014        ADD     Soma.
300F        SUB     Subtracção.
30CA        MULT    Multiplicação.
31AF        DIV     Divisão.
```

Eles funcionam assim. O número de cinco *bytes* guardado no endereço especificado por HL [isto é, o número cujos cinco *bytes* são guardados na RAM nos endereços (HL), (HL+1), (HL+2), (HL+3) e (HL+4)] é somado a, multiplicado por, dividido por, ou tem um outro número a subtrair dele. O segundo número é guardado na RAM no endereço apontado por DE. O resultado do cálculo é guardado no endereço especificado por HL.

Para multiplicar os dois números no topo da pilha do calculador, poderia utilizar o seguinte método:

```
2A655C      LD HL,(STKEND)
11FBFF      LD DE,FFFB
19          ADD HL,DE
E5          PUSH HL
22655C      LD (STKEND),HL
19          ADD HL,DE
D1          POP DE
CDCA30      CALL MULT
```

Portanto, HL é carregado com o conteúdo da variável de sistema STKEND, que dá o endereço do primeiro *byte* a seguir ao fim da pilha do calculador. DE é carregado com menos cinco e somado a HL, pelo que HL é diminuído em cinco. Este novo valor é carregado de novo em STKEND porque partimos com dois números na pilha e queremos acabar com apenas um. Este é o endereço de um dos números a ser multiplicado. Se se seguir a listagem cuidadosamente ver-se-á que DE acaba por ser carregado com este valor. HL é diminuído de cinco uma vez mais para apontar o início do outro número a ser multiplicado.

Para verificar que isto realmente funciona, corra este programa:

```
3E06        START   LD A,06
CD282D              CALL STACK_A
3E07                LD A,07
CD282D              CALL STACK_A
2A655C              LD HL,(STKEND)
```



```
11FBFF      LD DE,FFFB
19          ADD HL,DE
E5          PUSH HL
22655C      LD (STKEND),HL
19          ADD HL,DE
D1          POP DE
CDCA30      CALL MULT
CDA22D      CALL FP_TO_BC
C9          RET
```

Vamos fazê-lo correr, fazendo PRINT USR START. O que é que se obtém?

Mas certamente que existem maneiras mais fáceis de multiplicar seis por sete. Apesar de tudo, o programa acima apresentado parece muito complicado e muito difícil de recordar. É a partir daqui que vamos começar a tirar todo o partido da ROM. O seguinte programa faz exactamente o mesmo e já vamos ver porquê:

```
3E06        LD A,06
CD282D      CALL STACK_A
3E07        LD A,07
CD282D      CALL STACK_A
EF          RST 28
04          DEFB 04
38          DEFB 38
CDA22D      CALL FP_TO_BC
C9          RET
```

No *Spectrum*, RST 28 serve para executar aritmética de cinco *bytes*. Os *bytes* que se seguem a RST 28 indicam precisamente quais os cálculos que deve fazer. O *byte* 04 significa «multiplicar os dois números no topo da pilha» — toda a movimentação da pilha do calculador é feita automaticamente. O *byte* 38 é utilizado a seguir à instrução RST 28 para indicar que não há mais cálculos a efectuar a seguir e o próximo *byte* será uma instrução em código máquina.

Os códigos utilizados por RST 28 são: adição=0F, subtracção=03, multiplicação=04 e divisão=05. Estas operações actuam sobre os dois números no topo da pilha. Não devemos esquecer que será preciso um *byte* 38, também, para terminar a sequência de dados.

RST 28 permite-nos fazer muitas coisas, muito mais que aritmética simples. Todas as funções unárias do *Spectrum* ficam à nossa disposição. O código para qualquer função particular é simplesmente o código de carácter dessa função menos 93. Por exemplo, o código de carácter de SIN é B2, que menos 93 é 1F. Isto significa que podemos encontrar o seno de qualquer número no topo da pilha do calculador utilizando a sequência:

```
EF          RST 28
1F          DEFB 1F (sin)
38          DEFB 38 (end-calc).
```

Para multiplicar dois números (no topo da pilha do calculador) e achar a raiz quadrada do resultado podemos empregar a sequência:

```
EF          RST 28
04          DEFB 04 (mult)
28          DEFB 28 (sqr)
38          DEFB 38 (exit).
```

Para que se veja bem o resultado, vamos correr este programa, que multiplica cinco por vinte e então achamos a raiz quadrada do resultado:

```
3E05        LD A,05
CD282D      CALL STACK_A
3E14        LD A,14
CD282D      CALL STACK_A
EF          RST 28
04          DEFB 04 (mult)
28          DEFB 28 (sqr)
38          DEFB 38 (end_calc)
CDA22D      CALL FP_TO_BC
C9          RET
```

Repare-se que é a primeira vez que utilizamos mais que um código de operação a seguir a RST 28. De facto, podemos utilizar tantos quantos quisermos, desde que se acabe a lista com 38.

Para pouparmos o trabalho de calcular todos os códigos, o Apêndice Seis contém uma lista de códigos RST 28 disponíveis.

Poderá utilizar-se, imediatamente a maioria destes sem ser precisa nenhuma explicação adicional. Algumas das funções que figuram nesta lista são surpreendentes. Por exemplo, temos duas funções — uma chamada «usr-sr» e outra «usr-n». «usr-sr» é a função USR *(string)* em BASIC que dá o endereço da zona de RAM onde se encontra a matriz de pontos para um qualquer carácter gráfico «para uso do utilizador». «usr-n» é a função USR (número) e isto pode parecer muito estranho. O que acontece quando a ROM encontra o *byte* 2D (usr-n) entre os *bytes* de dados de RST 28, é que o número no topo da pilha do calculador é carregado em BC. Então, a sub-rotina em código máquina a partir do endereço BC é chamada. Ao retornar desta sub-rotina, o valor corrente de BC é «empilhado» no topo da pilha do calculador (em forma de cinco *bytes*, é claro) e, seguidamente, o próximo *byte* na sequência de dados de RST 28 (isto é, a seguir a 2D) é interpretada.

PEEK (*byte* 2B) funciona da mesma maneira, retirando um endereço da pilha do calculador, achando o conteúdo desse endereço e, finalmente, "empilhando" esse valor no topo da pilha do calculador.

Todas as funções, quando empregues deste modo, provocam a remoção do número de momento no topo da pilha do calculador e a sua substituição por um novo. Por exemplo, se o número no topo da pilha é 3.5, se a função INT for chamada, o número 3.5 será removido e substituído pelo número três.

As funções da *string* CODE, VAL e LEN e também VAL$, STR$ e CHR$ precisam de alguma explicação. Repare-se que, além de guardar números, a pilha do calculador também pode guardar *strings* e assim, se começar com o número 2000d no topo da pilha e chamar STR$ (empregando o código 2E após RST 28), o item no topo da pilha da calculadora será agora a *string* "000". Podemos demonstrar isto como se vê abaixo. Note-se que a pilha do calculador na verdade não guarda *strings* mas, sim, parâmetros — o endereço do início da *string* e o seu comprimento. Esta informação é guardada em cinco *bytes*, sendo o primeiro sem significado, o segundo e terceiro o endereço do início e o quarto e o quinto, o comprimento da *string*.

```
01D007      LD BC,2000d
CD2B2D      CALL STACK_BC
```

```
EF              RST 28
2E              DEFB 2E (str$)
1C              DEFB 1C (code)
38              DEFB 38 (exit)
CDA22D          CALL FP__TO__BC
C9              RET
```

Isto deve dar o resultado de CODE STR$ 2ØØØ. Mas será que o dá?

EMPREGO DA MEMÓRIA DO CALCULADOR

Se passarmos uma vista de olhos pelo manual, veremos que uma das variáveis de sistema, MEMBOT, tem trinta *bytes* de comprimento. Esta é a zona de memória do calculador. Um cálculo rápido, com uma divisão por cinco, mostra que isto é o espaço suficiente para guardar seis números diferentes de cinco *bytes*. As seis zonas diferentes da memória podem ser, cada uma delas, utilizadas por RST 28 para guardar ou recuperar tanto os números quanto as *strings* do topo da pilha do calculador. Existem doze códigos diferentes para realizarmos isto, que são:

CØ	guarda na memória zero
C1	guarda na memória um
C2	guarda na memória dois
C3	guarda na memória três
C4	guarda na memória quatro
C5	guarda na memória cinco
EØ	chama da memória zero
E1	chama da memória um
E2	chama da memória dois
E3	chama da memória três
E4	chama da memória quatro
E5	chama da memória cinco

O efeito de guardar um número é copiá-lo do topo da pilha do calculador (sem o retirar daí) e o efeito de chamar um número é aumentar a pilha de um item, "empilhando" o conteúdo da memória especificada. (Isto sem destruir o número que anteriormente se encontrava no topo.)

Eis, no entanto, alguns avisos a respeito do uso destas memórias. Primeiro, algumas funções, como SIN, COS e STR$ (só para mencionar algumas), apagam o conteúdo anterior das memórias zero, um e dois, de modo que, para estar realmente seguro, é bom utilizar apenas as memórias três, quatro e cinco.

Há ainda uma outra complicação — sempre que se utilizar RST 10 para escrever os caracteres gráficos obtidos a partir das teclas numéricas (isto é, os caracteres com os códigos de 8Ø a 8F) as memórias zero e um serão apagadas. Tendo isto em conta, vamos ver o que podemos fazer daqui para a frente.

