Este segundo volume sobre o código-máquina, da autoria de um estudante de dezassete anos, é dedicado ao leitor que já possui noções básicas desta linguagem e pretende aprofundar os seus conhecimentos. São aqui longamente tratados aspectos como os saltos, *flags,* chamadas de subrotinas e outras instruções, assim como os *bits,* operações lógicas, *bytes,* produção de som, utilização do visor e outros. No final, foram incluídos apêndices com instruções para o Z80 assim como mnemónicas, para comodidade do utilizador.
Munido destes valiosos conhecimentos, o leitor poderá partir à descoberta de novos mundos, dentro do seu Spectrum!

EDITORIAL PRESENÇA

Código de Máquina para Programadores Avançados
162

PAUL HOLMES

CÓDIGO DE MÁQUINA PARA PROGRAMADORES AVANÇADOS

CULTURA E TEMPOS LIVRES

1. ABC do Xadrez, *Petar Trifunovitch* e *Sava Vukovitch*
4. ABC do Bridge, *Pierre Jais* e *H. Lahana*
5. ABC Prático de Fotografia, *W. D. Emanuel*
6. ABC do Judo, *E. J. Harrison*
7. Como Fazer Cinema, *Paul Petzold*
8. Bridge Moderno, *Pierre Jais* e *H. Lahana*
9. Fotografia – Técnicas e Truques I, *Edwin Smith*
10. ABC dos Estilos, da Arquitectura ao Mobiliário, *A. Aussel*
11. Fotografia – Técnicas e Truques II, *Edwin Smith*
12. A Pesca Submarina, *António Ribera*
13. Teoria dos Finais de Partida, *Yuri Averbach*
14. Aprenda Rádio, *B. Fighiera*
15. Guia do Cão, *Louise Laliberté-Robert* e *Jean-Pierre Robert*
16. ABC do Aquário, *Anthony Evans*
17. Iniciação à Electricidade e Electrónica, *Fernand Huré*
18. Os Transistores, *Fernand Huré*
19. Karaté I, *Albrecht Pflüger*
20. Iniciação ao Radiocomando dos Modelos Reduzidos, *C. Péricone*
21. Construa o seu Receptor, *B. Fighiera*
22. Montagens Electrónicas, *B. Fighiera,*
23. O Berbequim Eléctrico, *Villy Dreier*
24. Cactos, *J. Nilaus Jensen*
25. Iniciação à Alta Fidelidade, *Peter Turner*
26. O Aquário de Água Doce, *Paulo de Oliveira*
27. ABC do Ténis, *Fonseca Vaz*
28. Karaté II, *Albrecht Pflüger*
29. ABC da Criação de Canários, *Curt Af Enehjelm*
30. Ginástica Feminina, *Sonja Helmer Jensen*
31. Cartomancia, *Rhea Koch*
32. Calculadoras Electrónicas de Bolso, *E. Dam Ravn*
33. O Pastor Alemão, *Gilles Legrand*
34. Xadrez – Teoria do Meio Jogo, *I. Bondarevsky*
35. Manual do Super 8 – I, *Myron A. Matzkin*
36. ABC da Criação de Periquitos, *Cyril H. Rogers*
37. O Livro dos Gatos, *Bärbel Gerber* e *Horst Bielfeld*
38. Manual do Super 8 – II, *Myron A. Matzkin*
39. ABC do Mergulho Desportivo, *Walter Mattes*
40. Circuitos Integrados/Aplicações Práticas, *F. Bergtold*
41. A Apicultura, *H. R. C. Riches*
42. ABC do Cultivo de Plantas, *H. G. Witham Fogg*
43. ABC da Criação de Pombos, *Kai R. Dahl*
44. Construção de Caixas Acústicas de Alta Fidelidade, *R. Brault*
45. Raças de Canários, *Klaus Speicher*
46. Jogos de Cartas, *Graciano Dolma*
47. Cocker Spaniels, *H. S. Lloyd*
48. ABC da Caça, *Fabián Abril*
49. Aprenda Televisão, *Gordon J. King*
50. Iniciação à Pesca, *Juan Nadal*
51. Basquetebol, *Marius Norregard*
52. Cães de Caça, *Santiago Pons*
53. Aprenda Electrónica, *T. L. Squires* e *C. M. Deason*
54. A Avicultura, *Jim Worthington*
55. A Produção de Coelhos, *P. Surdeau* e *R. Henaff*
56. ABC dos Computadores, *T. F. Fry*
57. Natação para Crianças, *John Idorn*
58. O Boxer, *Anni Mortensen*
59. Voleibol, *Ole Hasen* e *Per-Göran Person*
60. Iniciação à Vela, *Donald Law*
61. ABC da Filatelia, *Jacqueline Caurat*
62. A Pesca à Beira-Mar, *J.-M. Boëlle* e *D. Doyen*
63. Enxerto de Árvores de Fruto, *Alejo Rigau*
64. A Cultura do Morangueiro, *Luís Alsina Grau.*
65. Emissores-Receptores (Walkies-Talkies), *P. Duranton*
66. Iniciação à Fotoelectrónica, *Heinz Richter*
67. Doces e Conservas de Fruta, *Robin Howe*
68. A Criação de Hamsters, *C. F. Snow*
69. A Criação de Porcos, *Roy Genders*
70. Calendário do Horticultor, *Luís Alsina Grau*
71. Jogos Electrónicos, *F. G. Rayer*
72. Cultivo de Cogumelos e Trutas, *Alejo Rigau*
73. Aprenda Televisão a Cores, *Gordon J. King*
74. Gravação em Fita Magnética, *Ian R. Sinclair*
75. Poda de Árvores e Arbustos, *Roy Genders*
76. Como Treinar o Seu Cão, *E. Fitch Daglish*
77. Instrumentos de Medida e Verificação, *Heinrich Stöckle*
78. A Criação de Caracóis, *Matías Josa*
79. Rádio – Fundamentos e Técnicas, *Gordon J. King*
80. Como fazer Gelados, *Sylvie Thiébault*
81. Iniciação à Jardinagem, *Noel Clarasó*
82. A Congelação dos Alimentos, *Suzanne Lapointe*
83. Windsurf – Prancha à Vela, *Ernstfried Prade*
84. Raças de Cães, *O. Hasselfeldt*
85. Rummy e Canasta, *Claus D. Grupp*
86. A Encadernação, *Annie Persuy*
87. Aprenda Electricidade, *Heinz Richter*
88. Taxidermia, Embalsamamento de Aves e Mamíferos, *Harry Hjortaa*
89. Jogging – Correr Para Manter a Forma, *Werner Sonntag*
90. ABC da Cozinha Chinesa, *Sonya Richmond*
91. Jogos TV, *C. Tavernier*
92. Amplificadores de Som, *Richard Zierl*
93. O Livro do Poker, *Claus D. Grupp*
94. Aprenda a Desenhar, *Rose-Marie de Prémont* e *Nicole Philippi*
95. O Minitrampolim na Escola, *Sonja Helmer Jensen* e *Klaus Dano*
96. Jogos de Luzes e Efeitos Sonoros para Guitarras, *B. Figuiera*
97. O Cultivo de Tomate, *Louis N. Flawn*
98. Pilhas Solares, *F. Juster*
99. A Criação Doméstica de Coelhos, *C. F. Snow*
100. Iniciação ao Futebol, *Wieland Männle* e *Heinz Arnold*
101. Horóscopos Chineses, *Georg Haddenbach*
102. Guia Prático de Marcenaria, *Charles H. Hayward*
103. Andebol, *Fritz* e *Peter Hattig*
104. Dispositivos Anti-Roubo, *H. Schreiber*
105. Perus, Pintadas e Codornizes, *Jerome Sauze*
106. Crepes – Doces e Salgados, *Florence Arzel*
107. Aperitivos e Entradas, *Myrette Tiano*
108. Ténis de Mesa, *Leslie Woollard*
109. Aprenda Surf, *R. Abbott* e *M. Baker*
110. Futebol – Técnica e Táctica, *Kurt Lavall*
111. A Vaca Leiteira, *Colin T. Whittemore*
112. O Cubo Mágico, *Josef Trajber*
113. O Perdigueiro Português, *José M. Correia*
114. Pizzas e Massas à Italiana, *Marieanne Ränk*
115. O Cubo Para Quem Já o Faz, *Josef Trajber*
116. A Pirâmide Mágica, A Torre, O Barril do Diabo, *M. Mrowka-W. J. Weber*
117. Gansos e Patos, *Marie Mourthe*
118. Iniciação ao Kung-Fu, *A. P. Harrington*
119. Electrónica e Fotografia, *Hans-Peter Siebert*
120. O Livro da Fortuna, *Douglas Hill*
121. Construção de um Alimentador de Corrente, *Waldemar Battinger*
122. Hóquei em Patins, *Francisco Velasco*
123. Técnicas de Tiro, *Anton Kovacic*
124. Aprenda a Tricotar, *Uta Mix*
125. ABC da Patinagem, *Christa-Maria* e *Richard Kerler*
126. A Pesca e os seus Segredos, *Armand Deschamps*
127. O Osciloscópio, *R. Rateau*
128. Guia Prático da Banda do Cidadão, *T. M. Normand*
129. Sumos e Batidos, *Manfred Donderski*
130. Introdução à Programação de Microcomputadores, *Peter C. Sanderson*
131. Aprenda Croché, *Uta Mix*

132. ABC do Microprocessador, *P. Mélusson*
133. Guia Prático de Basic, *Roger Hunt*
134. Introdução à Electrónica Digital, *Ian R. Sinclair*
135. ABC do Video, *David Matthewson*
136. Fotografia em Movimento, *Dom Morley*
137. Guia Prático de Cobol, *Ray Welland*
138. Fotografia a Pequena Distância, *Sidney F. Ray*
139. Guia Prático da Canaricultura, *Manuel Gonçalves*
140. Minielectrónica para Amadores, *Heinz Richter*
141. ABC da Programação de Computadores, *John Shelley*
142. TAROT – O Futuro Pelas Cartas, *Edwin J. Nigg*
143. ABC da Equitação, *Dorothy Johnson*
144. Como Programar o seu ZX 81, *Patrick Gueulle*
145. 100 Avarias TV e a Maneira Prática de as Detectar, *P. Duranton*
146. ABC da Horticultura, *Louis Giordano*
147. Basic Para Microcomputadores, *A. P. Stephenson*
148. Como Programar o seu ZX Spectrum, *Tim Hartnell* e *Dilwyn Jones*
149. Iniciação aos Motores Diesel, *David S. Maclean*
150. 60 Jogos Para o ZX Spectrum, *David Harwood*
151. As Linhas da Mão, *Rose Hubert*
152. Cozinha Italiana, *Rotraud Degner*
153. Manual do ZX Spectrum, *R. J. Simpson* e *T. J. Terrell*
154. Z80 Assembler Para o Spectrum – Iniciação ao Código de Máquina, *João Paulo Fragoso*
155. Aeróbica, *Hans Shulz*
156. ABC do Atletismo, *Denis Watts*
157. 26 Programas Basic para Microcomputadores, *Derrick Daines*
158. Aprenda Pascal no Seu Microcomputador, *Jeremy Ruston*
159. Guia Moderno da Suinicultura, *Colin Whittemore*
160. O Bar em Sua Casa – 888 Cocktails, *Aladar von Wesendonk*
161. Código de Máquina Para Principiantes, *James Walsh*
162. Código de Máquina Para Programadores Avançados, *Paul Holmes*

PAUL HOLMES

CÓDIGO DE MÁQUINA PARA PROGRAMADORES AVANÇADOS

EDITORIAL PRESENÇA

Título original
SPECTRUM MACHINE CODE MADE EASY:
VOLUME TWO FOR ADVANCED PROGRAMMERS

Tradução de Conceição Jardim e Eduardo Nogueira
capa de António Marques

EDITORIAL PRESENÇA, LDA.
Rua Augusto Gil, 35-A — 1000 LISBOA

SOBRE O AUTOR

Paul Holmes é um estudante de dezassete anos de idade que começou a dedicar-se à informática há cerca de dois anos usando um ZX80. Desde então escreveu dois conjuntos de programas, para o ZX81, de auxiliares de programação e auxiliares gráficos, comercializados pela JRS Software em Inglaterra e pela Softsync nos Estados Unidos. Durante algum tempo fez críticas de programas na ZX Computing. Mais recentemente escreveu outros programas comerciais, e está a preparar um livro sobre jogos para o ZX Spectrum.

AGRADECIMENTOS

Gostaria de apresentar os meus cumprimentos a:

Tim Hartnell, que primeiro sugeriu a ideia deste livro.

Richard Lawrence, que fez nascer o meu interesse pela informática.

Liz North, que me trouxe o almoço quando o meu trabalho me fez esquecer a hora da refeição.

Sr. Coulson, o meu professor de física, que me apoiou constantemente enquanto escrevi o livro.

Dedico este livro aos meus amigos em Sutton Coldfield.

INTRODUÇÃO

O leitor resolveu portanto aprender código-máquina. Tomou conhecimento de alguns aspectos básicos, e agora quer aprofundar o que sabe. Bom, espero que tenha pegado no livro certo; de facto parti do princípio de que já possui alguns conhecimentos básicos neste campo. Gostaria, porém, de dizer algumas palavras àqueles que nada conhecem de código-máquina mas que se arriscaram a adquirir este livro. Como já fiz notar, a obra é dedicada aos que já possuem alguns conhecimentos básicos, mas se não é esse o seu caso experimente ler a primeira parte do capítulo um; se não lhe parecer difícil de digerir prossiga. Se no entanto o seu cérebro lhe começa a enviar "mensagens de erro" semelhantes às que já se habituou a ver no Spectrum, do tipo "Sem sentido em português", talvez seja melhor familiarizar-se primeiro com algumas questões básicas em obras como o primeiro volume, da autoria de James Walsh; este autor parte do princípio e conduz lentamente o leitor às noções gerais que fundamentam o uso do código-máquina. Se já o leu, respire fundo e mergulhe no primeiro capítulo deste segundo volume. Se está numa livraria, compre o livro ou deixe-o onde está — afinal de contas, onde julga que está? Numa biblioteca?

I

COMO VIAJAR DENTRO DO SPECTRUM

Antes de começarmos a escrever programas em código-máquina, convirá talvez saber o que se vai passar nos capítulos seguintes e definir o melhor modo de usar este livro.

Como já disse na introdução, esta obra não é pensada para quem ainda nada conhece de código-máquina, e sim para aqueles que já têm sobre ele algumas noções básicas. Para o caso de o leitor querer recordar estas noções, vamos fazer um breve resumo daquilo que convirá saber.

O código-máquina, ou linguagem-máquina, é uma linguagem de programação que a CPU (Unidade Processadora Central) do computador é capaz de entender. Consiste numa série de números que a CPU transforma em acções. Um exemplo desta linguagem são os programas contidos na ROM do seu Spectrum, ou seja, na parte inicial da memória onde se encontram as rotinas que organizam este computador. Esta ROM compreende as instruções Basic, tradu-las e executa rotinas em código-máquina que correspondem a funções Basic como a PRINT e a GO TO.

O código-máquina é uma linguagem muito mais próxima da máquina, muito mais rápida do que a Basic e nada tem a ver com a ROM; a pastilha integrada Z80 que forma a CPU do Spectrum é a única parte do computador que trata o código-máquina. Este código, apesar de guardado na memória, possui nomes apropriados para cada instrução, pelo que nos é fácil ter uma ideia do que cada número «faz». A esses nomes chamamos «mnemónicas». O Z80 possui um certo número de registos, sendo em torno a estes

que a linguagem se estrutura. Os registos em causa são usados do mesmo modo que as variáveis em Basic. Existem dois conjuntos de registos «normais», e alguns «especiais» que podemos considerar equivalentes às variáveis de sistema já conhecidas da Basic.

Conjunto normal de registos

A	F
B	C
D	E
H	L
IX	
IY	
SP	
IV	
R	

A'	F'
B'	C'
D'	E'
H'	L'

Conjunto alternativo de registos

Cada caixa contém um registo. As caixas mais pequenas podem guardar números entre 0 e 255, e as maiores entre 0 e 65535. Se juntarmos duas caixas adjacentes, podemos criar caixas grandes. Por exemplo, se juntarmos B e C obteremos um «par de registos» chamado BC. Só podemos utilizar os registos A, B, C, D, E, H, L, os seus equivalentes do conjunto alternativo de registos, IX e IY. Os outros registos são utilizados pela própria CPU Z80.

Por outro lado só podemos usar um dos conjuntos alternativos de registos de cada vez. A instrução:

```
EXX
```

permite-nos trocar de conjunto de registos.

Para atribuir um valor a uma variável Basic dizemos:

```
LET A = 20
```

Em código-máquina usamos em vez disto a palavra «load» (''carregar''), abreviada para LD, e dizemos:

```
LD A,14
```

O número aqui usado não é um número decimal, mas sim hexadecimal, ou seja, corresponde a um sistema de números de

«base 16». Nesta base hexadecimal, o número 14 corresponde ao decimal 20. Com efeito, 1 × 16 + 4 é igual a 20. O algarismo da direita indica as unidades, e o da esquerda grupos de 16 unidades, enquanto que no sistema decimal o da esquerda indica grupos de dez unidades. Se utilizarmos um número de dois bytes, o algarismo que se segue (3.º a contar da direita) corresponde a grupos de 255 unidades, e o 4.º algarismo indica grupos de 4096 unidades. Nesta base usamos portanto, como unidades, os algarismos 0 a 15. Necessitamos obviamente de uma forma de indicar os algarismos 10 a 15; usamos para tal as primeiras letras do alfabeto. Segue-se uma tabela dos primeiros 20 números em hexadecimal e em decimal.

Hexadecimal	Decimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15
10	16
11	17
12	18
13	19
14	20

Podem-se guardar dois algarismos hexadecimais em cada byte de memória, ou seja, números correspondentes de 0 a 255 em decimal.