Suponhamos que queremos calcular SIN X+COS X. Podemos utilizar a seguinte técnica. Partimos do princípio que X está no topo da pilha:

```
EF              RST 28
C5              DEFB C5 (store__5)
1F              DEFB 1F (sin)
E5              DEFB E5 (recall 5)
2Ø              DEFB 2Ø (cos)
ØF              DEFB ØF (add)
38              DEFB 38 (end__calc)
```

Note-se que X não precisa de ser chamado imediatamente depois de ter sido guardado, porque o acto de o guardarmos não o retira da pilha. Guardar um número na memória deixa o comprimento da pilha do calculador sem alteração.

Por outro lado, chamar um número da memória aumenta, na realidade, o comprimento da pilha do calculador por lhe acrescentar mais um item. A rotina acima dada funciona começando por guardar X na memória cinco, alterando X para SIN X, tornando a chamar X ao topo da pilha e mudando este X para COS X. Nesta altura encontram-se dois números na pilha: SIN X e COS X. A rotina ADD, chamada a seguir, substitui ambos por um só resultado, aquele que pretendemos, SIN X+COS X.

Realizámos agora um cálculo bastante complexo envolvendo funções trigonométricas, em apenas sete *bytes!*

Há ainda uma ou duas coisas que podemos fazer com RST 28. Podemos usar as funções lógicas AND e OR (isto é, as funções de BASIC AND e OR). Podemos trocar os dois items do topo, apagar

o item do topo ou duplicá-lo e assim por diante. Que outras coisas interessantes conseguiremos fazer?

NÚMEROS ALEATÓRIOS

A seguinte rotina irá colocar um número aleatório decimal entre zero e um (obrigatoriamente menor que um, mas podendo ser igual a zero) no topo da pilha do calculador. Tente-se compreender como ela funciona:

```
3EA5        LD A,"RND"
CD282D      CALL STACK_A
061D        LD B,1D
EF          RST 28
2F          DEFB 2F (chr$)
1D          DEFB 1D (val)
38          DEFB 38 (end_calc)
```

E para terminar, a seguinte rotina irá deixar no par de registos BC um inteiro aleatório entre um e o valor inicial de BC:

```
CD2B2D      CALL STACK_BC
3EA5        LD A,"RND"
CD282D      CALL STACK_A
061D        LD B,1D
EF          RST 28
2F          DEFB 2F (chr$)
1D          DEFB 1D (val)
04          DEFB 04 (mult)
27          DEFB 27 (int)
38          DEFB 38 (end_calc)
CDA22D      CALL FP_TO_BC
03          INC BC
```

A seguinte sequência eleva um número à potência de um outro. Depois de RTS 28: 01 25 04 26 34. (Note que o *byte* 06 realiza só por si aproximadamente os mesmos resultados.)

Podemos fazer muito mais matemática com RST 28 do que com uma só expressão BASIC — calcular séries de Taylor para começar. No entanto, deixaremos tudo isso para outra ocasião.

A continuar...

NOTAS DO TRADUTOR

CAPÍTULO 8

[1] O processador *Z80* admite dois tipos de interrupções: as normais *(maskable interrupts)*, que podem ser ignoradas pelo processador, ou atendidas, conforme indicação nesse sentido contida no programa, e as interrupções NMI *(non maskable interrupts)*, que são sempre atendidas.

Ao receber uma interrupção normal, o processador verifica o estado do *flag* IFF1; se for zero, o pedido de interrupção é ignorado, prosseguindo a execução normal do programa; se for um, a interrupção é atendida, sendo efectuado um CALL a uma rotina apropriada, guardando-se na pilha o endereço de retorno (o da instrução que ia ser executada ao ser atendida a interrupção). Note-se no entanto que as instruções «automáticas» (LDIR, LDDR, CPIR, CPDR, INIR, INDR, OTIR e OTDR) podem ser interrompidas. A repetição «automática» consegue-se, nestas instruções, impedindo que o *program counter* seja incrementado, antes do fim da instrução. Ao ser recebida uma interrupção (normal ou NMI) o *program counter* é guardado na pilha e a execução é transferida para a rotina de interrupção conveniente. O retorno dessa rotina faz-se portanto para o endereço da instrução «automática» que estava a ser executada. Desde que a rotina de interrupção salvaguarde os registos utilizados na instrução «automática», a execução desta prossegue sem perturbações. É isto que permite no *Spectrum* copiar longos blocos de memória sem perturbação das interrupções que ocorrem cinquenta vezes por segundo. Para impedir a interferência de outras interrupções, o *flag* IFF1 torna-se zero, até que seja executada uma instrução EI.

Normalmente, uma rotina de interrupção começa por salvaguardar todos os registos a serem utilizados (normalmente na pilha), de forma que, antes do retorno, possa ser estabelecido o estado do processador que existia na altura do atendimento da interrupção.

Se a rotina exige a atenção exclusiva do processador, a instrução EI será colocada imediatamente antes do retorno; se não, é possível fazer EI imediatamente após a salvaguarda dos registos, o que permite o atendimento de outras interrupções de dentro de uma rotina de interrupção.

[2] O processador *Z80* pode operar num de três modos de interrupção: modo 0, modo 1 e modo 2.

Modo 0: é aquele em que o processador entra automaticamente, ao ser-lhe aplicada a alimentação ou após um sinal RESET. Neste modo, o processador espera receber do exterior o código da próxima instrução a efectuar, em vez de o ler na memória. Esta instrução, cuja execução é intercalada com as instruções do programa, é normalmente um RST, por ser uma instrução de um *byte*, com a possibilidade de desviar a execução para qualquer um de oito endereços possíveis (00, 08, 10, 18, 20, 28, 30 e 38), guardando um endereço de retorno na pilha. (Este é o módo de interrupção do processador *8080* e existe no *Z80* para permitir a compatibilidade entre os dois processadores.)

Modo 1: neste modo é intercalada automaticamente a instrução RST 38; é o modo em que o *Spectrum* normalmente trabalha. Este modo, o menos poderoso, destina-se a simplificar o *hardware* em sistemas pouco complexos, nos quais basta a resposta a um tipo de interrupção externa.

Modo 2: este é o modo mais poderoso. Neste caso, é necessário existir na memória uma tabela de endereços das várias rotinas de interrupção; são possíveis até 128 rotinas. A parte alta do endereço inicial dessa tabela deve ser carregada no registo I *(Interrupt vector register)*. Ao atender a interrupção, o processador espera receber do exterior um *byte* — a parte baixa de um endereço — que, reunido ao valor do registo I, produz o endereço de uma posição na tabela de interrupção. Nesse endereço, deve estar guardado o endereço da rotina de interrupção requerida, para onde a execução é transferida, após salvaguarda na pilha do endereço de retorno — o da instrução seguinte à que estava a ser executada, ao ser recebida a interrupção.

A escolha de um destes modos de interrupção, dependente do *hardware*, é efectuada pelas instruções IM Ø, o que selecciona o modo Ø (que é o modo por defeito), IM 1, seleccionando o modo 1, e IM 2, que selecciona o modo 2.

Devido a particularidades de *hardware*, o *Spectrum* opera também no modo Ø, sem qualquer cuidado especial (e também sem utilidade), e no modo 2, desde que se proceda da seguinte forma:

1.º carregar numa zona livre da memória a nova rotina de interrupção, tendo o cuidado de incluir na rotina a salvaguarda de todos os registos utilizados e antecedendo o retorno por EI;

2.º achar dois *bytes* livres na memória, de endereços 256*n+255 e 256*(n+1) (em que n é um inteiro < 255) e guardar nessas posições o endereço da rotina referida em 1 (primeiro parte baixa, a seguir a alta, como é costume);

3.º carregar o registo I com o número n por meio de LD A,n/LD I,A;

4.º passar ao modo 2 através de IM 2. O interesse de operar no modo 2, no *Spectrum*, é o de permitir sincronizar certas acções com o relógio interno, que gera cinquenta interrupções por segundo.

[3] O registo R (*Refresh register* — registo de refrescamento) é um registo de oito *bits* que se destina a facilitar a utilização das chamadas memórias dinâmicas (o *Spectrum* usa dessas memórias), que necessitam, para manter intacta a informação nelas guardada, de um "refrescamento" pelo menos a cada dois ms.

Os sete *bits* inferiores de R formam um contador que é incrementado de um por cada instrução executada (algumas instruções incrementam-no em mais de uma unidade). O oitavo *bit*, que não participa na contagem, mantém o seu valor — se for inicialmente zero, continua zero; se for um continua sempre um, até ser mudado por meio de LD R,A. De facto, se em algum ponto de um programa houver uma instrução LD A,R, é pouco provável que se consiga saber antecipadamente o valor que A irá conter, ao ser executada a instrução.

Embora destinado essencialmente ao refrescamento da memória, é possível usar este registo para obter um número aleatório entre 0 e 127, ou entre 128 e 255.

[4] A instrução *Return from Interrupt*, que tem a mesma acção que RET, tem por finalidade terminar as rotinas de interrupção, no modo 2. O seu código é reconhecido por certos periféricos, que ficam a saber desta maneira que o processador está livre para aceitar novas interrupções. Isto é necessário se for preciso atender interrupções de prioridade diferentes (no *Spectrum* não tem utilidade).