Os registos permitem-nos realizar bastantes operações. Dis-

pomos de instruções como a LD para definir um valor, e outras como ADD (''somar'') para somar o conteúdo de dois registos. Os programas são guardados, como já disse, sob a forma de uma série de códigos. Por exemplo, a instrução ''Carregar dado em A'' é representada por dois números hexadecimais:

LD A,d ———— 3E d

O primeiro número é 3E, em hexadecimal, que indica o facto de querermos carregar um valor em A; o segundo, ''d'', designa o número de um byte que queremos carregar neste registo. Se quisermos carregar um par de registos, por exemplo o BC, devemos indicar números de dois bytes, mas estes são normalmente representados em código-máquina pela ordem inversa:

LD BC, 15 6A ———— 01 6A 15

O primeiro valor, 01, indica que vamos carregar um valor em BC; o segundo é o byte ''menos significativo'' do dado, isto é, os dois algarismos que normalmente seriam escritos do lado direito; o terceiro número corresponde ao byte ''mais significativo'', ou seja, aos dois algarismos que normalmente seriam escritos à esquerda.

Existem muitas outras instruções, sendo possível encontrar um resumo destas no Apêndice D. O leitor já deve conhecer as instruções de carga (''load''), as instruções aritméticas como a ADD (somar) e SUB (subtrair), a instrução RET (retorno no final da execução da rotina em código-máquina) e ter alguns conhecimentos sobre o ''stack'' se bem que este venha a ser referido mais adiante.

Se descobrir num programa uma instrução que não compreende e não for explicada suficientemente ao longo do livro, poderá encontrar a sua descrição no Apêndice A.

Ao longo do livro utilizei um programa hexadecimal, em Basic, para introduzir programas em código-máquina no computador. Se possuir um dos programas de montagem (auxiliares para desenvolvimento de programas) que existem no mercado, use-o. Todas as listagens existentes no livro indicam os endereços e os cgdbgo, a cxrregar neles. Note por outro lado que usamos a base hexadecimal em todas as listagens, mas que no texto que se segue

aos números se encontra normalmente ''hex'' ou ''decimal'' para esclarecer qual o tipo de número usado. Se possui um programa Assembler pode introduzir na máquina os programas aqui apresentados em mnemónicas, mas verifique se o seu Assembler aceita as mnemónicas ''standard'' para o Z80 e pode convertê-las em código-máquina carregado a partir dos endereços indicados.

No segundo capítulo começaremos por estudar os vários métodos de executar saltos no interior de um programa. Em seguida estudaremos o modo como o computador pode tomar decisões, indicando como exemplo um programa para produção de caracteres com o dobro da altura normal. No capítulo três estudaremos os bits que formam cada byte, o modo de actuar sobre eles e de verificar o estado em que se encontram, e estuadaremos como exemplo o ficheiro de atributos.

Terminaremos o capítulo três com um programa que permite ao Spectrum transformar-se numa máquina de escrever ''inteligente''. O capítulo quatro trata das instruções lógicas XOR, AND e OR, usando novamente o ficheiro de atributos como exemplo. O capítulo seguinte estudará as instruções de rotação e os seus usos; e referirá os decimais codificados em binário e o modo de os usar com instruções do Z80.

O capítulo sexto é dedicado aos ''ports'' de entrada/saída, e discute o modo como um computador comunica com o mundo exterior e as maneiras de nos servirmos desse conhecimento. Dedicaremos algum tempo a estudar a produção de efeitos sonoros pelo Spectrum. O último capítulo é dedicado a um tópico directamente relacionado com o hardware — o sistema de interrupção. Este é explicado recorrendo ao auxílio de material presente na ROM, sendo descritos os dois tipos de interrupções e os três modos que estas podem assumir.

No final do livro encontram-se quatro apêndices que listam números decimais e hexadecimais, os respectivos códigos ASCII e as mnemónicas Z80; o modo como as instruções Z80 influem sobre as ''flags''; as variáveis de sistema, dando alguns esclarecimentos sobre elas; e finalmente as definições de todos os tipos de instruções usadas pelo microprocessador Z80.

Depois de fazer esta breve resenha do que virá a ser tratado no livro, podemos passar ao nosso primeiro tópico: a execução de saltos em código-máquina, um aspecto essencial de qualquer programa nesta linguagem.

Quando se programa em Basic é muitas vezes necessário enviar a execução para uma parte diferente daquela onde nos encontramos. É fácil fazê-lo recorrendo à declaração GOTO. Esta instrução não necessita sequer de ser seguida de um número, que pode ser substituído por uma expressão. Esta possibilidade de empregar uma expressão como

```
GO TO 10 * A + 100
```

permite-nos executar saltos dentro de um programa de uma forma bastante versátil. A palavra "salto" é de facto mais apropriada a uso em código-máquina do que em Basic. Mas é de qualquer modo uma palavra suficientemente boa para qualquer linguagem que nos permita "mover-nos" de um ponto de um programa para outro.

Em código-máquina tudo se passa de um modo levemente diferente. Não existem "expressões", mas dispomos de dois tipos diferentes de "instrução" para dizer ao microprocessador do Spectrum (um Z80-A, como deveria saber) que passe para outro ponto de um programa.

Mas como é habitual o código-máquina parece dispor de soluções mais complicadas para tudo. Comecemos portanto por estudar o nosso caro amigo Contador de Programa (PC), a fim de tornar mais claras algumas explicações futuras.

No interior de uma CPU (unidade processadora central) Z80 existem muitos registos, mais do que em muitas outras CPU's, que nos são muito úteis em programação. Alguns são quase exclusivamente usados pela CPU para sua própria consulta, equivalente até certo ponto às variáveis de sistema. Um destes é o Contador de Programa.

O Contador de Programa indica à CPU a localização da instrução que está a executar. Consideremos o programa seguinte.

```
8000   LD A,10      3E  10
8002   LD B,08      06  08
8004   ADD A,B      80
8005   RET          C9
```

À esquerda vemos endereços a partir de 8000. É neles que se encontra guardado o código a executar. Olhando para este, verifi-

camos que se começa por carregar 10 hexadecimal no registo A. Mantenha presente que trabalharemos sempre em hexadecimal, senão ficará certamente confuso! Em seguida carrega-se 08 hexadecimal no registo B. Depois somam-se os conteúdos dos registos A e B. A instrução RET reenvia-nos ao comando Basic.

Voltemos ao Contador de Programa. À medida que a CPU "lê" cada instrução, o contador de Programa passa ao endereço seguinte. De início, ao executarmos o programa acima, o Contador de Programa (PC) guarda o endereço 8000 (ou seja aquele onde se encontra a primeira instrução). A CPU reconhece que deve carregar no registo A um "valor directo", e portanto lê-o no endereço seguinte, 8001, nesse momento disponível no Contador de Programa que entretanto foi incrementado. A CPU carrega aquele valor, 10 hexa, no Acumulador (registo A), e entretanto o PC é novamente incrementado de modo a permitir a leitura da instrução seguinte no endereço 8002. Este processo é continuado até ser lida uma instrução que mande executar um salto; neste caso existe uma instrução RET, mas por agora não falaremos dela.

O leitor perguntará talvez "como posso ordenar à CPU que dê um salto?". Convirá portanto esclarecer este ponto. Utilizamos a instrução JP seguida de um endereço (para onde se deseja saltar). Esta instrução é de facto muito semelhante à GO TO, exceptuando o facto de não se poder aqui utilizar expressões; é possível usar alguns registos mas falaremos do assunto mais tarde. Observemos por agora o pequeno programa que se segue.

```
8000   LD A,10
8002   LD B,01
8004   ADD A,B
8005   JP 8004
```

Este programa carrega 10 hexadecimal no Acumulador e 01 hexadecimal no registo B. Em seguida soma B a A. A instrução que se segue obriga a execução a saltar para o endereço 8004, onde B é novamente somado a A. Esta operação continua indefinidamente, a menos que o leitor tropece como é habitual no cabo de alimentação...

O que de facto acontece é que quando a CPU encontra a instrução JP lê os dois bytes que se seguem e carrega o valor neles contido no Contador de Programa. Este registo duplo passa

portanto a conter um endereço diferente, 8004, o que leva por sua vez a CPU a procurar a instrução seguinte no novo endereço. Note o modo como o número 8004 é guardado no programa, começando por 04 e seguindo-se 80. Tenha sempre em conta este modo de guardar números quando estes ocupam dois bytes. No caso da instrução JP, é *sempre* seguida de dois bytes.

Por exemplo, se quiser saltar para o endereço 0001 não pode esquecer o primeiro byte (ou seja, os dois primeiros zeros):

```
8000  JP  0001      C3  01  00
```

Observemos agora o programa que se segue:

```
8000  JP  8006      C3  06  80
8003  JP  8009      C3  09  80
8006  JP  8003      C3  03  80
8009  JP  8003      C3  03  80
800C  RET
800D  JP  800C      C3  0C  80
```

A CPU atingirá alguma vez a instrução RET que lhe permitirá voltar ao comando Basic?

Como o leitor já deve ter compreendido, o código da instrução JP é C3 (hexadecimal), sendo seguida de dois bytes que indicam à CPU o endereço para o qual deve saltar.

Existem outras formas da instrução JP, a maior parte das quais estudaremos no segundo capítulo, mas vejamos já algumas:

```
JP  (HL)
JP  (IX)
JP  (IY)
```

Em Basic poderemos dizer:

```
GO TO A
```

Em código-máquina podemos actuar do mesmo modo com algumas limitações desde que usemos os registos de indexação ou o par de registos. HL. O registo de indexação IY não deverá porém ser usado porque afecta o funcionamento do Spectrum, se

bem que não permanentemente, sendo no entanto possível que provoque a perda do programa que se encontra na máquina.

```
JP  (HL)
```

A instrução anterior provoca um salto para o endereço contido em HL. Por exemplo, se HL contém 700C (hexadecimal), será executado um salto para a instrução que se encontra neste endereço.

```
6FF9  01  02  00    LD    BC,0002
6FFC  21  00  70    LD    HL,7000
6FFF  E9            JP    (HL)
7000  09            ADD   HL,BC
7001  E9            JP    (HL)
7002  C9            RET
```

O programa que acabamos de apresentar não é particularmente útil, mas convirá ao leitor tentar apreceber-se do modo como funciona.

6FF9 — Carrega 2 em BC
6FFC — Carrega 7000 em HL
6FFF — Executa um salto para o endereço indicado em HL, neste momento 7000.
7000 — Soma BC a HL, pelo que este último registo passa a conter 2 + 7000, ou seja, 7002.
7001 — Novo salto para o endereço em HL, desta vez 7002 (irrelevante dado que se trata do byte seguinte).
7002 — RET, retorno a Basic.

Podemos tentar agora introduzir alguns programas curtos no seu Spectrum. O primeiro será precisamente aquele de que acabámos de falar, o que permitirá ao leitor verificar como funciona.

Para introduzir um programa no Spectrum necessitaremos de um pequeno programa Basic que guarde os valores necessários na memória, acima da RAMTOP. Escreva o seguinte:

```
10 LET address = 26665
15 READ a$
```

```
20 LET byte = 0
30 FOR i = 1 TO 0 STEP -1
40 LET a = CODE a$ -55: IF a$<":" THEN LET a=a+7
50 LET byte = byte + a * 16 + i: LET a$ = a$ (2 TO):
   NEXT i
60 POKE address, byte: LET address = address + 1:
   IF LEN a$ = 0 THEN GO TO 15
70 GO TO 20
80 DATA "010200","210070","E9","09","E9","C9"
```

Este programa traduzirá números hexadecimais, guardando-os depois em qualquer ponto da ROM por nós escolhido. Na linha 10 vemos que à variável "address" ("endereço") foi atribuído o valor 28665. Isto deve-se ao facto de a primeira instrução do programa em código-máquina ser armazenada em 6FF9 (ou seja, 28665 em decimal).

Os códigos de operação, em hexadecimal, são guardados numa declaração DATA na linha 80. Baixamos a RAMTOP usando a instrução CLEAR com o sufixo 28600; trata-se de um valor inferior ao necessário mas convém muitas vezes deixar algum espaço livre para alterar um programa, por exemplo.

Escreva em seguida:

```
RUN
```

O programa será interrompido com a mensagem de erro:

```
Out of DATA Line 80
```

ou seja, falta de dados na linha 80. Esta mensagem não é aqui importante — acontece apenas que o programa não sabe que já dispõe dos dados suficientes.

O programa é carregado em dez bytes da RAM, começando em 28665 decimal ou 6FF9 hexadecimal. Podemos colocá-lo agora em execução — escreva:

```
PRINT USR 28665
```

Depois de dar entrada a esta ordem verá impresso no visor o número dois.

Este simples facto informa-nos já de que a CPU atingiu a instrução RET no endereço 7002 (hexadecimal). Se assim não fosse o programa executaria um ciclo sem fim, o que nos obrigaria a desligar a máquina da alimentação. O segundo aspecto curioso é a impressão do número dois.

De facto, quando o Spectrum termina a execução do nosso programa em código-máquina (ao encontrar a instrução RET), imprime no visor o valor contido no par de registos BC. Se observarmos o programa verificaremos que contém a instrução:

```
LD BC, 0002
```

que como sabemos carrega em BC o valor dois. É por isto que, quando usamos o prefixo PRINT numa instrução USR é impresso um valor. Nem sempre nos interessa porém que a máquina imprima o valor de BC no visor; é isso que acontece por exemplo num jogo. Mas podemos evitar isto usando a declaração LET. Usamos então uma variável que não tem qualquer utilidade além de respeitar a sintaxe da declaração LET. Experimente:

```
LET variável = USR 28665
```

Desta vez nada aparece escrito no visor, mas a rotina foi executada do mesmo modo. Para ter a certeza, escreva:

```
PRINT variável
```

Surpresa! Surgiu novamente no visor o número dois...

Por vezes queremos deslocar um programa de uma posição em memória para outra. Por exemplo, escrevemos um programa que deve ser guardado no topo da memória de um Spectrum de 16 K. Mas um amigo deseja usar também o seu programa, localizando-o porém no topo da memória de um Spectrum de 48 K. Usando o programa carregador de Basic, devemos obviamente alterar o valor da variável "address", na linha 10. Mas se tivermos usado a instrução JP no programa, teremos igualmente de alterar o endereço indicado por esta.

```
7FF0    LD B,08
7FF2    LD A,4C
```

```
7FF4    ADD A,B
7FF5    JP   7FFF
          :
          :
          :
7FFF    RET
```

O programa anterior é o escrito para o Spectrum de 16 K. Prolonga-se até 7FFF, o último byte de memória. Soma 08 a 4C (hexadecimal), e em seguida salta para 7FFF onde "encontra" uma instrução RET que reenvia o comando para a Basic. Vejamos o que acontece se nos limitarmos a guardá-lo num endereço mais elevado num Spectrum de 48 K:

```
FFF0    LD   B,08
FFF2    LD   A,4C
FFF4    ADD   A,B
FFF5    JP    7FFF
          :
          :
          :
FFFF    RET
```

O programa soma novamente 08 e 4C, mas em seguida salta para 7FFF. Este byte pode conter qualquer coisa, e portanto é necessário alterar a instrução RET. Neste caso é fácil fazer a alteração, mas imagine o que acontecerá num programa com 50 instruções JP... Seria necessário algum tempo para alterá-las a todas. Necessitamos portanto de um programa que possa ser "re-localizado" ("relocatable", em inglês), isto é, de um programa que possa ser passado para um ponto diferente da memória sem necessidade de alterações.

Em código-máquina dispomos de uma instrução especial que nos possibilita isto. Possui mais algumas limitações quando comparada à JP, mas continua a ser bastante versátil. Para a usar devemos no entanto compreender a convenção designada pelo nome "complemento para dois".

O salto em causa é chamado Salto Relativo (JR), sendo esta mnemónica seguida de um número que especifica a quantidade de bytes que devem ser ignorados e se o salto é realizado para a

frente ou para trás. Se o salto é executado para trás usa-se um número negativo, e é aqui que se usa a convenção "complemento para dois".

Normalmente a CPU trata apenas números positivos, mas por vezes, como acontece no caso dos saltos JR, pode reconhecer certos tipos de números de um só byte como sendo negativos.

Para formar um número negativo, digamos 10 (hexadecimal), subtraimo-lo de 100 hexadecimal (ou seja, de 256 decimal).

```
100 hex  −  10 hex  =  F0 (ou − 10 hex)
256 dec  −  16 dec  =  240 (ou -16 dec)
```

Estes dois cálculos são iguais, mas executados nas duas bases hexadecimal e decimal. Talvez seja mais fácil ao leitor realizar a subtracção em decimal, pelo que necessitará de fazer uma conversão. Pode utilizar o Apêndice A do Manual Sinclair ou o Apêndice A deste livro para converter os valores.

Entre as instruções Z80 existe porém uma instrução que realiza este cálculo em nosso lugar:

```
NEG
```

Esta instrução transforma um número positivo que se encontre no Acumulador num número negativo. Vamos experimentar um pequeno programa para verificar os resultados desta instrução.

Introduza na máquina o programa Basic já listado, não se esquecendo porém de escrever:

```
CLEAR 28600
```

Em seguida, utilize o programa em código-máquina que se segue:

```
7000  3E  10      LD    A,10
7002  ED  44      NEG
7004  4F          LD    C,A
7005  06  00      LD    B,00
7007  C9          RET
```

Note que NEG é uma instrução de dois bytes. Utiliza o prefixo ED hexadecimal, e 44 como código de operação.

Para carregar o programa no Spectrum altere a linha 80, a linha DATA do programa Basic, para:

```
80 DATA "3E10","ED44","4F","0600","C9"
```

Altere ainda a linha 10 para:

```
10 LET address = 28672
```

Em seguida escreva:

```
RUN
```

e depois:

```
PRINT USR 28672
```

Tal como esperávamos surge no visor o número 240, a resposta em decimal que obtivémos já anteriormente.