[5] Interrupções NMI. Estas interrupções são sempre atendidas, independentemente

do *flag* IFF1 — podem ocorrer, portanto, dentro de outras rotinas de interrupção. Destina-se a forçar o atendimento a acontecimentos que requerem obrigatoriamente e sem demora a atenção do processador — como por exemplo, falhas na alimentação, tentativas de acesso a zonas protegidas da memória, etc.

Ao ser recebida uma destas interrupções, o processador executa o equivalente a um RST, mas para o endereço 66h, onde deve estar a rotina correspondente.

Para evitar a interferência de outras interrupções (excepto as NMI, é claro) o *flag* IFF1 é posto a zero. No entanto, uma vez que, ao ser atendida esta interrupção, o processador podia estar num estado em que as interrupções normais eram autorizadas, ou proibidas, isto é, IFF1 podia ser um ou zero, para permitir o restabelecimento total do estado do processador, é guardada uma cópia no valor inicial de IFF1 num outro *flip-flop*, chamado IFF2. Uma vez completada a rotina de interrupção, se for conveniente (por vezes não é, nos casos em que o motivo que provoca a interrupção torna impossível a continuação da operação normal), é possível restabelecer o estado anterior de IFF1 por meio da instrução RETN *(RETurn from Non maskable interrupt)*, que provoca o retorno após ter copiado o *flag* IFF2 para IFF1.

Em resumo: a instrução EI torna IFF1 e IFF2 iguais a um. A instrução DI, ou o reconhecimento de uma interrupção normal, torna-os ambos iguais a zero. O atendimento de um NMI faz IFF1 igual a zero, sem afectar IFF2. A execução de RETN provoca, além do retorno, a cópia de IFF2 para IFF1. O valor de IFF2 pode ser testado pelas instruções LD A,I ou LD A,R — este *flag* é copiado para o *flag* P/V.

No *Spectrum*, as instruções NMI não são utilizadas pelo *hardware* nem são utilizáveis externamente, devido a um erro na ROM. Sugestão: examinemos a ROM a partir do endereço 66 hex = 102d para ver se descobrimos o erro. Note-se também que a rotina NMI termina por RETN.

[6] Um assunto importante que o autor não abordou é o dos tempos de execução das várias instruções, necessários, em muitos casos, para determinar o tempo exacto de execução de certas sequências de instruções.

O processador *Z80* "mede" o tempo por meio de um sinal externo — o sinal *clock* — proveniente de um oscilador (controlado a quartzo normalmente), aplicado ao pino *clock*. Os tempos de execução das instruções dependem da própria instrução, que demora sempre um certo número de ciclos *clock* e da frequência deste sinal *clock*. Para calcular em microssegundos o tempo de execução de uma instrução, basta dividir o número de ciclos de *clock* em Mhz.

No *Z80*, as instruções mais rápidas demoram quatro ciclos de *clock* e as mais lentas, vinte e três (sem falar das "automáticas"). É interessante notar que não existe relação directa entre o número de *bytes* de uma instrução e o seu tempo de execução. Assim, por exemplo, a instrução LD IX,O, de quatro *bytes* (DD 21 ØØ ØØ), demora 14 ciclos de *clock*, enquanto que a instrução EX (SP),HL, de um *byte* (E3), demora 19 ciclos.

A execução de cada instrução estende-se por certo número de "ciclos máquina" — entre um e seis, cada um deles demorando vários ciclos de *clock*.

O "ciclo máquina" fundamental é o que lê a instrução a executar da memória. Este ciclo, chamado M1, demora quatro ciclos de *clock*, sendo os dois primeiros ocupados na leitura da instrução da memória e os dois últimos na descodificação e

eventual execução dessa instrução. (Simultaneamente nestes últimos ciclos ocorre um "refrescamento" da memória, sendo o registo R incrementado.) No caso de a instrução lida ser um só *byte*, e se a sua execução pode ser completada durante a segunda parte do ciclo M1, não havendo necessidade de aceder à memória, a instrução completa demora apenas os quatro ciclos de *clock* do ciclo M1.

É esse o caso de instruções como LD registo, registo (de oito *bits*) excepto I e R, por exemplo.

Se as instruções obrigarem à transferência de informação de e para a memória, ao ciclo M1 juntam-se ciclos suplementares, de três ciclos de *clock* por cada *byte* transferido — por exemplo, ADD A,n ou SUB (HL) demoram sete ciclos de *clock*: quatro do ciclo M1 e mais três para transferir o *byte* suplementar. Por vezes, além dos ciclos impostos pelas transferências com a memória, a execução de uma instrução exige ciclos internos suplementares para se efectuar. É o caso, por exemplo, de JR C,n; se *carry* igual a zero, a instrução demora, como seria de esperar, sete ciclos *clock*; se *carry* igual a um, no entanto, o número de ciclos passa a ser de 12 — isto porque é necessário adicionar o número n ao *program counter*, o que requer cinco ciclos adicionais. Na tabela em apêndice encontram-se os tempos de execução de todas as instruções do processador.

CAPÍTULO 13

[1] Os endereços indicados são diferentes dos do original. O autor não se apercebeu de que o programa não funciona correctamente abaixo de 8000, indicando para endereço original 6800; iremos apresentar o programa com os endereços alterados, começando em E800. Isto significa que os possuidores do *Spectrum* de 16 K não conseguirão fazê-lo funcionar correctamente.

CAPÍTULO 14

[1] O autor certamente não está a falar a sério. Convém desenvolver a técnica de programação de forma a conseguir obter o máximo proveito dos recursos ao nosso alcance sem impor limitações desnecessárias. Neste caso, o emprego injustificado de DI poderá vir a impedir um desenvolvimento frutuoso do programa (por exemplo, um que, sendo capaz de estudar com avanço as possíveis respostas do adversário, utilizasse FRAMES para impor um tempo máximo de resposta). Convém também ter em conta a generalidade da técnica desenvolvida — se no *Spectrum* é possível fazer DI sem grandes problemas, nem sempre isso se passa em sistemas mais complexos. Dá-se mesmo o caso, em certos processadores de 16 *bits*, como o *Z8000* ou *MC68000*, de as instruções equivalentes a DI e EI nem sequer serem permitidas a um programa de utilizador, sendo reservadas aos programas de sistema (a propósito, o novo computador *Sinclair*, o *QL*, utiliza o processador *68008*, que é essencialmente o *MC 68000*).

APÊNDICES

Estes apêndices destinam-se a proporcionar uma referência rápida e fácil a um grande número de informações úteis.

Podemos encontrar no Capítulo Oito uma lista minuciosa do efeito preciso de cada instrução *Z80*. Este deve ser tratado como um apêndice separado.

Os apêndices são os seguintes:

APÊNDICE UM	– Uma tabela de conversão hexadecimal/decimal e vice-versa e outra tabela de conversão hexadecimal/binário e vice-versa.
APÊNDICE DOIS	– As variáveis de sistema.
APÊNDICE TRÊS	– Uma tabela de conversão de hex para *assembly*.
APÊNDICE QUATRO	– Uma tabela de conversão de *assembly* para hex, incluindo o efeito de cada instrução nos *flags*.
APÊNDICE CINCO	– O conjunto de caracteres do *Spectrum*.
APÊNDICE SEIS	– Significado de alguns dos códigos dos *bytes* de dados utilizados a seguir a RST 28.

APÊNDICE UM

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	-128	-127	-126	-125	-124	-123	-122	-121	-120	-119	-118	-117	-116	-115	-114	-113
9	-112	-111	-110	-109	-108	-107	-106	-105	-104	-103	-102	-101	-100	-99	-98	-97
A	-96	-95	-94	-93	-92	-91	-90	-89	-88	-87	-86	-85	-84	-83	-82	-81
B	-80	-79	-78	-77	-76	-75	-74	-73	-72	-71	-70	-69	-68	-67	-66	-65
C	-64	-63	-62	-61	-60	-59	-58	-57	-56	-55	-54	-53	-52	-51	-50	-49
D	-48	-47	-46	-45	-44	-43	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33
E	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17
F	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

APÊNDICE DOIS

Decimal	Hex		Nome
23552	5C00	IY+00	KSTATE
23560	5C08	IY+08	LAST_K
23561	5C09	IY+09	REPDEL
23562	5C0A	IY+0A	REPPER
23563	5C0B	IY+0B	DEFADD
23565	5C0D	IY+0D	K_DATA
23566	5C0E	IY+0E	TVDATA
23568	5C10	IY+10	STRMS
23606	5C36	IY+36	CHARS
23608	5C38	IY+38	RASP
23609	5C39	IY+39	PIP
23610	5C3A	IY+3A	ERR_NR
23611	5C3B	IY+3B	FLAGS
23612	5C3C	IY+3C	TVFLAG
23613	5C3D	IY+3D	ERR_SP
23615	5C3F	IY+3F	LISTSP
23617	5C41	IY+41	MODE
23618	5C42	IY+42	NEWPPC
23620	5C44	IY+44	NSPPC
23621	5C45	IY+45	PPC
23623	5C47	IY+47	SUBPPC
23624	5C48	IY+48	BORDCR
23625	5C49	IY+49	E_PPC
23627	5C4B	IY+4B	VARS
23629	5C4D	IY+4D	DEST
23631	5C4F	IY+4F	CHANS
23633	5C51	IY+51	CURCHL
23635	5C53	IY+53	PROG
23637	5C55	IY+55	NXTLIN
23639	5C57	IY+57	DATADD
23641	5C59	IY+59	E_LINE
23643	5C5B	IY+5B	K_CUR
23645	5C5D	IY+5D	CH_ADD
23647	5C5F	IY+5F	X_PTR
23649	5C61	IY+61	WORKSP
23651	5C63	IY+63	STKBOT
23653	5C65	IY+65	STKEND
23655	5C67	IY+67	B_REG
23656	5C68	IY+68	MEM
23658	5C6A	IY+6A	FLAGS2

Decimal	Hex		Nome
23659	5C6B	IY+6B	DF_SZ
23660	5C6C	IY+6C	S_TOP
23662	5C6E	IY+6E	OLDPPC
23664	5C70	IY+70	OSPPC
23665	5C71	IY+71	FLAGX
23666	5C72	IY+72	STRLEN
23668	5C74	IY+74	T_ADDR
23670	5C76	IY+76	SEED
23672	5C78	IY+78	FRAMES
23675	5C7B	IY+7B	UDG
23677	5C7D	IY+7D	COORDS
23679	5C7F	IY+7F	P_POSN
23680	5C80	IY+80	PR_CC
23681	5C81	IY+81	SPARE1
23682	5C82	IY+82	ECHO_E
23684	5C84	IY+84	DF_CC
23686	5C86	IY+86	DFCCL
23688	5C88	IY+88	S_POSN
23690	5C8A	IY+8A	SPOSNL
23692	5C8C	IY+8C	SCR_CT
23693	5C8D	IY+8D	ATTR_P
23694	5C8E	IY+8E	MASK_P
23695	5C8F	IY+8F	ATTR_T
23696	5C90	IY+90	MASK_T
23697	5C91	IY+91	P_FLAG
23698	5C92	IY+92	MEMBOT
23728	5CB0	IY+B0	SPARE2
23730	5CB2	IY+B2	RAMTOP
23732	5CB4	IY+B4	P_RAMT

VARS. DE SISTEMA	ENDR.	DEL. DE IY	N.º DE BYTES	PROPÓSITO
ATTR_P	5C8D	53	1	Cores permanentes correntes (empregados por CLS, etc).
ATTR_T	5C8F	55	1	Cores temporárias correntes (empregadas por RST 10).
BORDCR	5C48	0E	1	Cores temporárias correntes para a parte inferior do écran. Também utilizado para guardar a cor de moldura.*8.
B_REG	5C67	2D	1	Utilizado pelo calculador de vírgula flutuante.
CHANS	5C4F	15	2	Endereço de início da zona de informação dos canais.
CHARS	5C36	-	2	Endereço do conjunto de caracteres *standard* menos 0100.
CH_ADD	5C5D	23	2	Endereço do próximo carácter a interpretar.
COORDS	5C7D	43	2	Coordenadas do último ponto referido numa instrução PLOT.
CURCHL	5C51	17	2	Endereço da informação relativa ao canal corrente de entrada/saída.
DATADD	5C57	1D	2	Endereço dentro do programa do carácter a seguir ao último item de DATA lido.
DEFADD	5C0B	-	2	Endereço dos argumentos das funções «para definição pelo utilizador».
DEST	5C4D	13	2	Endereço da variável a que se vai ter acesso.
DF_CC	5C84	4A	2	Endereço da posição de impressão na parte superior do écran.
DFCCL	5C86	4C	2	Endereço da posição de impressão na parte inferior do écran.
DF_SZ	5C6B	31	1	Número de linhas na parte inferior do écran.
ECHO_E	5C82	48	2	Coordenadas na parte inferior do écran para lá das quais o «cursor direito» não tem efeito*.
ERR_NR	5C3A	00	1	Código de erro corrente menos um.
ERR_SP	5C3D	03	2	Endereço do endereço de retorno de RST 08.
E_LINE	5C59	1F	2	Endereço do início da linha edit.
E_PPC	5C49	0F	2	Número de linha da linha com o cursor.
FLAGS	5C3B	01	1	Vários *flags*.
FLAGS2	5C6A	30	1	Vários *flags*.
FLAGX	5C71	37	1	Vários *flags*.
FRAMES	5C78	3E	3	Incrementado a cada quadro imagem de TV (50 vezes por segundo).
KSTATE	5C00	-	8	Informação do teclado utilizada na repetição de uma tecla.
K_CUR	5C5B	21	2	Endereço do cursor.
K_DATA	5C0D	-	1	Guarda o segundo *byte* do código de controle de cores entrado a partir do teclado (exemplo, ambos os *shifts* 4 guardam o *byte* 04).
LAST_K	5C08	-	1	Código de carácter da última tecla carregada.
LISTSP	5C3F	05	2	Endereço do endereço de retorno da listagem automática.
MASK_P	5C8E	54	1	«Máscara» dos atributos, permanente.
MASK_T	5C90	56	1	«Máscara» dos atributos, temporária.
MEM	5C68	2E	2	Endereço de início da área da memória do calculador.
MEMBOT	5C92	58	1E	Área utilizada 1.º como memória do calculador, 2.º para calcular a forma decimal dos números em vírgula flutuante e 3.º para calcular a matriz de *pixels* dos caracteres gráficos de 80 a 8F quando escritos.
MODE	5C41	07	1	Cursor corrente (00=K/L/C,01=E,02=G).
NEWPPC	5C42	08	2	Número de linha da linha a executar a seguir.
NSPPC	5C44	0A	1	N.º da instrução dentro da linha p. onde é transferido o controle.
NXTLIN	5C55	1B	2	Endereço da linha de programa a executar a seguir.
OLDPPC	5C6E	34	2	Número de linha para a qual CONTINUE transfere o controle.
OSPPC	5C70	36	1	N.º da instrução na linha para a qual CONTINUE transfere o controle.
PIP	5C39	-	1	Duração do *click* do teclado.

PPC	5C45	0B	2	Número de linha da linha corrente a ser executada.
PROG	5C53	19	2	Endereço de início da área de programas BASIC.
PR_CC	5C80	46	1	Endr. da posição de impressão p. L PRINT (a parte alta deve ser 5B).
P_FLAG	5C91	57	1	Vários *flags*.
P_POSN	5C7F	45	1	Coordenada X da posição de impressão L PRINT*.
P_RAMT	5CB4	7A	2	Endereço do último *byte* realmente existente na RAM.
RAMTOP	5CB2	78	2	Endereço do último *byte* a apagar por NEW.
RASP	5C38	FE	1	Duração do besouro de aviso.
REPDEL	5C09	CF	1	Demora antes que uma tecla repita (em 1/50 de segundo).
REPPER	5C0A	D0	1	Demora entre repetições sucessivas (em 1/50 de segundo).
SCR_CT	5C8C	52	1	Número de *scrolls* a realizar mais um.
SEED	5C76	3C	2	Fonte para o gerador de números aleatórios.
SPARE1	5CB1	47	1	Um *byte* não utilizado.
SPARE2	5CB0	76	2	Dois *bytes* não utilizados.
STKBOT	5C63	29	2	Endereço do início da pilha do calculador.
STKEND	5C65	2B	2	Endereço do início da RAM livre.
STRLEN	5C72	38	2	Comprimento da *string* a definir.
STRMS	5C10	D6	26	Endereços dos canais ligados a *streams*.
SUBPPC	5C47	0D	1	Número de intrução dentro de uma linha a executar.
S_POSN	5C88	4E	2	Coordenadas da posição de impressão na parte superior do écran*.
SPOSNL	5C8A	50	2	Coordenadas da posição de impressão na parte inferior do écran*.
S_TOP	5C6C	32	2	Número de linha da linha no topo do écran.
TVDATA	5C0E	D4	2	Segundo e terceiro *bytes* dos códigos de controle a serem processados.
TVFLAG	5C3C	02	1	Vários *flags*.
T_ADDR	5C74	3A	2	Endereço do item seguinte na tabela de sintaxe.
UDG	5C7B	41	2	Endereço do início da área dos caracteres gráficos.
VARS	5C4B	11	2	Endereço do início da área das variáveis.
WORKSP	5C61	27	2	Endereço do início do espaço de trabalho temporário.
X_PTR	5C3F	25	2	Endereço do primeiro carácter sintacticamente incorrecto.

*As coordenadas são consideradas na convenção «-y,x», na qual a primeira coordenada é 24d menos o número de linha e a segunda coordenada é 33d menos o número de coluna. Estas duas coordenadas são, é claro, guardadas pela ordem inversa na zona das variáveis de sistema assim como todas as quantidades de dois bytes.