Observemos o programa:

7000 — Carrega-se 10 hex em A
7002 — O valor em A é tornado negativo
7004 — Passamos A para BC a fim de podermos ver a resposta no visor.
7005 — O byte mais significativo, B, é eliminado porque não é necessário no caso de números inferiores a 256.
7007 — RET reenvia o comando para Basic.

Vejamos agora o seguinte programa:

```
7000  0E  00        LD    C,00
7002  06  01        LD    B,01
7004  18  04        JR      700A
7006  0E  10        LD    C,10
7008  06  00        LD    B,00
700A  C9            RET
```


Primeiramente o programa limpa C e carrega em B o número 01 hexadecimal. Nestas condições BC passa a conter 0100. Depois encontramos uma instrução de salto relativo, significando 04 que são ignorados quatro bytes — os correspondentes às instruções LD C,10 e LD B,00, passando directamente à instrução RET. O número que se segue à instrução JR, neste caso, conta sempre a partir do byte correspondente quando positivo.

Se portanto se seguisse 08, seriam ignorados os oito bytes seguintes.

Vejamos agora este segundo programa:

```
7000  C9            RET
7001  3E  10        LD    A,10
7003  06  06        LD    B,06
7005  80            ADD   A,B
7006  18  F8        JR      7000
7008  00            NOP
```

Se iniciarmos a execução pelo byte 7001, será carregado 10 em A, 06 em B, somando-se em seguida o conteúdo dos dois registos e atingindo uma instrução de salto relativo. Desta vez o número que se segue ao código de operação é negativo. O salto é executado para trás, num total de oito bytes. Se começarmos na instrução NOP encontraremos uma RET que ordena à CPU que devolva o comando à Basic. Verifiquemos estas "contas":

```
256 - 8 = 242;
```

convertendo para hexadecimal:

```
242 = F8 hexadecimal
```

É de facto o número F8 que se segue à instrução JR, pelo que tudo parece correcto.

Usando a complementação para dois podemos usar números positivos entre 0 e 127 (decimal) ou negativos entre -1 e -128 (decimal). De facto, o 0 é contado como sendo positivo. Num programa comprido não poderemos usar sempre a instrução JR, porque os saltos deverão ser feitos muitas vezes para bytes a

maior distância. Não é portanto possível tornar todos os programas "re-localizáveis".

Estude o seguinte:

```
7000  18  FE        JR    7000
7002  C9            RET
```

O leitor reconhecerá certamente o facto de FE, em hexadecimal, ser interpretado como -2. O programa saltará portanto de novo para a própria instrução JR, mantendo-se nela indefinidamente, ou até a máquina ser desligada.

Se não se importa de introduzir novamente na máquina o programa carregador em Basic (se é que o tirou da memória do computador), pode experimentar o programa atrás.

Altere no entanto a linha 80 para:

```
80 DATA "18FE","C9"
```

Carregue na tecla RUN, e em seguida escreva:

```
PRINT USR 28672
```

Nada parece acontecer!

De facto, a máquina está a executar um ciclo sem fim... Experimente carregar em BREAK – nada acontece ainda. Recorde que BREAK não actua sobre programas em código-máquina. Mais tarde indicaremos a forma de recuperar esta função; mas por agora limitemo-nos a desligar o Spectrum como única forma de parar a execução do programa...

Por vezes, no decorrer de um programa, o leitor desejará usar a mesma rotina bastantes vezes. Para não se ver forçado a escrevê- -la repetidamente, pode usar uma subrotina. As subrotinas em código-máquina são semelhantes às que usamos em Basic. À instrução GOSUB corresponde em código a instrução CALL ("chamar"), e à declaração RETURN corresponde em código RET.

Quando usamos código-máquina no Spectrum, a nossa rotina é de facto uma subrotina chamada pela ROM. Quando queremos que a rotina termine voltando à ROM (e portanto à linguagem de comando Basic), temos de usar a instrução RET.

CALL utiliza um endereço directo, isto é, um endereço que é carregado directamente em PC e nunca pode ser calculado relativamente como nas instruções JR. Não existe qualquer forma relativa da instrução CALL, o que dificulta também a escrita de programas que possam ser carregados em qualquer endereço da RAM.

O programa que se segue soma os dois números de dois bytes guardados em BC e DE, colocando o resultado em HL.

```
7000  60        LD    H,B
7001  69        LD    L,C
7002  19        ADD   HL,DE
7003  C9        RET
```

Está escrito sob a forma de uma subrotina, e acrescentamos agora um curto programa para a utilizar:

```
7004  01  4C  2A     LD    BC,2A4C
7007  11  7E  4D     LD    DE,4D7E
700A  CD  00  70     CALL  7000
700D  44             LD    B,H
700E  4D             LD    C,L
700F  C9             RET
```

Este programa carrega 2A4C em BC e 4D7E em DE, chamando em seguida a subrotina "somar". O resultado é novamente carregado em BC, o que nos permite imprimir o resultado no visor, em decimal, quando voltamos à Basic. Vamos executar este programa, alterando porém a linha 10 para:

```
10 LET address = 28672
```

E a linha 80 para:

```
80 DATA "60","69","19","C9","014C2A","117E4D",
   "CD0070","44","4D"
```

Escreva RUN e

```
PRINT USR 28672
```

O programa imprime no visor o número 30666; é este o valor da resposta em decimal.

Existem certas subrotinas muito úteis em certos programas. Por exemplo, no sétimo capítulo inclui-se uma que verifica se se carregou na tecla BREAK. A própria ROM contém muitas subrotinas úteis, duas das quais estudaremos mais adiante.

Como o leitor já notou, o código hexadecimal de uma chamada (''CALL'') incondicional é CD. A seguir escrevê-se o endereço directo, em dois bytes, por exemplo:

CALL 70C0 — CD C0 70

Existem outras formas da instrução CALL, que acedem subrotinas condicionalmente, mas só as estudaremos no próximo capítulo. Por agora citemos um outro tipo de chamada de subrotina — a instrução RST (''restart'' — ''recomeçar''). Esta instrução só permite no entanto aceder algumas subrotinas bem definidas já existentes em ROM — as que se iniciam nos endereços:

0000
0008
0010
0018
0020
0028
0030
0038

Esta instrução é mais rápida do que as CALL, e apenas ocupa um byte porque cada uma delas tem um código diferente:

RST 00 — C7
RST 08 — CF
RST 10 — D7
RST 18 — DF
RST 20 — E7
RST 28 — EF
RST 30 — F7
RST 38 — FF

Talvez o leitor pergunte a si mesmo porque razão lhe falo no assunto, visto que se trata de endereços em ROM que não pode incluí-los nos seus programas.

Erro! Entre estes endereços encontra-se um dos mais importantes para o utilizador — o início da rotina em ROM que imprime texto no visor, acedido pela instrução RST 10.

Esta subrotina é uma espécie de mina de ouro para todos aqueles que programam em código-máquina, e poupa todo o trabalho de ter de reescrevê-la... Para a aceder devemos previamente carregar em A o código do caracter que desejamos imprimir, escrevendo em seguida a instrução RST 10. Por exemplo, se quisermos imprimir ''Olá'':

```
LD A, 4F
RST 10
LD A, 6C
RST 10
LD A, 61
RST 10
RET
```

Em primeiro lugar carrega-se no Acumulador o número 4F (hexadecimal). Se o leitor olhar para a tabela existente no Apêndice 1 do Manual Sinclair verificará que este número corresponde à letra maiúscula ''O''. Em seguida, RST leva a CPU a executar a rotina que se inicia no endereço 10 (hexadecimal). Passamos depois à impressão da letra ''l'' e da letra ''a''. Falta evidentemente o acento, e as letras são impressas normalmente a preto sobre fundo branco. Alterar este modo de imprimir a mensagem obriga necessáriamente a complicar o programa. Suponhamos que o leitor quer imprimir o acento sobre o ''a''. Pode aproveitar para isso a plica existente no teclado do Spectrum, cujo código é 39 decimal (27 hexadecimal), como pode verificar no Apêndice A do Manual Sinclair. Mas a letra ''a'' deve ser impressa sob o acento, e não à frente dele; dispomos no entanto de um código de retrocesso da posição de impressão no conjunto de caracteres do Spectrum (ainda no mesmo Apêndice A, o código 8, ''cursor left''). Executando a rotina de impressão para todos estes códigos o leitor conseguirá o seu objectivo:

```
LD A, 4F
RST 10
LD A, 6C
RST 10
LD A, 27
RST 10
LD A, 8
RST 10
LD A, 15
RST 10
LD A, 1
RST 10
LD A, 61
RST 10
RET
```

O leitor notará que além dos códigos correspondentes às letras e ao acento, e do código 8 correspondente ao retrocesso, são ainda empregues dois outros valores: 15 e 1. 15 (hexadecimal) é o código correspondente a OVER (consultar ainda o Apêndice A) seguindo-se o 1 da habitual instrução Basic OVER 1 que permite neste caso imprimir o "a" na mesma posição em que já se encontra o acento sem o apagar.

Podemos usar o mesmo processo para imprimir as letras a cores. Observemos novamente o Apêndice A, onde poderemos constatar que o comando de cor (INK) é 10 hexadecimal. Se agora enviarmos este código para a rotina de impressão, seguindo-o de um número entre 00 e 07, será escolhida a cor correspondente a este último número! Experimentemos por exemplo um Pedro em azul...

```
7000  3E 10      LD    A,10
7002  D7         RST   10
7003  3E  01     LD    A,01
7005  D7         RST   10
7006  3E  50     LD    A,50
7008  D7         RST   10
7009  3E  65     LD    A,65
700B   D7        RST   10
```



```
700C  3E  64     LD    A,64
700E  D7         RST   10
700F  3E  72     LD    A,72
7011  D7         RST   10
7012  3E  6F     LD    A,6F
7014  D7         RST   10
7015  C9         RET
```

Começamos por enviar o valor 10 para o Acumulador antes de aceder à rotina de impressão; em seguida temos de indicar o código de cor, neste caso 01 designando azul. Se não se tratar de um código válido (válido significa aqui entre 00 e 09) o computador devolve uma mensagem de erro:

K Invalid colour

Continuando a estudar o programa, o leitor facilmente identifica os códigos hexadecimais com as letras que formam a palavra "Pedro", consultando o Apêndice 1. Note em particular que o código de RST é D7. Talvez o leitor pense que o programa é excessivamente comprido para imprimir apenas "Pedro". Este é porém o método lento de construir o programa (apesar de muito rápido em execução). Mais tarde o leitor conseguirá descobrir um método mais rápido. Se não conseguir, pode usar a minha rotina descrita um pouco mais adiante. Vamos então executar este último programa.

Altere a linha 10 do carregador para:

```
10 LET address = 28672
```

(se não for já isto...).

Em seguida altere a linha 80 do seguinte modo:

```
80  DATA "3E10","D7","3E01","D7","3E50","D7",
    "3E65","D7","3E64","D7","3E72","D7",
    "3E6F","D7", "C9"
```

Depois de ter verificado se os dados estão certos, carregue em RUN e:

PRINT AT 0,0;:RANDOMIZE USR 28672

A primeira parte desta instrução (Print at 0,0) define a posição onde será impressa a palavra "Pedro". Observe o terceiro bloco de dados na linha 80 do carregador, "3E01". Experimente substituir 01 por outra cor a fim de verificar se a rotina funciona correctamente. Não se esqueça de que deve executar novamente o carregador antes de escrever a instrução USR. Não posso afirmar que sei tudo, e um facto para o qual não encontro uma razão clara é que quando se esquece a instrução PRINT AT... executando apenas a instrução USR a palavra "Pedro" é impressa durante curtos instantes desaparecendo imediatamente a seguir. Experimente por si mesmo:

RANDOMIZE USR 28672

Vê o que acontece? Descubra agora o que se passa quando escreve:

a) CLS:RANDOMIZE USR 28672
b) PRINT TAB 10:RANDOMIZE USR 28672
c) PRINT':RANDOMIZE USR 28672
d) PRINT;:RANDOMIZE USR 28672
e) PRINT,:RANDOMIZE USR 28672

É possível fazer muito mais coisas usando uma rotina de impressão. Pode-se fazer cintilar (FLASH) uma mensagem, ou imprimi-la com maior brilho. Talvez o leitor não se importe de olhar novamente para o famoso Apêndice A a fim de descobrir o código do comando de FLASH. Com sorte descobrirá que é 12 hexadecimal. Se não conseguir o seu manual está certamente mal impresso, ou o leitor necessita de óculos! Para "ligar" o FLASH basta usar a seguir a 12 o número 1 e para o desligar o número 0; o mesmo se passa com o comando BRIGHT, cujo código é 13 hexadecimal e que é ligado/desligado seguindo aquele código do número 1/0. Tentemos dar alguma animação ao nosso Pedro azul... Acrescentemos mais alguns dados no início da linha 80 (imediatamente antes de "3E10"):

"3E12","D7","3E01","D7","3E13","D7","3E01","D7"

Note que o 12 indica FLASH e 13 indica BRIGHT. Não se esqueça de verificar os dados. Execute o programa carregador, e finalmente escreva:

PRINT AT 0,0;:RANDOMIZE USR 28672

É possível usar outros códigos de comando – vejamos uma lista destes:

06 – Move a posição para a coluna 00 ou 16, a que estiver a seguir, tal como uma vírgula incluída numa instrução PRINT.

08 – Estes quatro códigos são usados pela máquina para
09 executar os comandos de cursor, deslocando este
0A para cima, para baixo, etc; nesta utilização,
0B porém, só 08 (esquerda) actua, sendo usado para imprimir "por cima" do que já foi escrito, como se descreve no Manual Sinclair. Os outros códigos imprimir um ponto de interrogação.

10 ,INK) — Já usámos este código; recordamos que deve ser seguido por um código válido de cor (00 a 99), alterando a cor de INK.

11 (PAPER) — Tal como 10, este código define uma cor – neste caso a de PAPER. Deve ser seguido igualmente de um código de cor válido.

12 (FLASH) — O código 12 "liga" o FLASH quando é seguido do valor 01, e "desliga-o" quando é seguido do valor 00. O código 08 pode ser usado para uma impressão "transparente", ou seja igual ao que está por baixo (esta questão é explicada no capítulo 16 do Manual Sinclair).

13 (BRIGHT) — Este código de comando altera a luminosidade (ou seja o brilho) do visor, do mesmo modo já descrito para FLASH.

14 (INVERSE) — "Liga" e "desliga" a impressão em inverso, do mesmo modo que BRIGHT.

15 (OVER) — Este código é usado para imprimir sobre outro caracter, apagando-o quando seguido do código 00 ou não o apagando quando seguido do código 01. Juntamente com o código de comando 08 pode ser usado para escrever letras com acentos, etc, como no exemplo de impressão da palavra "Olá".

II

IR OU NÃO IR

Já sabemos como executar saltos "absolutos" e relativos, e como chamar uma subrotina e voltar dela. Interessa agora estudar as decisões; de facto, o computador é definido como uma máquina capaz de realizar decisões lógicas.

Decisões, Decisões...

A CPU toma decisões baseando-se num único registo — o registo "F" (de "Flags"). Observemos melhor os bits do registo F, listados em seguida:

BIT 0 — C, flag "carry" (transporte de uma unidade em operações aritméticas).
1 — N, flag somar/subtrair.
2 — P/O, flag de paridade/"overflow".
3 — Não usado, sempre igual a zero.
4 — H, flag "half-carry".
5 — Não usado, sempre igual a zero.
6 — Z, flag zero.
7 — S, flag de sinal.

Como se pode verificar existem apenas seis "flags", não sendo usados dois dos bits. Cada bit está associado a uma letra ou duas que indicam o seu significado. Podemos ignorar a flag somar/subtrair (S) e a flag H porque a CPU não pode tomar

decisões baseando-se nelas, o que de resto pouca utilidade teria.

As flags alteram-se quando são executadas certas instruções e sempre de modos bem definidos. Normalmente alteram-se quando é realizada uma operação aritmética no Acumulador; alteram-se ainda quando se INCrementa ou DECrementa um registo, e noutras situações. Por exemplo, se acrescentarmos 10 ao Acumulador, as flags dir-nos-ão se o Acumulador passou a conter o valor zero, se é negativo (seguindo a convenção do completo para dois) ou se ultrapassou o valor máximo que este registo pode conter (''overflow'') e provocou o transporte de uma unidade.

As ''Flags''

A *flag ''carry''* diz-nos se ocorreu ou não um ''overflow'' no registo. Por exemplo, se somarmos 10 a F3 (sempre em hexadecimal), obtemos 103 hexadecimal, um valor excessivamente grande para um único byte; nestas condições o algarismo mais significativo (um) é eliminado, deixando como resultado apenas 03. Mas F3 + 10 não é de facto igual a 03, e a flag ''carry'' é passada ao valor 1 a fim de nos avisar do que se passou. Se pelo contrário for excedido o *menor* número que pode ser contido no byte (zero), a flag ''carry'' será igualmente passada ao valor 1. Se tirarmos 05 a 03 o resultado será um número negativo, representado em complemento para dois. Mas não sabemos se esse número é um valor hexadecimal positivo ou um número em complemento para dois; podemos no entanto verificar o estado da flag ''carry'' para verificar se foi ou não excedido o valor zero.

Flag paridade/''overflow''. O uso desta flag e as condições em que é passada ao valor um ou ao valor zero são um pouco complicados e não seria muito relevante explicá-los nesta parte do livro.

Flag zero. A utilidade desta flag é bastante simples: diz-nos se o resultado da última operação executada foi ou não zero. Por exemplo, carreguemos primeiramente o valor 02 hexadecimal no registo B. Se agora usarmos a instrução DEC B, a flag Zero será passada a zero porque B não contém ainda o valor zero. Mas se decrementarmos novamente B, este conterá de facto o valor zero e a flag Z será passada ao valor 1 para indicar o facto. Esta flag é extremamente útil, e tornaremos a falar dela mais adiante.

Flag de sinal. Se estamos a usar a complementação para dois, a flag de sinal dir-nos-á se o resultado da última operação foi negativo. Se tal acontecer esta flag será passada ao valor um, se não ao valor zero.

As outras duas flags, as H e N, servem apenas de referência à própria CPU — não vale portanto a pena preocuparmo-nos com elas.

Consideremos o exemplo seguinte:

```
LD A,10
LD B,10
SUB A,B
```

Subtrai-se 10 a 10, o que deixa zero em A; a flag Z passa portanto a 1. Como não houve transporte, a flag C passa a 0. A flag de sinal é passada a 0 indicando que o valor final guardado em A é positivo:

```
Z : 1
C : 0
S : 0
```

Observe os exemplos que se seguem; em todos eles indicamos o estado final das flags:

1.
```
LD A,00
LD B,B5
ADD A, B
```
```
Z : 0 (Zero)
C : 0 (Carry)
S : 1 (Sinal)
```

2.
```
LD A,00
SUB 20
```
```
Z : 0 (Zero)
C : 1 (Carry)
S : 1 (Sinal)
```

3.
```
LD A,00
SUB B5
```
```
Z : 0 (Zero)
C : 1 (Carry)
```

S : 0 (Sinal)

4. LD A,00
   ADD 20

Z : 0 (Zero)
C : 0 (Carry)
S : 0 (Sinal)

Qual seria o valor resultante se somássemos 10 hexadecimal a 70 hexadecimal?

Que valores conteriam as flags Zero, Carry, e Sinal?

Em primeiro lugar, se somarmos 70 hexadecimal a 10 hexadecimal obteremos 80 hexadecimal ou −7F em complemento para dois. Nem um nem outro destes valores é igual a zero, pelo que a flag Z contém o valor zero. Não foi ultrapassado o valor máximo ou mínimo possíveis, pelo que a flag Carry contém igualmente o valor zero.

Mas o bit 7 passa ao valor um, indicando que de acordo com a convenção de complementação para dois o número é negativo, pelo que a flag de sinal é passada a um.

''Mas afinal para que servem estas flags?'', perguntará o leitor. Bom, usando as instruções JP, JR, CALL e RET nas suas formas condicionais, podemos realizar saltos condicionados:

```
JP condição1, nn nn
CALL condição1, nn nn
JR condição2, dis
RET condição1
```

onde

condição1 = Z/NZ/NC/C/PO/PE/M/P
condição2 = Z/NZ/NC/C
dis (deslocamento) = ± número hexadecimal
nn nn = endereço hexadecimal

Como se pode ver, cada um dos saltos e a instrução de retorno utilizam agora um sufixo constituído por uma ou duas letras. Vejamos o que estas letras significam:

Z = Flag Zero ao valor 1.

NZ = Flag zero ao valor 0.
NC = Flag Carry ao valor 0.
C = Flag Carry ao valor 1.
PO = Flag P/V ao valor 0 (Paridade ímpar — Parity Odd)
PE = Flag P/V ao valor 1 (Paridade par — Parity Even)
M = Flag S ao valor 1 (negativo — Minus)
P = Flag S ao valor 0 (positivo).

Podemos portanto interpretar a instrução

JP Z,705C

como significando: ''se o resultado da última operação foi zero, saltar para 705C''. Do mesmo modo:

CALL NC,700A

pode ser interpretada como : se a última operação não provocou um transporte, chamar a subrotina que se inicia em 700A.

Por ''última operação'' entendemos a última instrução capaz de alterar as flags. Instruções como ''LD'' e muitas outras não actuam sobre as flags.

```
LD A,10
ADD A,05
LD B,03
LD HL,0735
RET Z
```

Quando, no exemplo acima, se atinge a instrução RET Z, as flags reflectem ainda o valor de A após a instrução ADD A,05. As várias instruções LD podem ser ignoradas quando se observa o estado das flags. A seguir apresenta-se uma lista das instruções capazes de alterar as flags que nos interessam, ou seja as flags Z, C, S e P/V.

```
ADC
ADD
AND
BIT
```

CCF
CP
CPJ
CPJR
CPDR
CPL
DAA
DEC (apenas em registos simples)
INC (apenas em registos simples)
IN
INI
IND
INIR
INDR
LD A,I (note-se que estas são as únicas instruções LD que afectam as flags)
LD A,R
LDI
LDD
LDIR
LDDR
NEG
OR
OUTI
OUTD
OTIR
OTDR
POP AF (as flags são definidas pelo byte superior do stack)
RLA
RL
RLCA
RLC
RLD
RRA
RR
RRCA
RRC
RRD
SBC
SCF

SLA
SRA
SRL
SUB
XOR

Ainda não estudámos algumas destas instruções; mas por enquanto não se preocupe com o assunto.

Como pode ver, a maior parte destas instruções têm a ver com a execução de operações matemáticas. Se deseja saber quais as flags alteradas por cada instrução consulte o Apêndice C no final do livro.

É muito vulgar utilizar as flags a seguir a instruções DEC e CP.

Em Basic podemos formar facilmente ciclos recorrendo às instruções FOR... TO... (STEP)... NEXT. Em código-máquina o problema apresenta-se de outro modo, se bem que o princípio aplicado seja o mesmo. Em primeiro lugar, para formar um ciclo bastante simples, carregamos num registo o número de vezes que desejamos ver feita determinada coisa; em seguida, onde utilizaríamos um NEXT em Basic, decrementamos esse registo, e se não for igual a zero fazemos executar o ciclo de novo. Observe o programa seguinte para tornar tudo isto mais claro. Imprime 5 letras "A", usando RST 10 como descrevemos no capítulo 1.

```
7000 06 05      LD      B,05
7002 3E 41      LD      A,41
7004 D7         RST     10
7005 05         DEC     B
7006 20 FA      JR      NZ,7002
7008 C9         RET
```

Estudemos um pouco este programa:

7000 – Carrega-se em B o número 5 hexadecimal.
7002 – Carrega-se em A 41 hexadecimal, porque é este o código de A.
7004 – Imprime o caracter.
7005 – B (o contador) é decrementado.
7006 – Se B não for ainda igual a zero, reenvia a 7002 a fim de imprimir um novo A — devido à instrução

JR NZ que significa 'Salto relativo se a flag zero não for 1''.

O registo B é muitas vezes usado como contador. Isto deve-se ao facto de dispormos de uma instrução muito útil que se aplica exclusivamente ao registo B:

DJNZ (dis) 10 xx

A instrução combina 'DEC B'' e ''JR NZ'' numa única ordem, e apresenta duas vantagens:

a) Gasta menos memória — dois únicos bytes, um para a instrução e outro para o deslocamento.

b) É executada mais rapidamente do que DEC B e JR NZ, o que pode por vezes ser uma questão crítica em ciclos a executar em tempos rigorosos.

Podemos portanto poupar um byte e reescrever o programa do seguinte modo:

```
7000 06 05      LD      B,05
7002 3E 41      LD      A,41
7004 D7         RST     10
7005 10 FA      DJNZ    7001
7007 C9         RET
```

Introduza esta versão modificada no computador, alterando do seguinte modo a linha 80:

80 DATA ''0605'',''3E41'',''D7'',''10FA'',''C9''

Assegure que a linha 10 é:

10 LET adress = 28672

Depois de fazer tudo isto e verificado os dados, escreva

RUN

e em seguida

PRINT AT 0,0;:RANDOMIZE USR 28672

Não se esqueça de que necessitamos da ordem PRINT AT 0,0 porque de outro modo o Spectrum pensa que está a montar uma linha e quando termina limpa o visor. Se tudo estiver certo, o leitor observará agora cinco A's maiúsculos no visor.

Este modo de executar ciclos é óptimo, mas que acontecerá se desejar cumprir um ciclo entre os valores 10 e 20? Este sistema já não servirá porque depende de uma contagem que termina em 0. É aqui que se torna muito útil a instrução de comparação:

CP

O significado literal é ''Compare o que se segue com o conteúdo do Acumulador''. O resultado da comparação é reflectido no estado das flags. A instrução subtrai o dado que se segue ao conteúdo de A sem colocar o resultado em A — limita-se a actualizar as flags para o resultado.

O dado que se segue à instrução pode ser um registo de um único byte, um número ou uma posição em memória indicada por HL, IX + deslocamento ou IY + deslocamento.

```
CP nn           FE nn
CP A            BF
CP B            B8
CP C            B9
CP D            BA
CP E            BB
CP H            BC
CP L            BD
CP (HL)         BE
CP (IX+(dis))   DD BE
CP (IY+(dis))   FD BE
```

À frente de cada instrução de comparação apresentamos os códigos correspondes.

A instrução de comparação é muito útil para resolução de situações que em Basic empregam as declarações IF... THEN; por exemplo, se compararmos com 20 e se A contiver 20 a flag Z será passada a 1 (20 − 20 = 0), se A for menor do que 20 a flag C será passada a 1, e se A for maior do que 20 a flag C será passada a 0.

As situações que apresentamos em seguida podem ser consideradas quase como tendo o significado que lhes damos em Basic:

1. CP 5F
   JP Z,7000

Como pode verificar, é subtraído teoricamente 5F ao conteúdo de A. Neste caso, se o cálculo produz zero, é realizado o salto. Este resultado só será obtido se em A estiver também 5F; poderíamos portanto escrever:

IF A=5F (hex) THEN GOTO 7000 (hex)

2. CP 2C
   JP C,7000

Neste caso subtrai-se teoricamente 2C a A (não se esqueça de que dizemos "teoricamente" porque o resultado nunca é colocado em A). Se A for maior do que 2C, a subtração desta quantidade dará um resultado positivo. Mas se A é inferior a 2C, dará um resultado negativo, passando a 1 a flag Carry. Poderíamos escrever isto sob a seguinte forma:

IF A < 2C (hex) THEN GOTO 7000 (hex)

3. CP 10
   JP NC,7000

Se tirarmos 10 a A e este registo contiver um número superior a 10, o resultado será positivo. Isto significa que não haverá transporte, e portanto a flag Carry passará a zero. Podemos portanto escrever:

IF A > 10 (hex) THEN GOTO 7000 (hex)

Voltemos agora ao nosso problema de um ciclo que não termine em zero. Observe o seguinte; trata-se de um ciclo que é cumprido entre 10 hexadecimal e 1F hexadecimal.

```
7000 3E 10      LD      A,10
7002 3C         INC     A
7003 FE 20      CP      20
7005 20 FB      JR        NZ,7002
```



```
7007 C9         RET
```

Primeiramente carrega-se 10 em A, sendo este o valor a que o ciclo se deve iniciar. Em seguida o Acumulador é incrementado (em 7002), e em seguida "CP 20" compara 20 com o conteúdo do Acumulador; se este não contém 20 a flag Z é passada a zero, sendo executado um salto para 7002. Comparamos com 20 porque é superior em uma unidade ao último valor que pretendemos. Se comparássemos com 1F A seria incrementado em 1E passando a conter 1F, o salto seria executado e o ciclo não seria executado para 1F.

Desenvolvendo este programa, vejamos um outro que imprime "A's" nas colunas 16 a 31 em decimal, ou 10 a 1F em hexadecimal.

```
7000 06 10      LD      B,10
7002 3E 17      LD      A,17
7004 D7         RST     10
7005 78         LD      A,B
7006 D7         RST     10
7007 3E 41      LD      A,41
7009 D7         RST     10
700A 78         LD      A,B
700B FE 20      CP      20
700D 20 F3      JR      NZ,7002
700F C9         RET
```

7000 B recebe o valor 10 em vez de A, porque necessitamos deste último registo para executar a impressão.
7002 Carrega-se em A o número 17 hexadecimal, que constitui o código de comando de TAB.
7004 O comando de TAB é "enviado".
7005 Passamos agora o número da coluna, que se encontra em B, para o Acumulador.
7006 É "enviado" o número da coluna.
7007 Carrega-se em seguida em A o código da letra "A".
7009 É impressa a letra A.
700A Passamos agora o valor em B para A, a fim de podermos compará-lo.
700B Compara-se com 20, porque o último valor que desejamos utilizar é IF (31 decimal).

700D Se o acumulador ainda não é 20 é impresso um novo caracter.
700F Retorno ao comando Basic.

No caso de ciclos que usem registos de um só byte a programação é bastante simples, mas quando necessitamos de fazer uma contagem envolvendo números de dois bytes a questão torna-se mais complicada. A decrementação de um registo de dois bytes afecta as flags pelo que estas não darão os resultados pretendidos. Não é portanto possível fazer o seguinte, por exemplo:

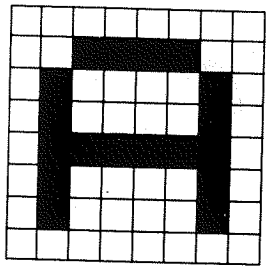
```
LD BC,062D
DEC BC
JR NZ...
```

Discutiremos o modo de resolver este problema no Capítulo IV, onde estudaremos todas as instruções que actuam "logicamente" sobre um registo.

O Conjunto de Caracteres

Antes de apresentar uma pequena rotina que permita imprimir caracteres com o dobro da altura normal, convirá rever o que se passa com o conjunto de caracteres usado pelo Spectrum.

Ainda bem que o computador permite o uso de caracteres definidos pelo utilizador, porque o leitor já estará familiarizado com o assunto e não necessito de prolongar aqui uma explicação sobre ele. Todos sabemos que cada caracter é constituído por 64 pontos formando um quadrado de oito vezes oito pontos (ou pixels). Muitos saberão ainda que cada fiada de oito pontos é guardada em memória sob a forma de oito bits, ou um byte.

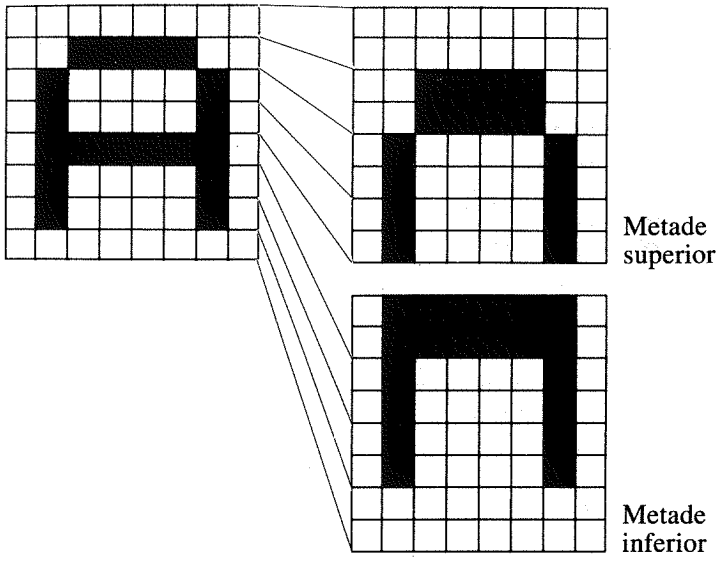


128	64	32	16	8	4	2	1	
0	0	0	0	0	0	0	0	00
0	0	1	1	1	1	0	0	3C
0	1	0	0	0	0	1	0	42
0	1	0	0	0	0	1	0	42
0	1	1	1	1	1	1	0	7E
0	1	0	0	0	0	1	0	42
0	1	0	0	0	0	1	0	42
0	0	0	0	0	0	0	0	00

O conjunto de caracteres normalmente usado pelo Spectrum está guardado em ROM a partir do endereço 3000 hexadecimal. Este conjunto define todos os caracteres desde o "espaço" até ao símbolo de copyright. Os gráficos definidos pelo utilizador encontram-se guardados noutro ponto da memória. Podemos no entanto redefinir o conjunto de caracteres usado pela máquina, guardando o novo conjunto a partir de um endereço RAM apropriado e alterando o valor da variável de sistema CHARS (5C36). Esta variável contém normalmente o valor 3C00, igual ao endereço do conjunto de caracteres menos 100 hexadecimal (256 decimal). Ao definirmos um novo conjunto de caracteres devemos também guardar nesta variável o novo endereço menos 100 hexadecimal para que a máquina o possa encontrar.

Caracteres de altura dupla

Para criar um caracter com o dobro da altura normal somos de facto forçados a criar dois conjuntos de caracteres. Um conterá a metade superior de todas as letras, e o outro conterá a sua metade inferior. Para produzir estes dois conjuntos de caracteres poderemos partir do conjunto original e "esticar" cada caracter de modo a ocupar o espaço de dois em altura. Por exemplo, no caso da letra A, podemos obter o caracter do modo indicado na figura da página seguinte.



Assim, os quatro bytes superiores são duplicados de modo a obter os oito bytes superiores do novo caracter; os quatro bytes inferiores do original são do mesmo modo transformados nos oito bytes da metade inferior do novo caracter.

Recorrendo a um programa em código-máquina podemos realizar este trabalho com grande rapidez. Em primeiro lugar necessitamos de um ciclo "exterior" que execute cada um dos 96 caracteres. Depois necessitaremos de um ciclo interior que trate a parte superior de cada caracter, e outro que trate a parte inferior destes.