APÊNDICE TRÊS

	0	1	2	3	4	5	6	7
0	NOP	LD BC,nn	LD (BC),A	INC BC	INC B	DEC B	LD B,n	RLCA
1	DJNZ e	LD DE,nn	LD (DE),A	INC DE	INC D	DEC D	LD D,n	RLA
2	JR NZ,e	LD HL,nn	LD (pq),HL	INC HL	INC H	DEC H	LD H,n	DAA
3	JR NC,e	LD SP,nn	LD (pq),A	INC SP	INC (HL)	DEC (HL)	LD (HL),n	SCF
4	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A
5	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A
6	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A
7	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A
8	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A
9	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A
A	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A
B	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A
C	RET NZ	POP BC	JP NZ,pq	JP pq	CALL NZ,pq	PUSH BC	ADD A,n	RST 00
D	RET NC	POP DE	JP NC,pq	OUT (n),A	CALL NC,pq	PUSH DE	SUB n	RST 10
E	RET PO	POP HL	JP PO,pq	EX (SP),HL	CALL PO,pq	PUSH HL	AND n	RST 20
F	RET P	POP AF	JP P,pq	DI	CALL P,pq	PUSH AF	OR n	RST 30

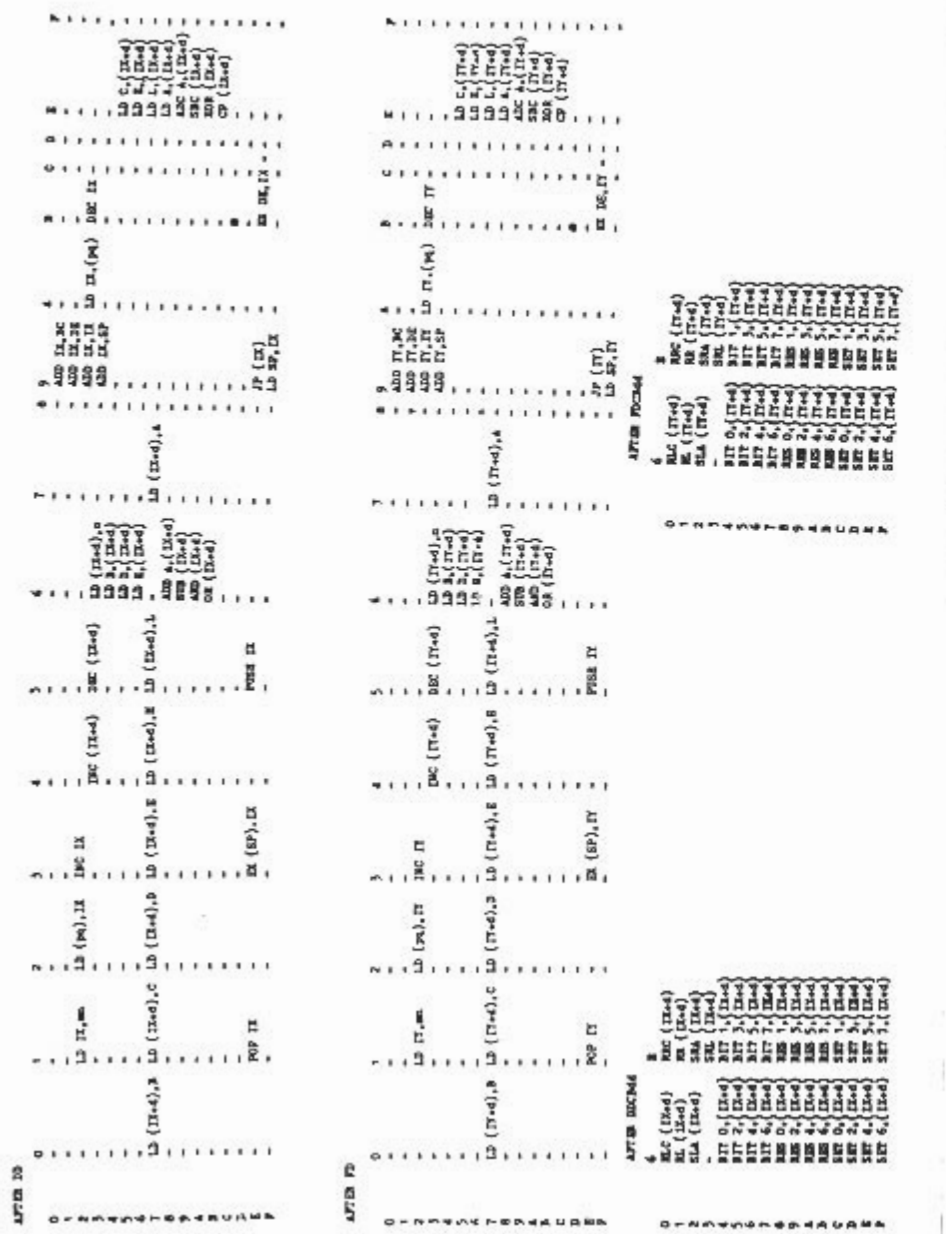
	8	9	A	B	C	D	E	F
0	EX AF,AF'	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,n	RRCA
1	JR e	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,n	RRA
2	JR Z,e	ADD HL,HL	LD HL,(pq)	DEC HL	INC L	DEC L	LD L,n	CPL
3	JR C,e	ADD HL,SP	LD A,(pq)	DEC SP	INC A	DEC A	LD A,n	CCF
4	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
B	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
C	RET Z	RET	JP Z,pq	●	CALL Z,pq	CALL pq	ADC A,n	RST 08
D	RET C	EXX	JP C,pq	IN A,(n)	CALL C,pq	●	SBC A,n	RST 18
E	RET PE	JP (HL)	JP PE,pq	EX DE,HL	CALL PE,pq	●	XOR n	RST 28
F	RET M	LD SP,HL	JP M,pq	EI	CALL M,pq	●	CP n	RST 38

AFTER CB

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	RLC B	RLC C	RLC D	RLC E	RLC H	RLC L	RLC (HL)	RLC A	RRC B	RRC C	RRC D	RRC E	RRC H	RRC L	RRC (HL)	RRC A
1	RL B	RL C	RL D	RL E	RL H	RL L	RL (HL)	RL A	RR B	RR C	RR D	RR E	RR H	RR L	RR (HL)	RR A
2	SLA B	SLA C	SLA D	SLA E	SLA H	SLA L	SLA (HL)	SLA A	SRA B	SRA C	SRA D	SRA E	SRA H	SRA L	SRA (HL)	SRA A
3	-	-	-	-	-	-	-	-	SRL B	SRL C	SRL D	SRL E	SRL H	SRL L	SRL (HL)	SRL A
4	BIT 0,B	BIT 0,C	BIT 0,D	BIT 0,E	BIT 0,H	BIT 0,L	BIT 0,(HL)	BIT 0,A	BIT 1,B	BIT 1,C	BIT 1,D	BIT 1,E	BIT 1,H	BIT 1,L	BIT 1,(HL)	BIT 1,A
5	BIT 2,B	BIT 2,C	BIT 2,D	BIT 2,E	BIT 2,H	BIT 2,L	BIT 2,(HL)	BIT 2,A	BIT 3,B	BIT 3,C	BIT 3,D	BIT 3,E	BIT 3,H	BIT 3,L	BIT 3,(HL)	BIT 3,A
6	BIT 4,B	BIT 4,C	BIT 4,D	BIT 4,E	BIT 4,H	BIT 4,L	BIT 4,(HL)	BIT 4,A	BIT 5,B	BIT 5,C	BIT 5,D	BIT 5,E	BIT 5,H	BIT 5,L	BIT 5,(HL)	BIT 5,A
7	BIT 6,B	BIT 6,C	BIT 6,D	BIT 6,E	BIT 6,H	BIT 6,L	BIT 6,(HL)	BIT 6,A	BIT 7,B	BIT 7,C	BIT 7,D	BIT 7,E	BIT 7,H	BIT 7,L	BIT 7,(HL)	BIT 7,A
8	RES 0,B	RES 0,C	RES 0,D	RES 0,E	RES 0,H	RES 0,L	RES 0,(HL)	RES 0,A	RES 1,B	RES 1,C	RES 1,D	RES 1,E	RES 1,H	RES 1,L	RES 1,(HL)	RES 1,A
9	RES 2,B	RES 2,C	RES 2,D	RES 2,E	RES 2,H	RES 2,L	RES 2,(HL)	RES 2,A	RES 3,B	RES 3,C	RES 3,D	RES 3,E	RES 3,H	RES 3,L	RES 3,(HL)	RES 3,A
A	RES 4,B	RES 4,C	RES 4,D	RES 4,E	RES 4,H	RES 4,L	RES 4,(HL)	RES 4,A	RES 5,B	RES 5,C	RES 5,D	RES 5,E	RES 5,H	RES 5,L	RES 5,(HL)	RES 5,A
B	RES 6,B	RES 6,C	RES 6,D	RES 6,E	RES 6,H	RES 6,L	RES 6,(HL)	RES 6,A	RES 7,B	RES 7,C	RES 7,D	RES 7,E	RES 7,H	RES 7,L	RES 7,(HL)	RES 7,A
C	SET 0,B	SET 0,C	SET 0,D	SET 0,E	SET 0,H	SET 0,L	SET 0,(HL)	SET 0,A	SET 1,B	SET 1,C	SET 1,D	SET 1,E	SET 1,H	SET 1,L	SET 1,(HL)	SET 1,A
D	SET 2,B	SET 2,C	SET 2,D	SET 2,E	SET 2,H	SET 2,L	SET 2,(HL)	SET 2,A	SET 3,B	SET 3,C	SET 3,D	SET 3,E	SET 3,H	SET 3,L	SET 3,(HL)	SET 3,A
E	SET 4,B	SET 4,C	SET 4,D	SET 4,E	SET 4,H	SET 4,L	SET 4,(HL)	SET 4,A	SET 5,B	SET 5,C	SET 5,D	SET 5,E	SET 5,H	SET 5,L	SET 5,(HL)	SET 5,A
F	SET 6,B	SET 6,C	SET 6,D	SET 6,E	SET 6,H	SET 6,L	SET 6,(HL)	SET 6,A	SET 7,B	SET 7,C	SET 7,D	SET 7,E	SET 7,H	SET 7,L	SET 7,(HL)	SET 7,A

AFTER ED

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	IN B,(C)	OUT (C),B	SBC HL,BC	LD (pq),BC	NEG	RETN	IM 0	LD I,A	IN C,(C)	OUT (C),C	ADC HL,BC	LD BC,(pq)	-	RETI	-	LD R,A
5	IN D,(C)	OUT (C),D	SBC HL,DE	LD (pq),DE	-	-	IM 1	LD A,I	IN E,(C)	OUT (C),E	ADC HL,DE	LD DE,(pq)	-	-	IM 2	LD A,R
6	IN H,(C)	OUT (C),H	SBC HL,HL	-	-	-	-	RRD	IN L,(C)	OUT (C),L	ADC HL,HL	-	-	-	-	RLD
7	-	-	SBC HL,SP	-	-	-	-	-	IN A,(C)	OUT (C),A	ADC HL,SP	LD SP,(pq)	-	-	-	-
8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
A	LDI	CPI	INI	OUTI	-	-	-	-	LDD	CPD	IND	OUTD	-	-	-	-
B	LDIR	CPIR	INIR	OTIR	-	-	-	-	LDDR	CPDR	INDR	OTDR	-	-	-	-
C	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
E	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
F	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-



APÊNDICE QUATRO

Esta tabela mostra, por uma ordem mais ou menos alfabética, todas as instruções *Z80* e ou os seus códigos hexadecimais ou as palavras «tabela 1» «tabela 2» ou «tabela 3». Neste caso, o código apropriado encontrar-se-á na tabela dada. Este apêndice indica também os *flags* alterados por cada instrução. Os símbolos utilizados para este efeito têm os seguintes significados:

–	O *flag* não é alterado pela instrução.
O	O *flag* é alterado da maneira que se poderia esperar.
0	O *flag* torna-se zero.
1	O *flag* torna-se um.
?	O *flag* é alterado de modo imprevisível.
X	Caso especial; será dada uma explicação.

INSTRUÇÕES		*FLAGS*								INSTRUÇÕES:		*FLAGS*							
Opcode	Hexcode	S	Z	–	H	–	P/V	N	C	cód. op.	cód. hex	S	Z	–	H	–	P/V	N	C
ADC A,r	*tabela* 1	O	O	–	O	–	O	0	O	HALT	76	–	–	–	–	–	–	–	–
ADC HL,s	*tabela* 2	O	O	–	O	–	O	0	O										
ADD A,r	*tabela* 1	O	O	–	O	–	O	0	O	IM 0	ED46	–	–	–	–	–	–	–	–
ADD HL,s	*tabela* 2	–	–	–	O	–	–	0	O	IM 1	ED56	–	–	–	–	–	–	–	–
ADD IX,s	*tabela* 2	–	–	–	O	–	–	0	O	IM 2	ED5E	–	–	–	–	–	–	–	–
ADD IY,s	*tabela* 2	–	–	–	O	–	–	0	O	INC r	*tabela* 1	O	O	–	O	–	O	0	–
AND r	*tabela* 1	O	O	–	1	–	O	0	0	INC s	*tabela* 2	–	–	–	–	–	–	–	–
										IN A,(n)	DBnn	–	–	–	–	–	–	–	–
BIT b,r	*tabela* 1	?	O	–	1	–	?	0	0	IN r,(C)	*tabela* 1	O	O	–	O	–	O	0	–
										INI	EDA2	?	x	–	?	–	?	1	–
CALL pq	CDqqpp	–	–	–	–	–	–	–	–	IND	EDAA	?	x	–	?	–	?	1	–
CALL c,pq	*tabela* 3	–	–	–	–	–	–	–	–	(Z torna-se um se B se tornar zero)									
CCF	3F	–	–	–	x	–	–	0	O	INIR	EDB2	?	1	–	?	–	?	1	–
(o *flag* H torna-se o valor anterior do *flag* C)										INDR	EDBA	?	1	–	?	–	?	1	–
CP r	*tabela* 1	O	O	–	O	–	O	1	O	JP pq	C3qqpp	–	–	–	–	–	–	–	–
CPI	EDA1	O	x	–	O	–	x	1	–	JP c,pq	*tabela* 3	–	–	–	–	–	–	–	–
CPD	EDA9	O	x	–	O	–	x	1	–	JP (HL)	E9	–	–	–	–	–	–	–	–
CPIR	EDB1	O	x	–	O	–	x	1	–	JP (IX)	DDE9	–	–	–	–	–	–	–	–
CPDR	EDB9	O	x	–	O	–	x	1	–	JP (IY)	FDE9	–	–	–	–	–	–	–	–
(P torna-se 0 se BC se tornar zero, Z torna-se 1 se A = (HL±1)										JR e	18ee	–	–	–	–	–	–	–	–
										JR c,e	*tabela* 3	–	–	–	–	–	–	–	–
CPL	2F	–	–	–	1	–	–	1	–										
										LD (BC),A	02	–	–	–	–	–	–	–	–
DAA	27	O	O	–	O	–	O	–	O	LD A,(BC)	0A	–	–	–	–	–	–	–	–
DEC r	*tabela* 1	O	O	–	O	–	O	1	–	LD (DE),A	12	–	–	–	–	–	–	–	–
DEC s	*tabela* 2	–	–	–	–	–	–	–	–	LD A,(DE)	1A	–	–	–	–	–	–	–	–
DI	F3	–	–	–	–	–	–	–	–										
DJNZ e	10ee	–	–	–	–	–	–	–	–	LD I,A	ED47	–	–	–	–	–	–	–	–
										LD R,A	ED4F	–	–	–	–	–	–	–	–
EI	FB	–	–	–	–	–	–	–	–	LD A,I	ED57	O	O	–	0	–	x	0	–
EX AF,AF'	08	–	–	–	–	–	–	–	–	LD A,R	ED5F	O	O	–	0	–	x	0	–
EX DE,HL	EB	–	–	–	–	–	–	–	–	(P/V toma o valor do flag IFF2)									
EX (SP),HL	E3	–	–	–	–	–	–	–	–										
EX (SP),IX	DDE3	–	–	–	–	–	–	–	–	LD SP,HL	F9	–	–	–	–	–	–	–	–
EX (SP),IY	FDE3	–	–	–	–	–	–	–	–	LD SP,IX	DDF9	–	–	–	–	–	–	–	–
EXX	D9	–	–	–	–	–	–	–	–	LD SP,IY	FDF9	–	–	–	–	–	–	–	–

TABELA DOIS

s	BC	DE	HL	SP	IX	IY
ADC HL,s	ED4A	ED5A	ED6A	ED7A	-	-
ADD HL,s	09	19	29	39	-	-
ADD IX,s	DD09	DD19	-	DD39	DD29	-
ADD IY,s	FD09	FD19	-	FD39	-	FD29
DEC s	0B	1B	2B	3B	DD2B	FD2B
INC s	03	13	23	33	DD23	FD23
LD s,nn	01nnnn	11nnnn	21nnnn	31nnnn	DD21nnnn	FD21nnnn
LD s,(pq)	ED4Bqqpp	ED5Bqqpp	2Aqqpp	ED7Bqqpp	DD2Aqqpp	FD2Aqqpp
LD (pq),s	ED43qqpp	ED53qqpp	22qqpp	ED73qqpp	DD22qqpp	FD22qqpp
POP s	C1	D1	E1	-	DDE1	FDE1
PUSH s	C5	D5	E5	-	DDE5	FDE5
SBC HL,s	ED42	ED52	ED62	ED72	-	-

INSTRUÇÕES		FLAGS
op cód.	cód. hex	S Z - H - P N C
LD r,r	tabela 1	- - - - - - - -
LD s,nn	tabela 2	- - - - - - - -
LD A,(pq)	3Aqqpp	- - - - - - - -
LD s,(pq)	tabela 2	- - - - - - - -
LD (pq),A	32qqpp	- - - - - - - -
LD (pq),s	tabela 2	- - - - - - - -
LDI	EDA0	- - - 0 - x 0 -
LDD	EDA8	- - - 0 - x 0 -
(P/V torna-se 0 se BC se tornar 0)		
LDIR	EDB0	- - - 0 - 0 0 -
LDDR	EDB8	- - - 0 - 0 0 -
NEG	ED44	● ● - ● - ● 1 ●
NOP	00	- - - - - - - -
OR r	tabela 1	● ● - 0 - ● 0 0
OUT (n),A	D3nn	- - - - - - - -
OUT (C),r	tabela 1	- - - - - - - -
OUTI	EDA3	? x - ? - ? 1 -
OUTD	EDAB	? x - ? - ? 1 -
(Z torna-se 1 se B se tornar zero)		
OTIR	EDB3	? 1 - ? - ? 1 -
OTDR	EDBB	? 1 - ? - ? 1 -
POP AF	F1	x x x x x x x x
(os *flags* são determinados pelo *byte* do topo da pilha)		
POP s	tabela 2	- - - - - - - -
PUSH AF	F5	- - - - - - - -
PUSH s	tabela 2	- - - - - - - -
RES b,r	tabela 1	- - - - - - - -
RET	C9	- - - - - - - -
RET c	tabela 3	- - - - - - - -
RETN	ED45	- - - - - - - -
RETI	ED4D	- - - - - - - -

INSTRUÇÕES		FLAGS
cód. op.	cód. hex	S Z - H - P N C
RES b,r	tabela 1	- - - - - - - -
RET	C9	- - - - - - - -
RET c	tabela 3	- - - - - - - -
RETN	ED45	- - - - - - - -
RETI	ED4D	- - - - - - - -
RLCA	07	- - - 0 - - 0 ●
RRCA	0F	- - - 0 - - 0 ●
RLA	17	- - - 0 - - 0 ●
RRA	1F	- - - 0 - - 0 ●
RLC r	tabela 1	● ● - 0 - ● 0 ●
RRC r	tabela 1	● ● - 0 - ● 0 ●
RL r	tabela 1	● ● - 0 - ● 0 ●
RR r	tabela 1	● ● - 0 - ● 0 ●
RRD	ED67	● ● - 0 - ● 0 -
RLD	ED6F	● ● - 0 - ● 0 -
RST 00	C7	- - - - - - - -
RST 08	CF	- - - - - - - -
RST 10	D7	- - - - - - - -