O programa funcionará portanto do seguinte modo:

```
TRATAR CADA CARACTER <-------+
                             |
TRATAR METADE SUPERIOR<-+    |
                        |    |
TRATAR METADE INFERIOR<-+    |
                             |
CARACTER SEGUINTE ---------->+
```


Teremos de decidir agora onde convirá guardar ambos os conjuntos de caracteres. O melhor local será acima da RAMTOP, para não serem corrompidos por outros programas. Se dispomos de um programa Disassembler ou Monitor já localizado acima da RAMTOP, teremos de alterar os endereços. Onde diz PUT1 e PUT2, na primeira parte do programa, use os endereços seguintes:

```
48K: (PUT1);FA      (PUT2);FD
16K: (PUT2);7A      (PUT2);7D
```

A nossa primeira tarefa consistirá em construir um ciclo que processe cada um dos caracteres. Necessitamos de utilizar dois endereços, que nos indicarão o início de cada um dos conjuntos de caracteres.

```
LD HL,BOTSET  21 00 (PUT2)
PUSH HL       E5
LD HL,TOPSET  21 00 (PUT1)
LD DE,SNCET   11 00 3D
LD C,60       0E 60
```

Definimos portanto o endereço da parte inferior (BOTSET) e guardamo-lo no stack. Definimos em seguida o endereço da parte superior (TOPSET) e deixamo-lo em HL. Carregamos DE com o valor SNCSET (3000), que é o início do conjunto de caracteres normal do Spectrum. Finalmente carregamos em C o valor 60 hexadecimal porque é necessário tratar 96 (60 em hexadecimal) caracteres. Em seguida tratamos do ciclo de execução da metade superior de cada caracter.

```
TPHALF  LD B,04       06 04
TSLICE  LD A,(DE)     1A
        LD (HL),A     77
        INC HL        23
        LD (HL),A     77
        INC (HL)      23
        INC DE        13
        DJNZ TSLICE   10 F8
```

Note que a primeira instrução carrega quatro em B. Define-se

assim B como contador, e como temos de construir quatro grupos de dois bytes iguais em cada meio-caracter é carregado com 4. Em seguida carrega-se no Acumulador o byte endereçado por DE; recordemos que este par de registos indica o conjunto de caracteres normal do Spectrum. Este byte é agora transferido para o primeiro conjunto de caracteres novos. Em seguida incrementa-se HL e o novo byte é novamente guardado em (HL) (recorde-se de que temos de duplicar cada byte para "esticar" a letra). Em seguida, DE é incrementado, passando a indicar o segundo byte do caracter original.

Este processo é repetido quatro vezes usando a instrução DJNZ que já estudámos anteriormente.

O último passo antes de encerrarmos o ciclo exterior consiste em construir a metade inferior do caracter. Isto é feito de uma maneira muito semelhante à usada para a metade superior.

```
        EX HL,(SP)      E3
        LD B,(04)       06 04
BSLICE  LD A,(DE)       1A
        LD (HL),A       77
        INC HL          23
        LD (HL),A       77
        INC HL          23
        INC DE          13
        DJNZ BSLICE     10 F8
        EX HL,(SP)      E3
        DEC C           0D
        JR NZ,TPHALF    20 E6
        POP HL          E1
        RET             C9
```

Este sector do programa inicia-se de modo diferente, por uma instrução EX HL,(SP). Para os não iniciados, esta instrução muito útil serve apenas para substituir o valor em HL pelo que se encontra no stack. Usamo-la aqui para substituir em HL o endereço do primeiro conjunto de caracteres pelo do segundo conjunto. Após DJNZ BSLICE encontra-se outra EX HL(SP), que obriga novamente HL a apontar para o primeiro conjunto de caracteres. C é então decrementado, e se não for zero executa um salto relativo para construir o caracter seguinte. Se já não houver

mais caracteres são executadas as duas últimas instruções, POP HL e RET. A primeira serve para remover o último valor do stack, que indica a posição do primeiro conjunto de caracteres.

```
7000  21 00 7D   LD    HL,7D00
7003  E5         PUSH  HL
7004  21 00 7A   LD    HL,7A00
7007  11 00 3D   LD    DE,3D00
700A  0E 60      LD    C,60
700C  06 04      LD    B,04
700E  1A         LD    A,(DE)
700F  77         LD    (HL),A
7010  23         INC   HL
7011  77         LD    (HL),A
7012  23         INC   HL
7013  13         INC   DE
7014  10 F8      DJNZ  700E
7016  E3         EX    (SP),HL
7017  06 04      LD    B,04
7019  1A         LD    A,(DE)
701A  77         LD    (HL),A
701B  23         INC   HL
701C  77         LD    (HL),A
701D  23         INC   HL
701E  13         INC   DE
701F  10 F8      DJNZ  7019
7021  E3         EX      (SP),HL
7022  0D         DEC   C
7023  20 E7      JR      NZ,700C
7025  E1         POP   HL
7026  C9         RET
```

Para fazer funcionar esta rotina introduza os códigos hexadecimais em declarações DATA no final do carregador. Não se esqueça de usar os números mais apropriados a 16K ou 48K (conforme a máquina que estiver a usar). No programa escrevi os valores para 16K, mas se os utilizadores de máquinas de 48K voltarem um pouco atrás verão como os devem alterar. Se estiver

a usar um programa Monitor de código-máquina que utilize os endereços 7A00 a 7FFF em 16K ou FA00 a FFFF em 48K, terá de fazer algumas alterações nos endereços senão o programa formará os novos conjuntos de caracteres em endereços já ocupados pelo Monitor...

Vejamos agora o modo de usar o nosso gerador de caracteres duplos. Partindo do principio de que já deu entrada a todos os dados e fez executar o carregador, interessar-lhe-á certamente saber como pode usar as novas letras.

Primeiramente deverá definir duas variáveis:

```
Para a máquina de 16K — LET top = 121
                       LET bot = 124
Para a máquina de 48K — LET top = 249
                       LET bot = 252
```

Imaginemos agora que pretende imprimir a palavra "Hello":

```
10 POKE 23607,top: PRINT "Hello"
20 POKE 23607,bot: PRINT "Hello":POKE 23607,60
```

Na primeira linha fazemos apontar a variável CHARS (uma variável de sistema, veja o Apêndice C) para o conjunto de caracteres superior. A máquina imprime então as metades superiores dos caracteres. Para podermos acrescentar a metade inferior dos caracteres imediatamente abaixo usamos novamente a declaração POKE levando CHARS a apontar para o segundo conjunto de caracteres. O último POKE da linha 20 é necessário para voltar ao conjunto de caracteres normal do ZX Spectrum.

Veja o que acontece quando se faz:

```
POKE 23607,top
```

em modo directo. Do mesmo modo, experimente:

```
POKE 23607,0
```

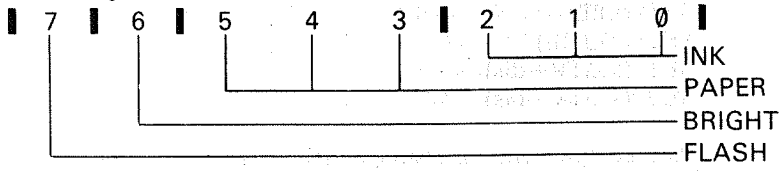
Porque razão se altera a aparência do visor? Consegue encontrar uma utilidade para isto?

III

OS BITS

Como o leitor sabe certamente cada byte é constituído por oito bits. Cada um destes é um algarismo binário. A soma de todos estes algarismos constitui um byte. Por vezes usamos um byte não no seu conjunto, como uma "palavra de oito bits", mas como um conjunto de bits independentes, usados como "flags" ou para indicar números menores. Já observámos o registo F e os seus bits, usados independentemente uns dos outros, e o modo como cada um destes bits nos diz alguma coisa sobre a última operação ou comparação executadas em código. O Ficheiro de Atributos (que se inicia em 5800 hexadecimal) dispõe de um byte para indicar as características em termos de cor, brilho, etc. de cada posição de caracter no visor. Cada byte de atributos indica ao processador a cor da tinta, a cor de fundo e o facto de essa posição ter uma côr mais brilhante ou estar a cilintar.

Vejamos como se distribuem os byts de atributos:



Se quisermos fazer cintilar um caracter, passamos ao nível 1 o bit 7 do correspondente byte de atributos, não sendo necessário alterar qualquer dos outros bits. Para conseguir isto dispomos da instrução:

SET

O programa que se segue faz cintilar o primeiro caracter do visor:

```
7000  21  00  58        LD    HL,5800
7003  CB  FE            SET   7,(HL)
7005  C9                RET
```

7000 Carrega-se em HL o valor 5800 hexadecimal. É este o endereço do primeiro byte de atributos.
7003 A instrução SET passa a 1 o bit 7 da posição (HL), e como HL aponta para o primeiro atributo e o bit 7 corresponde a ''flash'', o primeiro caracter passa a cintilar.
7004 Retorno ao comando Basic.

A instrução SET apenas passa a 1 o bit especificado — deixa todos os bits tal como estão. Pode ser usada para alterar o estado de qualquer bit (0 a 7) de qualquer registo simples (A,B,C,D,E,H, ou L) ou qualquer posição indicada por (HL), (IX + deslocamento) ou (IY + deslocamento). Vejamos em seguida uma lista das instruções SET (x é um número qualquer entre 0 e 7):

```
SET (x),A
SET (x),B
SET (x),C
SET (x),D
SET (x),E
SET (x),H
SET (x),L
SET (x),(HL)
SET (x),(IY + dis)
SET (x),(IX + dis)
```

Para complementar a instrução SET existe ainda a instrução RES. Actua de um modo muito semelhante à instrução anterior, passando no entanto o bit apropriado a zero.

```
RES (x),A
```



```
RES (x),B
RES (x),C
RES (x),D
RES (x),E
RES (x),H
RES (x),L
RES (x),(HL)
RES (x),(IY + dis)
RES (x),(IX + dis)
```

Assim, se quisermos que o primeiro caracter deixe de cintilar podemos passar o bit 7 para 0, usando a instrução RES.

```
7000  21  00  58        LD    HL,5800
7003  CB  BE            RES   7,(HL)
7005  C9                RET
```

Como existem tantas instruções SET e RES diferentes perderíamos muito espaço a listá-las completamente, mas o leitor pode encontrá-las no Apêndice A deste livro ou do manual Sinclair.

O programa que se segue demonstra o uso da instrução SET, colocando em ''flash'' as oito linhas superiores do visor.

```
7000  21  00  58        LD    HL,5800
7003  06  00            LD    B,00
7005  CB  FE            SET   7,(HL)
7007  23                INC   HL
7008  10  FB            DJNZ  7005
700A  C9                RET
```

7000 Carrega-se em HL o endereço do primeiro byte, 5800.
7003 Carrega-se em B, usado como contador, o valor 00 de modo a preencher os primeiros 256 bytes.
7005 Passa-se a um o bit ''flash'' da posição indicada por HL.
7007 Altera-se o endereço contido em HL para o atributo seguinte.
7008 A instrução DJNZ provoca a repetição da operação 256 vezes.
700A Retorno ao comando Basic.

Experimentemos agora o programa, usando o carregador Basic depois de substituir a linha 80:

```
80 DATA "210058","0600","CBFE","23",
   "10FB","C9"
```

e de verificar se a linha 10 é:

```
10 LET address = 28672
```

Em seguida carregue em RUN, e quando obtiver a mensagem de erro (ao terminarem os dados) escreva:

```
RANDOMIZE USR 28672
```

Como pode verificar, as oito linhas superiores "cintilam" — se tal não acontecer escreveu certamente algo incorrecto. Apenas para demonstrar que nada mais foi alterado nos vários caracteres, escreva:

```
LIST:RANDOMIZE USR 28672
```

Surgirá imediatamente no visor um programa cujas linhas superiores cintilam! Podemos agora tornar a imagem mais brilhante introduzindo uma pequena alteração:

```
7000  21  00    58      LD    HL,5800
7003  06  00            LD    B,00
7005  CB  F6            SET   6,(HL)
7007  23               INC   HL
7008  10  FB            DJNZ  7005
700A  C9               RET
```

Em vez de passarmos ao valor 1 o bit 7 façamo-los com o bit 6, o bit que indica o brilho dos caracteres. Alteremos portanto o valor "CBFE" da linha 80 para "CBF6".

Em seguida escreva:

```
RUN,
```


e depois

```
LIST:RANDOMIZE USR 28672
```

Se a cor de fundo for branca, o bit "lavado" pelo código-máquina será ainda mais branco!

Convém ainda estudarmos uma última instrução relacionada com os bits:

```
BIT
```

Esta instrução verifica qual é o estado de um determinado bit; se for zero passa a 1 a flag Z, e se for um passa a 0 a mesma flag. As diferentes formas da instrução BIT são semelhantes a RES e SET, e tal como estes dois tipos de instrução aquelas empregam o prefixo CB (hexadecimal). Os códigos podem uma vez mais ser encontrados no Apêndice A deste livro ou do manual Sinclair.

```
BIT (x),A
BIT (x),B
BIT (x),C
BIT (x),D
BIT (x),E
BIT (x),H
BIT (x),L
BIT (x),(HL)
BIT (x),(IX+dis)
BIT (x),(IY+dis)
```

O programa que se segue altera todos os caracteres, modificando para todos o atributo "FLASH":

```
7000  21  00    58      LD    HL,5800
7003  CB  7E            BIT   7,(HL)
7005  28  04            JR      Z,700B
7007  CB  FE            RES   7,(HL)
7009  18  02            JR       700D
700B  CB  FE            SET   7,(HL)
700D  23               INC   HL
```

```
700E  7C        LD    A,H
700F  FE  5B    CP    5B
7011  20  F0    JR    NZ,7003
7013  C9        RET
```

7000 : Carrega-se em HL o início do ficheiro de atributos.
7003 : É verificado o bit "flash" de (HL).
7005 : Se o caracter não cintila, a execução salta para 700B.
7007 : Senão deixa de cintilar.
7009 : Salto para 700D.
700B : Faz o caracter cintilar.
700D : Desloca HL para o atributo seguinte.
700E : Passa H para A e verifica se se encontra no final do ficheiro de atributos.
7011 : Se não estiver, passa a 7003 para tratar o byte seguinte.
7013 : Retorno ao Basic.

Experimentemos agora este programa.

Altera a linha 80 para o seguinte:

```
80 DATA "210058","CB7E","2804","CBBE",
   "1802","CBFE","23","7C","FE5B","20F0","C9"
```

Em seguida carregue em

RUN

e depois escreva:

RANDOMIZE USR 28672

O visor fica em FLASH. Em seguida escreva novamente no teclado:

RANDOMIZE USR 28672

O visor volta ao normal. Experimente escrever:

PRINT AT 10,0; FLASH 1; "Verificar a rotina FLASH 2"

Em seguida consegue fazer cintilar todo o visor, exceptuando a mensagem, escrevendo novamente:

RANDOMIZE USR 28672

Vamos agora aproveitar tudo o que já aprendemos para escrever um novo programa.

Máquina de escrever

Vamos desenvolver um programa a que chamaremos "Máquina de escrever". Trata-se de um programa bastante simples que permite utilizar o visor como uma página semelhante à usada nas máquinas de escrever — podendo-se imprimir nela instruções de programa depois passadas para uma impressora, por exemplo. As principais funções deste programa são:

a) Comandos do cursor: para cima, para baixo, para a esquerda e para a direita.
b) Repetição automática das teclas.
c) Envio do cursor para a posição inicial.
d) Apagar caracteres.
e) Retorno ao princípio da linha seguinte.

Em primeiro lugar necessitamos de dois bytes de dados que nos indiquem a posição do cursor no visor, e de dois outros que indiquem a posição de impressão no ficheiro de atributos. O quinto byte de dados dir-nos-á qual a tecla premida pelo utilizador. Vamos dar a estes bytes os seguintes nomes:

PRINT — 6D00
ATTRB — 6D02
KEY — 6D04

Começaremos por escrever o programa a partir de 6D05, ou seja imediatamente depois dos dados.

Até agora ainda não explicámos o modo de ler o teclado em código-máquina. É no entanto muito simples executar esta fun-

ção. Nas variáveis de sistema existe uma posição chamada LAST K, que contém o código da última tecla premida. Este valor é actualizado 50 vezes por segundo. O método usado para conhecer a tecla premida é portanto o seguinte:

1. Tornar a variável LAST K igual a zero.
2. Ler o teclado.
3. Passar o valor de LAST K para o Acumulador.
4. Se o Acumulador contém zero voltar ao passo 2.

O aspecto positivo deste sistema é que permite a repetição automática da tecla premida.

Podemos utilizar os nossos conhecimentos sobre a instrução RST 10 para imprimir caracteres no visor usando o código de comando AT, mas o cursor em flash não pode ser obtido através desta instrução porque destruiria o que se encontrasse escrito sob o cursor. Isto deve-se ao facto de o nosso cursor ser diferente do normalmente usado pelo Spectrum. Em vez de se encontrar sempre entre caracteres, estará sobre eles. Isto evita a necessidade de deslocar todos os caracteres à direita do cursor sempre que este é movido.

O programa fará uma de duas coisas sempre que é premida uma tecla: imprimir um caracter ou realizar uma operação de cursor, por exemplo apagar um caracter ou deslocar o cursor para cima. Deve saber quais os códigos que deve imprimir e quais os códigos que correspondem a comandos do cursor. Vamos portanto obrigar o programa a consultar uma tabela de códigos, tendo cada um deles um endereço de dois bytes à frente. Se o código da tecla premida corresponde a um dos códigos da tabela, o programa salta para o endereço indicado à frente do código. Se não encontra qualquer código igual, imprime o código respectivo. Observemos a tabela seguinte:

```
DEFB 07 — EDIT          0705 6D
DEFB 08 — LEFT          0883 6D
DEFB 09 — RIGHT         095A 6D
DEFB 0A — DOWN          0A69 6D
DEFB 0B — UP            0B76 6D
DEFB 0C — DELETE        0C94 6D
```



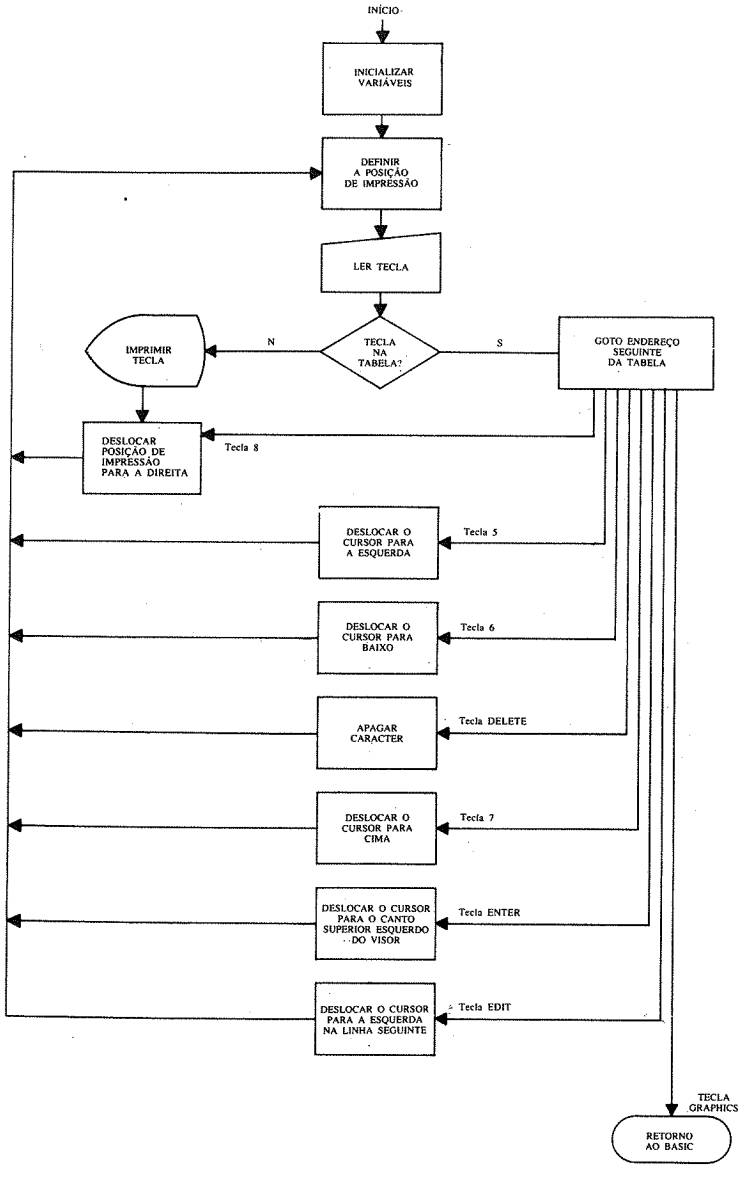
```
DEFB 0D — ENTER         0DA6 6D
DEFB 0F — GRAPHICS      0FA5 6D
DEFB FF — FINAL DE TABELA       FF
```

Nesta tabela encontram-se os códigos correspondente a várias teclas de comando — tendo cada um deles números de dois bytes à frente. Estes números de dois bytes indicam o início da rotina que trata o comando em causa:

```
07 — EDIT       — Desloca o cursor para 0,0.
08 — LEFT       — Desloca o cursor para a esquerda.
09 — RIGHT      — Desloca o cursor para a direita.
0A — DOWN       — Desloca o cursor para baixo.
0B — UP         — Desloca o cursor para cima.
0C — DELETE     — Elimina o caracter.
0D — ENTER      — Retorno de linha.
0F — GRAPHICS   — Saída do programa.
```

"Delete" apaga o caracter que se encontra por baixo do cursor, e desloca este uma posição para a esquerda. "Enter" desloca o cursor para a extremidade esquerda da linha imediatamente a seguir. 'Graphics" retorna ao comando Basic, deixando a imagem intacta a fim de permitir a sua passagem para uma impressora recorrendo à ordem COPY.

Resolvemos já alguns problemas, e podemos definir o fluxograma a partir do qual construiremos o programa. Na figura da página seguinte apresentamos este fluxograma.



A partir deste fluxograma poderemos escrever o programa de uma forma sistemática, tratando separadamente de cada uma das funções descritas naquele. Observemos portanto a primeira "caixa" da figura.

INÍCIO: DEFINIR PRINT AT 0,0;

Como já decidimos, a posição do cursor sob a forma de coordenadas AT será guardada em bytes de dados designados PRINT (6D00 hexadecimal). Logicamente, portanto, é necessário no caso da primeira caixa do fluxograma carregar 00 em PRINT (número de coluna) e PRINT+1 (número de linha).

```
6D05  RESET   LD HL,0000          21  00  00
              LD (PRINT),HL       22  00  6D
```

Definir posição de impressão de atributos

Necessitamos aqui de determinar o endereço do byte correspondente no ficheiro de atributos tendo em conta as coordenadas AT. Para tal multiplicaremos o número de linha por 32, somamos o resultado ao número de coluna e finalmente adicionamos o total a ATTRBS, o início do ficheiro de atributos.

```
6D0B  ATRSET  LD DE,(PRINT)       ED 5B 00 6D
              LD C,E              4B
              LD E,D              5A
              LD D,00             16 00
              LD HL,ATTRBS        21 00 58
              LD B,20             06 20
        MULT32 ADD HL,DE          19
              DJNZ MULT32         10 FD
              LD E,C              59
              ADD HL,DE           19
              LD (ATTRB),HL       22 02 6D
```

A última instrução refere-se a 6D02, que como dissemos é ATTRB, a posição de impressão no ficheiro de atributos. Não há necessidade de dar entrada ao programa à medida que o estamos a desenvolver — o leitor deverá apenas tentar compre-

endê-lo, e no fim trataremos da sua introdução na máquina e da sua execução.

Colocar o cursor no visor

Esta tarefa é bastante simples. HL contém a posição de impressão no ficheiro de atributos, porque foi essa posição que precisamente acabou de calcular. Basta-nos portanto passar a um o bit 7, que corresponde a FLASH, em (HL).

```
6D22  KEYGET   LD A,00          3E 00
               LD HL,LAST K     31 08 5C
               LD (HL),00       36 00
       KEYTST  CP (HL)          BE
               JR Z,KEYTST      28 FD
               LD A,(HL)        7E
               LD (KEY),A       32 04 6D
```

O valor da tecla premida é portanto armazenado em "KEY" (6D04).

Eliminar cursor

Agora eliminamos simplesmente o cursor, usando a instrução RES.

```
6D30  NOCURS   LD HL,(ATTRB)    2A 02 6D
               RES 7,(HL)       CB BE
```

A tecla existe na tabela?

Devemos agora determinar se uma dada tecla, por exemplo ENTER ou DELETE, existe na tabela, ou se a tecla premida corresponde simplesmente a um código que deve ser impresso. Para tal usaremos a tabela anteriormente listada, que se encontrará em memória a partir da posição "TABLE". O processamente será o seguinte:

1. Apontar DE para o início da tabela, 'TABLE" (6DAC).
2. Verificar se (DE), o código da tabela, é igual ao código

RES (x),B
da tecla premida, "KEY".
3. Se for, incrementar DE.
4. Ler os dois bytes seguintes da tabela, e passar os respectivos valores para HL. Em seguida executar um salto para este endereço contido em HL.
5. Se o código na tabela não é igual ao da tecla premida, incrementar três vezes DE a fim de ler o código seguinte da tabela.
6. Se o byte em causa contiver o valor FF (correspondente ao final da tabela), imprimir o código da tecla premida; se não, voltar ao passo 2.

```
6D35  SEARCH   LD DE,TABLE      11 AC 6D
               LD HL,KEY        21 04 6D
       TSTKEY  LD A,(DE)        1A
               CP (HL)          BE
               JR NZ,NEXBYT     20 07
               INC DE           13
               EX DE,HL         EB
               LD E,(HL)        5E
               INC HL           23
               LD D,(HL)        56
               EX DE,HL         EB
               JP (HL)          E9
       NEXBYT  INC DE           13
               INC DE           13
               INC DE           13
               CP FF            FE FF
               JR NZ,TSTKEY     20 EE
```

As caixas correspondentes a teclas especiais serão tratadas mais adiante.

Imprimir um caracter

No capítulo I encontrámos a instrução RST 10 que permite imprimir muito facilmente um caracter no visor.
Utilizaremos o código de comando 16 e a RST 10 para imprimir portanto o código correspondente à tecla premida.

```
6D4D  PUT KEY  LD DE,(PRINT)    ED 5B 00 6D
               LD A,16          3E 16
               RST 10           D7
               LD A,D           7A
               RST 10           D7
               LD A,E           7B
               RST 10           D7
               LD A,(HL)        7E
               RST 10           D7
```

Deslocar o cursor para a direita

Depois de ter sido impresso um caracter, o cursor é deslocado uma posição para a direita. Se, antes de ser deslocado, se encontrava na coluna 32, é movido para a primeira coluna da linha seguinte carregando 00 em 6D00 (número de coluna). A execução salta para esta linha sempre que se carrega na tecla "cursor para a direita". Depois de o cursor ter sido deslocado é executado um salto para ATRSET, a segunda caixa descrita.

```
6D5A  RIGHT    LD HL,6D00       21 00 6D
               INC (HL)         34
               LD A,20          3E 20
               CP (HL)          BE
               JR NZ,ATRSET     20 A8
6D63  NEXLIN   LD (HL),00       36 00
```

Deslocar o cursor para baixo

O cursor é movido para baixo uma posição a menos que se encontre na 22.ª linha do visor. É então executado um salto para ATRSET.

```
6D69  DOWN     LD HL,6D01       21 01 6D
               LD A,15          3E 15
               CP (HL)          BE
               JP Z,ATRSET      CA 0B 6D
               INC (HL)         34
               JP ATRSET        C3 0B 6D
```


Deslocar o cursor para cima

O cursor é movido uma posição para cima a menos que se encontre na primeira linha do visor. Em seguida é executado um salto para ATRSET.

```
6D76  CUR UP   LD HL,6D01       21 01 6D
               LD A,00          3E 00
               CP (HL)          BE
               JP Z,ATRSET      CA 0B 6D
               DEC (HL)         35
               JP ATRSET        C3 0B 6D
```

Deslocar o cursor para a esquerda

O cursor é movido para a esquerda a menos que se encontre na coluna zero, caso em que 6D00 (número de coluna) é carregado com 1F hexadecimal (31 decimal), o número da última coluna, e é executado um salto para CUR UP. Se o cursor se moveu para a esquerda, é executado um salto para ATRSET.

```
6D83  LEFT     LD HL,6D00       21 00 6D
               DEC (HL)         35
               LD A,FF          3E FF
               CP (HL)          BE
               JP NZ,ATRSET     C2 0B 6D
               LD (HL),1F       36 1F
               JP CUR UP        63 76 6D
```

Imprimir um espaço

Trata-se de uma rotina de "apagar" um caracter. Usando RST 10, imprime-se um espaço nas coordenadas em uso, sendo em seguida o cursor deslocado para trás uma posição saltando para LEFT.

```
6D94  DELETE   LD DE,(6D00)     ED 5B 00 6D
               LD A,16          3E 16
               RST 10           D7
```

```
LD A,D          7A
RST 10          D7
LD A,E          7B
RST 10          D7
LD A,20         3E 20
RST 10          D7
JP LEFT         C3 83 6D
```

Retorno ao comando Basic

Quando se carrega na tecla 9 juntamente com SHIFT (GRAPHICS), é executado um retorno ao comando Basic;

```
6DA5  EXIT    RET           C9
```

Colocar o cursor na extremidade esquerda

Isto nem sequer é feito... A tarefa em causa é deixada à fase final de RIGHT, designada por NEXLIN (6D63). A execução salta para esta rotina sempre que se carrega em Enter.

```
6DA6  ENTER   LD HL,6D00      21 00 6D
              JP NEXLIN       C3 63 6D
```

Tabela de teclas

Esta é a última questão a tratar antes de começarmos a introduzir o programa na máquina. Trata-se da tabela de endereços de teclas especiais.

```
6DAC  TABLE   DEFB 07 'EDIT'     07
              DEFB 056D          05 6D
              DEFB 08 'LEFT'     08
              DEFB 836D          83 6D
              DEFB 09 'RIGHT'    09
              DEFB 5A6D          5A 6D
              DEFB 0A 'DOWN'     0A
              DEFB 696D          69 6D
              DEFB 0B 'UP'       0B
```



```
              DEFB 766D          76 6D
              DEFB 0C 'DELETE'   0C
              DEFB 946D          94 6D
              DEFB 0D 'ENTER'    0D
              DEFB A66D          A6 6D
              DEFB 0F 'GRAPHICS' 0F
              DEFB FF 'END'      FF
```

Acabamos assim de estudar toda a listagem. Vou apresentar agora o programa completo, para facilitar ao leitor a sua consulta.

```
6D05  21 00 00      LD    HL,0000
6D08  22 00 6D      LD    (6D00),HL
6D0B  ED 5B 00 6D   LD    DE,(6D00)
6D0F  4B            LD    C,E
6D10  5A            LD    E,D
6D11  16 00         LD    D,00
6D13  21 00 58      LD    HL,5800
6D16  06 20         LD    B,20
6D18  19            ADD   HL,DE
6D19  10 FD         DJNZ  6D18
6D18  19            ADD   HL,DE
6D19  10 FD         DJNZ  6D18
6D1B  59            LD    E,C
6D1C  19            ADD   HL,DE
6D1D  22 02 6D      LD    (6D02),HL
6D20  CB FE         SET   7,(HL)
6D22  3E 00         LD    A,00
6D24  21 08 5C      LD    HL,5C08
6D27  36 00         LD    (HL),00
6D29  BE            CP    (HL)
6D2A  28 FD         JR    Z,6D29
6D2C  7E            LD    A,(HL)
6D2D  32 04 6D      LD    (6D04),A
6D30  2A 02 6D      LD    HL,(6D02)
6D33  CB BE         RES   7,(HL)
6D35  11 AC 6D      LD    DE,6DAC
6D38  21 04 6D      LD    HL,6D04
6D3B  1A            LD    A,(DE)
```

```
6D3C BE          CP   (HL)
6D3D 20 07       JR   NZ,6D46
6D3F 13          INC  DE
6D40 EB          EX   DE,HL
6D41 5E          LD   E,(HL)
6D42 23          INC  HL
6D43 56          LD   D,(HL)
6D44 EB          EX   DE,HL
6D45 E9          JP   (HL)
6D46 13          INC  DE
6D47 13          INC  DE
6D48 13          INC  DE
6D49 FE FF       CP   FF
6D4B ?0 EE       JR   NZ,6D3B
6D4D ED 5B 00 6D LD   DE,(6D00)
6D51 3E 16       LD   A,16
6D53 D7          RST  10
6D54 7A          LD   A,D
6D55 D7          RST  10
6D56 7B          LD   A,E
6D57 D7          RST  10
6D58 7E          LD   A,(HL)
6D59 D7          RST  10
6D5A 21 00 6D    LD   HL,6D00
6D5D 34          INC  (HL)
6D5E 3E 20       LD   A,20
6D60 BE          CP   (HL)
6D61 20 A8       JR   NZ,6D0B
6D63 36 00       LD   (HL),00
6D65 00          NOP
6D66 00          NOP
6D67 00          NOP
6D68 00          NOP
6D69 21 01 6D    LD   HL,6D01
6D6C 3E 15       LD   A,15
6D6E BE          CP   (HL)
6D6F CA 0B 6D    JP   Z,6D0B
6D72 34          INC  (HL)
6D73 C3 0B 6D    JP   6D0B
```



```
6D76 21 01 6D    LD   HL,6D01
6D79 3E 00       LD   A,00
6D7B BE          CP   (HL)
6D7C CA 0B 6D    JP   Z,6D0B
6D7F 35          DEC  (HL)
6D80 C3 0B 6D    JP   6D0B
6D33 21 00 6D    LD   HL,6D00
6D86 35          DEC  (HL)
6D87 3E FF       LD   A,FF
6D89 BE          CP   (HL)
6D8A C2 0B 6D    JP   NZ,6D0B
6D8D 36 1F       LD   (HL),1F
6D8F 00          NOP
6D90 00          NOP
6D91 C3 76 6D    JP   6D76
6D94 ED 5B 00 6D LD   DE,(6D00)
6D98 3E 16       LD   A,16
6D9A D7          RST  10
6D9B 7A          LD   A,D
6D9C D7          RST  10
6D9D 7B          LD   A,E
6D9E D7          RST  10
6D9F 3E 20       LD   A,20
6DA1 D7          RST  10
6DA2 C3 83 6D    JP   6D83
6DA5 C9          RET
6DA6 21 00 6D    LD   HL,6D00
6DA9 C3 63 6D    JP   6D63
```

Introduza agora os códigos hexadecimais indicados na coluna da direita em declarações DATA do carregador Basic.

Imediatamente a seguir a estes dados introduza os seguintes códigos, também em declarações DATA (mas não inclua os endereços indicados no lado esquerdo!):

```
6DAC 07 05 6D 08 83 6D 09 5A
6DB4 6D 0A 69 6D 0B 76 6D 0C
6DBC 94 6D 0D A6 6D 0F A5 6D
6DC4 FF 00 00 00 00 00 00 00
```

```
6DCC  00  00  00  00  00  00  00  00
6DD4  00  00  00  00  00  00  00  00
6DDC  00  00  00  00  00  00  00  00
6DE4  00  00  00  00  00  00  00  00
```

Como começamos a partir do endereço 6D 00 em vez de 70 00 como é habitual, teremos de alterar a linha 10 para:

```
10 LET Address = 27909
```

Para o caso de o leitor ter cometido um erro na introdução dos códigos (tal como os governos se enganam ao fazer estatísticas...), é melhor gravar em cassette o carregador antes de tentar executar o programa. Depois de o fazer, pode finalmente executar o carregador premindo na tecla RUN; em seguida pode executar o nosso programa escrevendo:

```
PRINT AT 0,0;: RANDOMIZE USR 27909
```

Como estudámos detalhadamente todo o programa, o leitor deve conhecer já o modo de o utilizar. Como referência indicamos agora os comandos:

SHIFT 5 a SHIFT 8	—	Comandos do cursor
DELETE	—	Apaga caracter
EDIT	—	Reenvia o cursor para 0,0.
GRAPHICS	—	Retorno ao comando Basic.