RST 18	DF	- - - - - - - -
RST 20	E7	- - - - - - - -
RST 28	EF	- - - - - - - -
RST 30	F7	- - - - - - - -
RST 38	FF	- - - - - - - -
SBC A,r	tabela 1	● ● - ● - ● 1 ●
SBC HL,s	tabela 2	● ● - ● - ● 1 ●
SCF	37	- - - 0 - - 0 1
SET b,r	tabela 1	- - - - - - - -
SLA r	tabela 1	● ● - 0 - ● 0 ●
SRA r	tabela 1	● ● - 0 - ● 0 ●
SRL r	tabela 1	● ● - 0 - ● 0 ●
SUB r	tabela 1	● ● - ● - ● 1 ●
XOR r	tabela 1	● ● - 0 - ● 0 0

TABELA UM

r	B	C	D	E	H	L	(HL)	A	(IX + d)	(IY + d)	n
ADD A,r	80	81	82	83	84	85	86	87	DD86dd	FD86dd	C6nn
ADC A,r	88	89	8A	8B	8C	8D	8E	8F	DD8Fdd	FD8Fdd	CEnn
AND r	A0	A1	A2	A3	A4	A5	A6	A7	DDA6dd	FDA6dd	E6nn
BIT 0,r	CB40	CB41	CB42	CB43	CB44	CB45	CB46	CB47	DDCBdd46	FDCBdd46	-
BIT 1,r	CB48	CB49	CB4A	CB4B	CB4C	CB4D	CB4E	CB4F	DDCBdd4E	FDCBdd4E	-
BIT 2,r	CB50	CB51	CB52	CB53	CB54	CB55	CB56	CB57	DDCBdd56	FDCBdd56	-
BIT 3,r	CB58	CB59	CB5A	CB5B	CB5C	CB5D	CB5E	CB5F	DDCBdd5E	FDCBdd5E	-
BIT 4,r	CB60	CB61	CB62	CB63	CB64	CB65	CB66	CB67	DDCBdd66	FDCBdd66	-
BIT 5,r	CB68	CB69	CB6A	CB6B	CB6C	CB6D	CB6E	CB6F	DDCBdd6E	FDCBdd6E	-
BIT 6,r	CB70	CB71	CB72	CB73	CB74	CB75	CB76	CB77	DDCBdd76	FDCBdd76	-
BIT 7,r	CB78	CB79	CB7A	CB7B	CB7C	CB7D	CB7E	CB7F	DDCBdd7E	FDCBdd7E	-
CP r	B8	B9	BA	BB	BC	BD	BE	BF	DDBEdd	FDBFdd	FEnn
DEC r	05	0D	15	1D	25	2D	35	3D	DD35dd	FD35dd	-
IN r,(C)	ED40	ED48	ED50	ED58	ED60	ED68	-	ED78	-	-	-
INC r	04	0C	14	1C	24	2C	34	3C	DD34dd	FD34dd	-
LD B,r	40	41	42	43	44	45	46	47	DD46dd	FD46dd	06nn
LD C,r	48	49	4A	4B	4C	4D	4E	4F	DD4Edd	FD4Edd	0Enn
LD D,r	50	51	52	53	54	55	56	57	DD56dd	FD56dd	16nn
LD E,r	58	59	5A	5B	5C	5D	5E	5F	DD5Edd	FD5Edd	1Enn
LD H,r	60	61	62	63	64	65	66	67	DD66dd	FD66dd	26nn
LD L,r	68	69	6A	6B	6C	6D	6E	6F	DD6Edd	FD6Edd	2Enn
LD (HL),r	70	71	72	73	74	75	-	77	-	-	36nn
LD A,r	78	79	7A	7B	7C	7D	7E	7F	DD7Edd	FD7Edd	3Enn
LD (IX + dl),r	DD70 dd	DD71 dd	DD72 dd	DD73 dd	DD74 dd	DD75 dd	-	DD77 dd	-	-	DD36 ddnn
LD (IY + dl),r	FD70 dd	FD71 dd	FD72 dd	FD73 dd	FD74 dd	FD75 dd	-	FD77 dd	-	-	FD36 ddnn
OR r	B0	B1	B2	B3	B4	B5	B6	B7	DDB6dd	FDB6dd	F6nn
OUT (C),r	ED41	ED49	ED51	ED59	ED61	ED69	-	ED79	-	-	-
RL r											
RES 0,r	CB80	CB81	CB82	CB83	CB84	CB85	CB86	CB87	DDCBdd86	FDCBdd86	-
RES 1,r	CB88	CB89	CB8A	CB8B	CB8C	CB8D	CB8E	CB8F	DDCBdd8E	FDCBdd8E	-
RES 2,r	CB90	CB91	CB92	CB93	CB94	CB95	CB96	CB97	DDCBdd96	FDCBdd96	-
RES 3,r	CB98	CB99	CB9A	CB9B	CB9C	CB9D	CB9E	CB9F	DDCBdd9E	FDCBdd9E	-
RES 4,r	CBA0	CBA1	CBA2	CBA3	CBA4	CBA5	CBA6	CBA7	DDCBddA6	FDCBddA6	-
RES 5,r	CBA8	CBA9	CBAA	CBAB	CBAC	CBAD	CBAE	CBAF	DDCBddAE	FDCBddAE	-
RES 6,r	CBB0	CBB1	CBB2	CBB3	CBB4	CBB5	CBB6	CBB7	DDCBddB6	FDCBddB6	-
RES 7,r	CBB8	CBB9	CBBA	CBBB	CBBC	CBBD	CBBE	CBBF	DDCBddBE	FDCBddBE	-
RLC r	CB00	CB01	CB02	CB03	CB04	CB05	CB06	CB07	DDCBdd06	FDCBdd06	-
RRC r	CB08	CB09	CB0A	CB0B	CB0C	CB0D	CB0E	CB0F	DDCBdd0E	FDCBdd0E	-
RL r	CB10	CB11	CB12	CB13	CB14	CB15	CB16	CB17	DDCBdd16	FDCBdd16	-
RR r	CB18	CB19	CB1A	CB1B	CB1C	CB1D	CB1E	CB1F	DDCBdd1E	FDCBdd1E	-
SET 0,r	CBC0	CBC1	CBC2	CBC3	CBC4	CBC5	CBC6	CBC7	DDCBddC6	FDCBddC6	-
SET 1,r	CBC8	CBC9	CBCA	CBCB	CBCC	CBCD	CBCE	CBCF	DDCBddCE	FDCBddCE	-
SET 2,r	CBD0	CBD1	CBD2	CBD3	CBD4	CBD5	CBD6	CBD7	DDCBddD6	FDCBddD6	-
SET 3,r	CBD8	CBD9	CBDA	CBDB	CBDC	CBDD	CBDE	CBDF	DDCBddDE	FDCBddDE	-
SET 4,r	CBE0	CBE1	CBE2	CBE3	CBE4	CBE5	CBE6	CBE7	DDCBddE6	FDCBddE6	-
SET 5,r	CBE8	CBE9	CBEA	CBEB	CBEC	CBED	CBEE	CBEF	DDCBddEE	FDCBddEE	-
SET 6,r	CBF0	CBF1	CBF2	CBF3	CBF4	CBF5	CBF6	CBF7	DDCBddF6	FDCBddF6	-
SET 7,r	CBF8	CBF9	CBFA	CBFB	CBFC	CBFD	CBFE	CBFF	DDCBddFE	FDCBddFE	-
SUB A,r	90	91	92	93	94	95	96	97	DD96dd	FD96dd	D6nn
SBC A,r	98	99	9A	9B	9C	9D	9E	9F	DD9Edd	FD9Edd	DEnn
SLA r	CB20	CB21	CB22	CB23	CB24	CB25	CB26	CB27	DDCBdd26	FDCBdd26	-
SRA r	CB28	CB29	CB2A	CB2B	CB2C	CB2D	CB2E	CB2F	DDCBdd2E	FDCBdd2F	-
SRL r	CB38	CB39	CB3A	CB3B	CB3C	CB3D	CB3E	CB3F	DDCBdd3E	FDCBdd3E	-
XOR r	A8	A9	AA	AB	AC	AD	AE	AF	DDAEdd	FDAEdd	EEnn

TABELA TRÊS

c	NZ	Z	NC	C	PO	PE	P	M
CALL c,pq	C4qqpp	CCqqpp	D4qqpp	DCqqpp	E4qqpp	ECqqpp	F4qqpp	FCqqpp
JP c,pq	C2qqpp	CAqqpp	D2qqpp	DAqqpp	E2qqpp	EAqqpp	F2qqpp	FAqqpp
JRc,e	20ee	28ee	30ee	38ee	-	-	-	-
RET c	C0	C8	D0	D8	E0	E8	F0	F8

APÊNDICE CINCO

Cada carácter de controle – isto é, cada carácter cujo código se encontre entre 00 e 1F – tem dois sentidos diferentes: o controle INPUT (como é lido por INKEY$) e o controle OUTPUT (como é dado por PRINT). Por vezes estes dois coincidem (exemplo, OD) mas, em geral, isto não se passa. Esta tabela apresenta as funções tanto em INPUT como em OUTPUT de cada um dos caracteres de controle.