IV

OPERAÇÕES LÓGICAS

Já falámos sobre bits e sobre as instruções que nos permitem manipulá-los individualmente. Neste capítulo veremos algumas novas instruções que também alteram os bits, chamadas *lógicas*. Este nome deve-se ao facto de todas elas se fundamentarem na chamada lógica *booleana*. Neste momento o leitor provavelmente não tem qualquer ideia sobre o que possa ser a lógica booleana, mas de facto trata-se simplesmente de uma designação que cobre certas instruções lógicas, tal como "linguagem estruturada" descreve uma linguagem com estrutura como a Pascal ou a FORTH.

Em Basic dispomos das instruções OR, AND e NOT. Usamo-las em declarações IF — THEN a fim de tomar decisões. Estas três instruções são também "booleanas", mas empregam-se aqui num sistema muito mais flexível. Não podemos construir equivalentes à declaração IF — THEN em código-máquina, mas é ainda possível obter efeitos semelhantes usando o que já conhecemos sobre saltos condicionais (por exemplo JP Z,nnnn) e acrescentando as "instruções lógicas" estudadas em seguida.

Observemos um pouco os princípios aplicados nas instruções Basic OR e AND. Estas declarações, como sabemos, actuam do seguinte modo:

1. x AND y: Se a condição x for verdadeira e a condição y também o for, o resultado será verdadeiro. Por exemplo, 1=1 AND 10=10 é verdadeiro, e "H"="H" AND a$=a$ também,

ao contrário de 2=3 AND 1=1 ou "H"="H" AND "B"="C", que são falsas.

2. x OR y: Se a condição x, a condição y ou ambas forem verdadeiras, o resultado será igualmente verdadeiro. Por exemplo, 2=2 OR 3=1 é verdadeiro, tal como a a$=a$ OR b$=c$; mas 1=2 OR 5=0 é sempre uma condição falsa, tal como "C"="Dé' OR "B"="A".

Em código-máquina não trabalhamos com cadeias ou variáveis, e sim com "bits" que podem assumir o valor "0" ou "1". Podemos usar a linguagem Basic para obter este tipo de processamento. O computador sabe dizer-nos se uma dada coisa é verdadeira ou não. Experimente escrever:

```
PRINT 1=1
```

Esta afirmação pode parecer estúpida, mas o computador imprime um "1". Porquê? Porque sabe que 1 é de facto igual a 1, e exprime a verdade lógica desta afirmação através do número "1". Experimente agora escrever:

```
PRINT 1=0
```

Todos sabemos que esta igualdade não é verdadeira, e o mesmo se passa com o Spectrum. Dá-nos a conhecer o facto imprimindo um zero. Podemos portanto concluir que a máquina usa zeros e uns para exprimir a verdade de uma afirmação lógica, do modo que se segue:

0 = falso
1 = verdadeiro

Experimentemos agora escrever:

```
LET falso=0: LET verdadeiro=1

PRINT verdadeiro AND verdadeiro
```


Como verdadeiro=1, o computador imprime um "1". Isto deve-se ao facto de a variável "verdadeiro" ser equivalente a uma expressão verdadeira. Tendo em conta que a variável "falso"=0 e que zero significa "falso", veja se consegue prever o que acontecerá ao escrever:

```
PRINT falso AND verdadeiro
```

A máquina imprime um zero (indicando "falso") porque as duas condições não são "verdadeiras". Podemos construir uma tabela (a que se chama "tabela de verdade") para a instrução AND:

AND		RESULTADO
falso	falso	falso
verdadeiro	verdadeiro	verdadeiro
falso	verdadeiro	falso
verdadeiro	falso	falso

Podemos agora converter esta tabela de modo a usar apenas zeros e uns, recordando que "1" significa verdadeiro e "0" significa falso:

AND:			
	0	0	0
	0	1	0
	1	0	0
	1	1	1

Se fizessemos as mesmas experiências com a função OR, obteríamos a seguinte tabela de verdade:

OR:	0	0	0
	0	1	1
	1	0	1
	1	1	1

Estas tabelas correspondem exactamente ao funcionamento das instruções OR e AND em código-máquina, exceptuando o facto de apenas actuarem sobre dois conjuntos de oito bits.

Se aplicarmos a instrução AND ao Acumulador e a outro registo, a CPU compara os dois bits da mesma ordem de cada um dos registos. O resultado é necessariamente um "1" ou um "0". A tabela que já construímos permite-nos prever o resultado desta operação. Por exemplo:

```
AND:        1 1 0 1 0 1 1 0
            0 1 1 0 1 1 0 1
            ---------------
Resultado   0 1 0 0 0 1 0 0
```

Note que os únicos bits do resultado que se encontram ao valor "1" são os que correspondem à comparação de dois bits com esse valor. Vejamos ainda um outro exemplo; convirá ao leitor seguir o resultado bit a bit tentando compreender o que se passa:

```
AND:        0 0 1 1 1 1 0 1
            1 1 1 1 0 1 0 0
            ---------------
Resultado   0 0 1 1 0 1 0 0
```

Já compreendeu? Veja se consegue descobrir os resultados sozinho:


```
AND:        1 0 1 1 0 1 1 1
            1 0 1 0 1 1 0 1
            ---------------
Resultado           ?
```

Recordando que só 1 AND 1 produz o resultado "1", é óbvio que a resposta é:

1 0 1 0 0 1 0 1

Voltemos agora à tabela de verdade da função OR:

OR:	0	0	0
	1	0	1
	0	1	1
	1	1	1

Se portanto aplicarmos a instrução OR aos dois bytes que se seguem:

```
0 1 0 1 1 0 0 1
0 1 1 1 0 1 0 1
```

Obteremos o resultado:

0 1 1 1 1 1 0 1

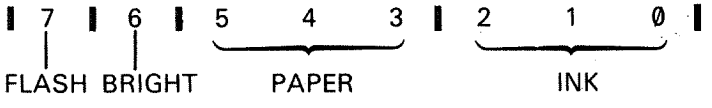
Note que os únicos bits do resultado iguais a zero são os que correspondem à comparação de dois zeros.

A instrução AND toma a forma de "AND r" (onde r indica um registo). Funciona comparando logicamente os bits do registo r com os bits correspondentes do Acumulador, guardando em seguida o resultado neste último. Do mesmo modo, "OR r" compara logicamente os bits do registo r com os bits correspondentes do Acumulador. Vejamos agora uma lista de todas as instruções AND e OR, juntamente com os respectivos códigos:

AND A	A7
AND B	A0
AND C	A1
AND D	A2
AND E	A3
AND H	A4
AND L	A5
AND (HL)	A6
AND (IX+dis)	DD A6 dis
AND (IY+dis)	FD A6 dis
AND nn	E6 nn

OR A	B7
OR B	B0
OR C	B1
OR D	B2
OR E	B3
OR H	B4
OR L	B5
OR (HL)	B6
OR (IX+dis)	DD B6 dis
OR (IY+dis)	ED B6 dis
OR nn	F6 nn

Uma das utilidades da função AND consiste em verificar certos bits de um byte. Consideremos como exemplo um byte de atributos, como fizemos no Capítulo III.



O nosso objectivo consiste em alterar todos os caracteres em INK 0 para INK 2 (ou seja passá-los de negros para vermelhos). Obviamente necessitaremos de um ciclo para processar cada byte de atributos. Mas o código crucial encontrar-se-á no interior do ciclo. Em primeiro lugar, queremos considerar os três bits menos significativos (ou seja os bits 0, 1 e 2). É aqui que AND mostra a sua utilidade. Observe o seguinte:

Atributos:	1	1	0	1	0	1	1	1
Valor 07:	0	0	0	0	0	1	1	1
Resultado:	0	0	0	0	0	1	1	1

O leitor nota certamente que só os bits 0, 1 e 2 do byte de atributos surgem com o valor um no resultado. Isto deve-se ao facto de ter sido executada uma função AND com o número 07 hexadecimal (equivalente, em binário, a 00000111). Qualquer que seja o valor dos restantes bits do byte de atributos, são necessariamente desprezados. Consideremos ainda um outro exemplo:

Atributos:	1	1	1	1	0	1	1	0
Valor 07:	0	0	0	0	0	1	1	1
Resultado:	0	0	0	0	0	1	1	0

O reñul)ado (neste caso 06 hexadecimal porque a função AND executada sobre o primeiro bit produz zero. Vemos portanto que quando usamos a função AND sobre um byte, os bits iguais a zero no número que é comparado "mascaram" quaisquer bits desse byte que sejam iguais a um. Este processo permite-nos portanto "esconder" certos bits.

Se então utilizarmos a instrução AND 07 sobre o byte de atributos, obtemos apenas a cor de INK, passando todos os outros bites necessariamente a zero. Podemos em seguida comparar este valor de INK usando as flags, verificando se deve ser alterado:

7000	21	00	58	LD	HL,5800
7003	7E			LD	A,(HL)
7004	E6	07		AND	07
6006	20	02		JR	NZ,700A
7008	CB	CF		SET	1,A
700A	77			LD	(HL),A
700B	23			INC	HL
700C	7C			LD	A,H
700D	FE	5B		CP	5B
700F	20	F2		JR	NZ,7003
7011	C9			RET	

Se observarmos o programa anterior passo a passo, veremos que primeiro se carrega em HL o endereço do início do ficheiro de atributos. Em seguida carrega-se no Acumulador o byte de atributos em causa. Depois usa-se a instrução AND 07, que passa a zero todos os bits excepto os que indicam a cor INK. Se o byte não for zero, é executado um salto relativo para 700A, porque nessas condições a cor INK não é preto. Se o for, o bit 1 do Acumulador é passado a 1, passando a cor para vermelho.

Verifiquemos o funcionamento deste programa:

Comece por preparar o carregador Basic. Em seguida altere a linha 80, introduzindo os códigos apresentados na listagem do programa.

Em seguida carregue na tecla RUN, escrevendo depois:

INK 2: LIST : RANDOMIZE USR endereço

Deverá verificar (se a cor de fundo não for vermelha...) que os caracteres negros passam imediatamente para vermelho!

Para demonstrar que apenas são alterados os caracteres a negro, escreva:

INK 0: LIST : INK 2 : LIST : RANDOMIZE USR endereço

Um uso muito habitual da instrução OR consiste em passar certos bits de um byte para o valor um. Suponhamos que nos interessa passar ao valor um os bits seis e sete de um byte de atributos a fim de imprimir um caracter em BRIGHT e FLASH. Podemos fazê-lo partindo de um byte que possua apenas ao valor um os bits seis e sete, e executando depois a função OR entre este byte e o de atributos. Se introduzirmos isto num ciclo, podemos alterar todo o visor.

```
Atributos:  0 0 1 0 1 0 0 1
Valor E0:   1 1 0 0 0 0 0 0
            -----------------
Resultado:  1 1 1 0 1 0 0 1
```

Não são aqui apenas alterados os bits seis e sete, o que corresponde exactamente ao que pretendíamos. Vamos agora examinar um programa que coloca todo o visor em FLAH e BRIGHT.


```
7000  21  00  58    LD    HL,5800
7003  7E            LD    A,(HL)
7004  F6  E0        OR    E0
7006  77            LD    (HL),A
7007  23            INC   HL
7008  7C            LD    A,H
7009  FE  5B        CP    5B
700B  20  F6        JR      NZ,7003
700D  C9            RET
```

O modo de trabalho deste programa é muito simples. Carrega-se em HL o endereço inicial do ficheiro de atributos, e em seguida carrega-se o byte correspondente em A. Os bits seis e sete são passados ao valor 1 (através da instrução OR E0), carregando-se o byte de atributos com o novo valor. Finalmente incrementa-se HL, verifica-se se atingiu o final do ficheiro de atributos, e se tal não tiver acontecido passa ao byte de atributos que se segue.

Verifiquemos a execução do programa. Utilize a seguinte linha 80 no programa carregador:

80 DATA "2100587EF6E077237CFE5B20F6C9"

Em seguida, depois de carregar na tecla RUN, escreva no teclado:

LIST : RANDOMIZE USR endereço

Verificará que o visor passa instantaneamente a FLASH e BRIGHT.

Algumas experiências

O leitor pode agora fazer algumas das experiências que se seguem:

1. Utilize valores diferentes de E0 na instrução OR, e tente descobrir o porquê dos resultados obtidos.
2. Experimente usar AND em vez de OR com diferentes valores. Descubra as razões dos seus efeitos.
3. Em vez de usar OR para passar a 1 os bits seis e sete, tente modificar o programa de modo a usar SET, a instrução estudada no Capítulo anterior.

OR Exclusivo

Existe ainda uma outra instrução lógica. Chama-se OR Exclusivo e é bastante semelhante à instrução OR. É usada pelo Spectrum quando se utiliza OVER. O que se segue deveria ter algum sentido para o leitor.

```
PAPER  + PAPER  = PAPER
INK    + PAPER  = INK
PAPER  + INK    = INK
INK    + INK    = PAPER
```

São estes os resultados quando OVER tem o valor 1. Utiliza-se a instrução XOR quando se marca um pixel ou se imprime um caracter no visor. Se pensarmos em INK como valendo 1 e PAPER como correspondendo a zero, notaremos que a tabela anterior pode ser expressa do seguinte modo:

```
0 0 0
1 0 1
0 1 1
1 1 0
```

É esta a tabela lógica da função XOR, consistindo a única diferença em relação à OR no facto de dois "uns" produzirem um "zero" em vez de um "um". Vejamos uma lista dos códigos de operação das instruções XOR:

```
XOR A          AF
XOR B          A8
XOR C          A9
XOR D          AA
XOR E          AB
XOR H          AC
XOR L          AD
XOR (HL)       AE
XOR (IX+dis)   DD AE dis
XOR (IY+dis)   FD AE dis
XOR nn         EE nn
```


Experimente usar XOR no último programa e observe os seus efeitos. Use-a quando tem alguma coisa em FLASH ou BRIGHT no visor. Se não compreender estes efeitos, realize as "somas" dos bits e veja se a questão se torna mais clara para si. Se tiver dificuldades, consulte a tabela seguinte:

XOR			
	0	0	0
	0	1	1
	1	0	1
	1	1	0

Vejamos alguns exercícios simples:

1. 1 XOR 0 = ?

2. 1 XOR 1 = ?

3.
```
  1 0 1 1
XOR 0 1 1 0
      ?
```

4.
```
  1 1 0 0
  1 0 1 0
      ?
```

5.
```
  1 0 1 1 1 0 1 1
XOR 0 1 1 0 1 0 1 0
          ?
```

A última questão que iremos estudar neste capítulo é um método para "verificação do zero em dois bytes". No Capítulo II observámos um ciclo usando registos de um só byte. Quando desejamos realizar uma operação mais de 256 (decimal) vezes, necessitamos de utilizar como contador um par de registos. Um modo de o fazermos consiste em usar a instrução CP.

```
        LD HL,1000
PROG      ...
          ...
        LD A,H
```

```
CP 00
JR NZ,PROG
LD A,L
CP 00
JR NZ,PROG
RET
```

Como o leitor terá notado, H e L são verificados separadamente a fim de saber se ambos contêm zero. A instrução CP nn utiliza dois bytes, mas podemos poupar espaço neste caso usando AND. Se utilizarmos AND A, o Acumulador não é alterado mas as flags são configuradas de modo a representarem A. Vejamos porque razão não é alterado o conteúdo de A. Não se esqueça que "AND A" apenas executa a função AND entre o Acumulador e ele próprio, pelo que os valores dos bits comparados são sempre iguais.

```
1 1 0 0 1 0 1 0
1 1 0 0 1 0 1 0
1 1 0 0 1 0 1 0
```

Tenha em conta a tabela de verdade da função AND:

```
0 0 0
0 1 0
1 0 0
1 1 1
```

Como podemos ver, A não sofre qualquer modificação, apesar de as flags serem alteradas de modo a representarem o estado desse registo. Isto permite-nos tomar uma decisão válida, e portanto podemos substituir a instrução CP 00 por uma AND A.

```
        LD HL,1000
PROG    ...
        ...
        INC HL
        LD A,H
        AND A
        JR NZ,PROG
        LD A,L
        AND A
        JR NZ,PROG
        RET
```

Mas ainda não fizemos a nossa descoberta mais interessante. Podemos comprimir ainda mais esta verificação de zeros, recorrendo à instrução OR. Se carregarmos H em A e fizermos OR L, poderemos descobrir qualquer bit que se encontre no valor 1, e nesse caso A não será igual a zero. Se pelo contrário todos os bits de HL forem iguais a zero, A apresentará também o resultado zero. Vejamos uma forma modificada do mesmo programa:

```
PROG    LD HL,1000      21 00 10
START   ...
        ...
        INC HL          23
        LD A,H          7C
        OR L            B5
        JR NZ,START     20 ??
        RET             C9
```

Experimentemos agora o resultado deste programa, usando-o para imprimir 200 (hexadecimal) asteriscos no visor.

```
7000  21  00  20      LD    HL,2000
7003  3E  2A          LD    A,2A
7005  D7              DST   10
7006  23              INC   HL
7007  7C              LD    A,H
7008  B5              OR    L
7009  20  FA          JR      NZ,7005
700B  C9              RET
```

Introduza o programa na máquina utilizando na linha de DATA do carregador os códigos hexadecimais indicados à frente

dos endereços na listagem anterior. Em seguida carregue em

RUN

e depois escreva:

PRINT AT 0,0;: RANDOMIZE USR endereço

A máquina imprimirá em seguida exactamente 200 (hexadecimal) asteriscos.