Abreviaturas: bs significa «ambos os *shifts* juntos seguidos de.. »
cs significa «*caps shift* simultaneamente com...»
ss significa «*symbol shift* simultaneamente com...»

CÓDIGO	CONTROLE INPUT		CONTROLE OUTPUT
	COMO SE OBTÉM	SIGNIFICADO	ESCRITO COMO
00	bs cs 8	*flash* «desligado»	?
01	bs cs 9	*flash* «ligado»	?
02	bs 8	brilho «desligado»	?
03	bs 9	brilho «ligado»	?
04	cs 3	cores «verdadeiras»	?
05	cs 4	cores «invertidas»	?
06	cs 2	*caps lock* (só maiúsc.)	controle de vírgula ††
07	cs 1	*edit*	?
08	cs 5	cursor à esquerda	*backspace*
09	cs 8	cursor à direita	cores correntes
0A	cs 6	cursor para baixo	?
0B	cs 7	cursor para cima	?
0C	cs 0	*delete*	?
0D	enter	*enter*	*newline* (ou CR)
0E†	bs	controle-E	?
0F	cs 9	gráfico	?
10	bs 0	fundo preto	controle INK*
11	bs 1	fundo azul	controle PAPER*
12	bs 2	fundo vermelho	controle FLASH*
13	bs 3	fundo magenta	controle BRIGHT*
14	bs 4	fundo verde	controle INVERSE*
15	bs 5	fundo *cyan*	controle OVER*
16	bs 6	fundo amarelo	controle AT**
17	bs 7	fundo branco	controle TAB**
18	bs cs 0	tinta preta	?
19	bs cs 1	tinta azul	?
1A	bs cs 2	tinta vermelha	?
1B	bs cs 3	tinta magenta	?
1C	bs cs 4	tinta verde	?
1D	bs cs 5	tinta *cyan*	?
1E	bs cs 6	tinta amarela	?
1F	bs cs 7	tinta branca	?

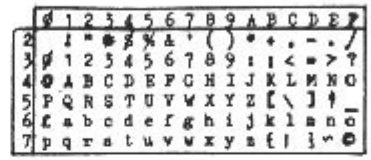
† O *byte* OE, quando ocorre numa listagem de programa BASIC, é interpretado como significando «seguem-se agora cinco *bytes* de dados». O *byte* OE e os primeiros cinco *bytes* imediatamente a seguir tornar-se-ão invisíveis numa listagem.

†† O *byte* 06 (controle de vírgula) pode entrar directamente a partir do teclado, fazendo-se bs 6 cs 0 (fundo amarelo seguido de DELETE).

* Necessita de um *byte* de dados adicional a seguir.

** Necessita de dois *bytes* de dados adicionais a seguir.

Os caracteres cujos códigos se encontram entre 20 e 7F são o conjunto de caracteres ASCII *standard**. Estes são listados na tabela abaixo. Note que o símbolo correspondente ao código 20 é *space*.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	_
6	£	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	©

Os códigos restantes são códigos gráficos ou *tokens* que escrevem palavras completas. São os que se seguem:

80	gráfico	espaço
81	gráfico	1
82	gráfico	2
83	gráfico	3
84	gráfico	4
85	gráfico	5
86	gráfico	6
87	gráfico	7
88	gráfico	tecla 7
89	gráfico	tecla 6
8A	gráfico	tecla 5
8B	gráfico	tecla 4
8C	gráfico	tecla 3
8D	gráfico	tecla 2
8E	gráfico	tecla 1
8F	gráfico	tecla 8
90	gráfico	A
91	gráfico	B
92	gráfico	C
93	gráfico	D
94	gráfico	E
95	gráfico	F
96	gráfico	G
97	gráfico	H
98	gráfico	I
99	gráfico	J
9A	gráfico	K
9B	gráfico	L
9C	gráfico	M
9D	gráfico	N
9E	gráfico	O
9F	gráfico	P
A0	gráfico	Q
A1	gráfico	R
A2	gráfico	S
A3	gráfico	T
A4	gráfico	U

*Note-se que os códigos 60 e 7F são não *standard*.

A5	RND
A6	INKEY$
A7	PI
A8	FN
A9	POINT
AA	SCREEN$
AB	ATTR
AC	AT
AD	TAB
AE	VAL$
AF	CODE
B0	VAL
B1	LEN
B2	SIN
B3	COS
B4	TAN
B5	ASN
B6	ACS
B7	ATN
B8	LN
B9	EXP
BA	INT
BB	SQR
BC	SGN
BD	ABS
BE	PEEK
BF	IN
C0	USR
C1	STR$
C2	CHR$
C3	NOT
C4	BIN
C5	OR
C6	AND
C7	<=
C8	>=
C9	<>
CA	LINE
CB	THEN
CC	TO
CD	STEP
CE	DEF FN
CF	CAT
D0	FORMAT
D1	MOVE

D2	ERASE	
D3	OPEN	
D4	CLOSE	
D5	MERGE	
D6	VERIFY	
D7	BEEP	
D8	CIRCLE	
D9	INK	
DA	PAPER	
DB	FLASH	
DC	BRIGHT	
DD	INVERSE	
DE	OVER	
DF	OUT	
E0	LPRINT	
E1	LLIST	
E2	STOP	
E3	READ	
E4	DATA	
E5	RESTORE	
E6	NEW	tecla A
E7	BORDER	tecla B
E8	CONTINUE	tecla C
E9	DIM	tecla D
EA	REM	tecla E
EB	FOR	tecla F
EC	GO TO	tecla G
ED	GO SUB	tecla H
EE	INPUT	tecla I
EF	LOAD	tecla J
F0	LIST	tecla K
F1	LET	tecla L
F2	PAUSE	tecla M
F3	NEXT	tecla N
F4	POKE	tecla O
F5	PRINT	tecla P
F6	PLOT	tecla Q
F7	RUN	tecla R
F8	SAVE	tecla S
F9	RANDOMIZE	tecla T
FA	IF	tecla U
FB	CLS	tecla V
FC	DRAW	tecla W
FD	CLEAR	tecla X
FE	RETURN	tecla Y
FF	COPY	tecla Z

APÊNDICE SEIS

A tabela seguinte dá os códigos a usar depois de RST 28, utilizados para manipular a pilha do calculador. Só estão aqui apresentados os códigos que foram anteriormente mencionados neste livro, já que alguns dos outros exigiriam explicações demasiado prolongadas.

01	exchange	Troca os dois itens no topo da pilha.
02	delete	Apaga o item do topo da pilha.
03	subtract	Apaga os dois itens do topo da pilha e «empilha» o resultado da sua subtracção.
04	multiply	Apaga os dois itens do topo da pilha e «empilha» o seu produto.
05	divide	Apaga os dois itens do topo da pilha e «empilha» o resultado da sua divisão.
06	power	Apaga os dois itens do topo da pilha e «empilha» o resultado de elevar um à potência do outro.
0F	add	Apaga os dois itens do topo da pilha e «empilha» a sua soma.
17	s_add	Como 0F mas soma *strings* e não números.
18	val$	Substitui o item do topo da pilha por VAL$ desse item.
19	usr_s	Substitui o item do topo (uma *string*) por USR dessa *string*.
1B	negate	Substitui o item do topo (um número) pelo simétrico desse número.
1C	code	}
1D	val	} Substitui o item do topo (uma *string*) pelo resultado da função apropriada aplicada a essa
1E	len	} *string*.
1F	sin	}
20	cos	} Substitui o item do topo (um número) pelo resultado da função apropriada aplicada a esse
21	tan	} número.
22	asn	}
23	acs	}
24	atn	}
25	ln	}
26	exp	}
27	int	}
28	sqr	}
29	sgn	}
2A	abs	}
2B	peek	}
2C	in	}
2D	usr_n	}
2E	str$	}
2F	chr$	}
31	duplicate	Empilha uma cópia do item do topo da pilha.
38	end_calc	Último *byte* na sequência; retorna ao código máquina normal.
A0	const_0	Empilha o número zero.
A1	const_1	Empilha o número um.
A2	const_½	Empilha o número zero ponto cinco.
A3	const_π/2	Empilha o número PI/2 (1.5707963).
A4	const_10	Empilha o número dez.
Cn	store_n	Faz a memória n igual ao item no topo da pilha.
En	recall_n	Empilha o conteúdo da memória n.