V

ROTAÇÕES

Este capítulo tratará das rotações. Estas instruções servem basicamente para deslocar os bits de um registo para a esquerda ou para a direita, em diversas combinações envolvendo a flag "carry". O melhor modo de representar este modo de funcionamento consiste em recorrer a diagramas, mas apresento também uma explicação escrita além dos códigos de operação de cada tipo.

As instruções de rotação não têm uma utilidade óbvia, mas podem ser usadas essencialmente em aplicações em que necessitamos de "construir" um byte bit a bit, ou verificar um byte bit a bit. Estas instruções podem igualmente ser usadas para multiplicação por potências de dois (2, 4, 6, 8, etc.), existindo algumas que permitem usar a forma Decimal Codificado em Binário, que explicarei mais adiante. A maior parte das instruções possuem uma variante que actua especificamente sobre o Acumulador, além de outra que actua sobre os registos simples habituais, incluindo o Acumulador. Estas últimas utilizam códigos de operação de dois bytes, e portanto para o Acumulador existem duas instruções diferentes, tendo uma delas um único byte e a outra dois bytes. É óbvio que em situações em que se pretende rodar o Acumulador convirá preferir a versão de um só byte, a fim de poupar alguma memória.

A instrução geral tem o seguinte formato:

RLC r

onde r pode ser qualquer dos registos A, B, C, D, E, H, L, (HL), (IX + dis) ou (IY + dis).

Esta instrução roda o registo ou byte "r" para a esquerda; o bit mais significativo (bit 7) é rodado de modo a passar para o bit menos significativo. O bit ou flag "carry" é passado a um ou a zero de acordo com o conteúdo do bit 7 no início da operação. Isto permite ao programador determinar o que se encontrava no bit sete antes da rotação, verificando o estado da Flag "carry" após a instrução de rotação.

Vejamos uma lista das instruções desta forma, começando pelo de um único byte, correspondente ao registo A:

RLCA	07
RLC A	CB 07
RLC B	CB 00
RLC C	CB 01
RLC D	CB 02
RLC E	CB 03
RLC H	CB 04
RLC L	CB 05
RLC (HL)	CB 06
RLC (IX + dis)	DD CB dis 06
RLC (IY + dis)	FD CB dis 06

RL r

Esta instrução roda o registo ou byte "r" para a esquerda através do bit de transporte. O bit sete é deslocado para o bit "carry", e o conteúdo deste bit é deslocado para o bit zero do registo rodado.

Vejamos a lista de instruções:

RLA	17
RL A	CB 17
RL B	CB 10
RL C	CB 11
RL D	CB 12
RL H	CB 14
RL L	CB 15
RL (HL)	CB 16
RL (IX + dis)	DD CB dis 16
RL (IY + dis)	FD CB dis 16

RRC r

Esta operação é semelhante à instrução RLC r, mas a rotação é realizada para a direita. Desta vez o bit zero será passado para o bit sete e para a flag "carry":

RRCA	0F
RRC A	CB 0F
RRC B	CB 08
RRC C	CB 09
RRC D	CB 0A
RRC E	CB 0B
RRC H	CB 0C
RRC L	CB 0D
RRC (HL)	CB 0E
RRC (IX + dis)	DD CB dis 0E
RRC (IY + dis)	FD CB dis 0E

RR r

Trata-se da operação inversa de RL r. Neste caso o bit zero de r é deslocado para a flag "carry"; ¨nro o valor inicial desta passado para o bit sete de "r":

RRA	1F
RR A	CB 1F
RR B	CB 18
RR C	CB 19
RR D	CB 1A
RR E	CB 1B
RR H	CB 1C
RR L	CB 1D
RR (HL)	CB 1E

RR (IX+dis)	DD CB dis 1E
RR (IY+dis)	FD CB dis 1E

Por vezes é preferível "deslocar" um registo em vez de o "rodar". Em vez de "rodar" o bit zero passando-o para o bit sete ou vice-versa, o bit sete pode perdido ou deixado como está ao mesmo tempo que é passado para o bit seguinte. Estas instruções de deslocamento dos bits não possuem uma instrução específica para o Acumulador; são todas formadas por dois bytes, começando pelo prefixo CB (hexadecimal).

SLA r

Esta instrução desloca simplesmente todos os bits para a esquerda. O bit sete é passado para a flag "carry", sendo introduzido um zero no bit zero.

SLA A	CB 27
SLA B	CB 20
SLA C	CB 21
SLA D	CB 22
SLA E	CB 23
SLA H	CB 24
SLA L	CB 25
SLA (HL)	CB 26
SLA (IX+dis)	DD CB dis 26
SLA (IY+dis)	FD CB dis 26

SRA r

Esta instrução desloca todos os bits do registo ou byte para a direita. O bit zero é deslocado para o bit "carry". O bit sete é copiado no bit seis, mas não é alterado.

SRA A	CB 2F
SRA B	CB 28
SRA C	CB 29
SRA D	CB 2A
SRA E	CB 2B

SRA H	CB 2C
SRA L	CB 2D
SRA (HL)	CB 2E
SRA (IX+dis)	DD CB dis 2E
SRA (IY+dis)	FD CB dis 2E

SRL r

Esta instrução desloca os bits para a direita, mas ao contrário do que é habitual não lhe corresponde qualquer equivalente que desloque os bits para a esquerda. Existe até um espaço livre nos códigos das rotações onde caberiam estas instruções, o que indica que deveriam fazer parte da CPU Z80 mas que os fabricantes desta não conseguiram fazer funcionar. De qualquer modo, SRL desloca os bits para a direita, colocando um zero no bit sete e passando o bit zero para a flag "carry".

SRL A	CB 3F
SRL B	CB 38
SRL C	CB 39
SRL D	CB 3A
SRL E	CB 3B
SRL H	CB 3C
SRL L	CB 3D
SRL (HL)	CB 3E
SRL (IX+dis)	DD CB dis 3E
SRL (IY+dis)	FD CB dis 3E

DECIMAIS

Antes de estudarmos as instruções RLD e RRD devemos compreender a forma Decimal em Codificação Binária (BCD). Esta forma de representar números é muito importante em computadores. É uma espécie de "interface" entre os seres humanos e a representação binária, que tenta ligar ambos. A maior parte dos indicadores LED aceitam os algarismos sob a forma BCD. Um único algarismo BCD é bastante fácil de compreender; corresponde a quatro algarismo binários, cujo valor não ultrapassa nove (senão não seria decimal...).

Quando pretendemos armazenar números numa forma deci-

mal podemos usar a rotação BCD. Por exemplo, se quisermos guardar 99 sob a forma decimal, seria normalmente necessário converter este número para hexadecimal voltando depois a convertê-lo se quiséssemos, por exemplo, imprimi-lo em decimal no visor. Vejam (s 'o ˆo ñeıia represent°éo c número decimal 56:

Quatro bits mais representativos: 0 1 0 1 = 5
Quatro bits menos representativos: 0 1 1 0 = 6

Se quiséssemos pensar no número 56 como sendo hexadecimal, convertendo-o depois (veja o valor 56 hexadecimal no apêndice do Manual Spectrum), verificaríamos que corresponde a 86 decimal — mas estamos a considerar aqui 56 como decimal e não como hexadecimal.

Se se carregar um número hexadecimal no Acumulador é possível convertê-lo para BCD usando uma instrução DAA — Decimal Adjust Accumulator:

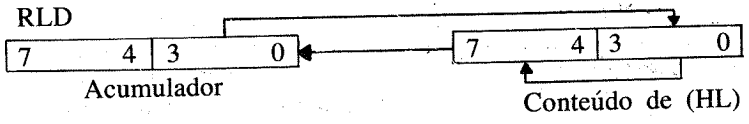
DAA 27

Imaginemos então que carregámos no Acumulador o número 43 hexadecimal. Se observarmos o Manual, verificaremos que 43 hexadecimal corresponde a 67 decimal. Se aplicarmos agora uma instrução DAA ao Acumulador o conteúdo deste passa de 43 para 67, um número BCD.

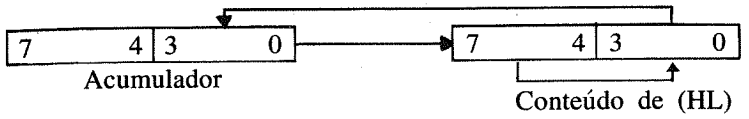
A instrução DAA só pode alterar decimais até 99, porque só podemos representar dois algarismos em cada registo.

Se somarmos dois números BCD, o resultado deverá ser ajustado podendo-se usar DAA para corrigir o resultado em BCD. Do mesmo modo, se se subtrairem dois números em BCD a instrução DAA corrigirá o resultado para BCD.

Vejamos então como funcionam as instruções RLD e RRD.



f instrução RLD roda os três blocos de quatro bits pela metade menos significativa do Acumulador e pelas duas metades da posição de memória indicada por HL, sempre para a esquerda. Estes quatro blocos de bits encontram-se obviamente na forma BCD, e dispõem de um nome próprio — "nibble". Vejamos agora o diagrama correspondente a RRD:



A instrução RRD actua no sentido contrário a RLD, deslocando o algarismo BCD mais significativo de (HL) para a metade menos significativa, passando esta para a metade menos significativa do Acumulador, e a menos significativa do Acumulador para a mais significativa de (HL).

Encerremos aqui este capítulo e esperamos que no próximo o leitor se distraia um pouco ao som de música...

VI

PORTS

Os computadores devem obviamente poder comunicar com o mundo exterior. O "mundo exterior" inclui normalmente o leitor, as impressoras, gravadores, teclados e visores — pelo menos no que se refere ao "mundo" do ZX Spectrum. Se não puder comunicar connosco o computador torna-se inútil, um pouco como uma aparelhagem vídeo que não esteja ligada a um receptor de televisão. Dividimos os modos de comunicação de um computador em duas categorias principais, as entradas e as saídas. As entradas são as informações recebidas pelo computador, por exemplo através do teclado. Pelo contrário, as saídas são as informações enviadas pelo computador por exemplo para um monitor vídeo. Para poder comunicar com o exterior, o sistema computador, quer se trate de uma máquina de lavar roupa programável ou do ZX Spectrum, utiliza aquilo que é designado tecnicamente por *ports*. Este nome (que equivale a *portos* em português) tem alguma lógica — podemos imaginar um navio transportando as informações deste para outro local. Obviamente, se olharmos para o interior do Spectrum não veremos navios movendo-se de um lado para o outro, mas o princípio de "funcionamento" é o mesmo. Se o leitor gostar de se entreter com o hardware, pode utilizar alguns dos "portos" livres no Spectrum — mas deve nesse caso ler o Capítulo XXIII do Manual Sinclair para dispor de informações sobre o assunto.

Mas os "ports" não servem apenas àqueles que se interessam pela electrónica. Não necessitamos de comprar equipamento

extra para sabermos o que fazem, já que dispomos de alguns ports muito úteis "à mão de semear" — os ports usados pelo Spectrum para comunicar connosco. Espero que o leitor tenha uma ideia de como se acedem os ports usando o comando Basic, mas começamos de qualquer modo por este assunto dado que constitui uma introdução bastante lógica ao tratamento feito em código-máquina. Em primeiro lugar deixe-me recordar-lhe alguns aspectos do problema. Normalmente podem existir até 256 ports num sistema Z80, mas devido a uma subtileza deste microprocessador é possível usar até 65536 ports em certas situações; voltaremos a este assunto mais adiante.

O Spectrum só utiliza na sua configuração original alguns destes ports, e aquele que iremos estudar imediatamente é o que produz sons e comunica a cor da margem ao visor.

Introduza na máquina este pequeno programa, e execute-o em seguida:

```
10 FOR i = 0 TO 7: OUT 254,1: PAUSE 3: NEXT i: GO TO 10
```

Como pode verificar, a margem do visor assume em sequência as oito cores disponíveis no Spectrum. O primeiro aspecto a notar é que estamos a utilizar o port 254. Este port encontra-se associado portanto à margem do visor, mas também ao microfone do ZX Spectrum e às tomadas EAR e MIC. A instrução OUT assume na linguagem Basic o seguinte formato:

```
OUT port,dado
```

No nosso exemplo, o port tem o número 254 e o dado é a cor da margem. Note que o port encontra-se porém dividido em várias partes. Cada uma delas é comandada por um bit ou bits diferentes. Vejamos para que servem os oito bits do port 254:

0 1 2	}	Cor de BORDER
3	–	Tomada MIC
4	–	Altifalante

5
6
7 } Não usados

Sabemos agora o que cada um dos bits faz; talvez já nos seja possível obter alguns sons. Antes do mais devemos compreender que um "1" no bit quatro liga o altifalante, e um "0" na mesma posição desliga-o. Se portanto alternarmos o um e o zero devemos obter um ruído. Introduza o programa seguinte para verificar o que se passa:

```
10 OUT 254,2↑4*1: OUT 254,2↑4*0: GO TO 10
```

Existem duas ordens OUT, uma que liga o altifalante e outra que o desliga. Se observar o programa, verificará que o primeiro dado fornecido é 2↑4*1. O dois é usado porque estamos a trabalhar em binário, base dois. O quatro indica o bit que nos interessa, e o um indica que desejamos ligar o altifalante. Do mesmo modo, na segunda declaração OUT, existe um zero que desliga o altifalante. Tal como se encontra, o programa é muito lento e o som consiste apenas nuns frágeis "clicks". Para tornar a execução mais rápida eliminamos as contas que fazem perder tempo ao computador, e substituimos as expressões dos dados pelos seus valores resultantes, 16 e zero, alterando portanto o programa do seguinte modo:

```
10 OUT 254,16: OUT 254,0: GO TO 10
```

Mas mesmo mais rápido, o ruído produzido é ainda muito fraco; estamos bastante longe dos "zaps", "pums" e "bongs" da máquina de jogos local. Para "aquecer" um pouco o ambiente devemos portanto concentrar os nossos esforços no código-máquina, agora que já adquirimos alguns conhecimentos sobre os ports.

Na linguagem-máquina do Z80 dispomos de instruções directamente relacionados com declarações Basic. Chegam até a ter os mesmos nomes, IN e OUT. Por exemplo, se quisermos enviar os dados presentes no Acumulador para o port 254, podemos simplesmente "dizer":

```
OUT (FE),A
```

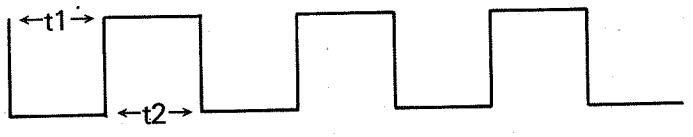

Ao contrário da instrução Basic, porém, colocamos o número do port entre parêntesis (FE, correspondente a 254 decimal). Isto deve-se ao facto de considerarmos o número de port como um endereço, e portanto para seguir o formato geral das instruções Z80 utilizamos parêntesis para significar que a informação, neste caso o conteúdo de A, é enviada para um endereço, o do port. Tentemos então traduzir o nosso programa Basic em código-máquina. Como no entanto esta linguagem é muito rápida teremos de introduzir pausas entre as instruções OUT, senão no preciso momento em que a membrana do altifalante consegue começar a mexer-se num sentido recebe ordens para se mover no outro, e o resultado líquido é que não produz som nenhum... Para introduzir estas pausas utilizaremos o registo B. Indicaremos em C o número de "clicks" a realizar. O nome verdadeiro destes clicks é, aliás, "ciclos". Passemos então ao programa.

Para o utilizar escreva os códigos hexadecimais no carregador Basic, e proceda do modo habitual. As duas estranhas instruções no início e no fim do programa não são erros: explicá-las-emos no próximo capítulo. Basicamente, servem para assegurar que o computador dedica toda a sua energia à produção dos sons, em vez de ignorar o assunto em cada 1/50 de segundo, preocupando-se com outras coisas como a leitura do teclado.

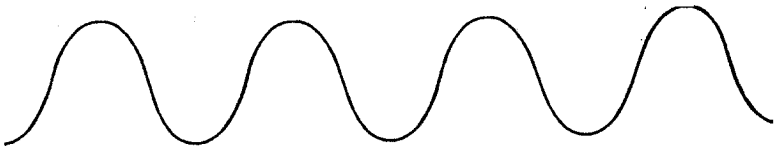
```
7000  F3        DI
7001  0E  FF    LD      C,FF
7003  3E  00    LD      A,00
7005  D3  FE    OUT       (FE),A
7007  06  C0    LD      B,C0
7009  10  FE    DJNZ    7009
700B  3E  18    LD      A,18
700D  D3  FE    OUT       (FE),A
F00F  06  C0    LD      B,C0
7011  10  FE    DJNZ    7011
7013  0D        DEC     C
7014  20  ED    JR        NZ,7003
7016  FB        EI
7017  C9        RET
```

Neste programa carregamos FF em C, de modo a serem produzidos 255 (decimal) ciclos. Isto define o comprimento da

nota. Os dois ciclos que utilizam B decidem o tempo durante o qual o altifalante é ligado e desligado. Ajustando estes valores em B podemos determinar a tonalidade do som. Vejamos um diagrama que ilustra a questão:



A duração de tl é decidida pelo primeiro ciclo B, e a duração de t2 pelo segundo ciclo B. O número de ciclos é determinado por C. Como se pode verificar, os ciclos têm uma forma quadrada, sendo por isso que lhes chamamos "ondas quadradas". Os diferentes sons produzem ondas de forma diferente. Uma flauta, por exemplo, tende a produzir uma onda do seguinte tipo:



Este som é muito mais suave. Infelizmente, no Spectrum, estamos limitados a ondas quadradas a menos que compremos um acessório apropriado; nestas condições o som é um tanto duro, pouco musical. Mas mesmo tendo em conta esta limitação é possível obter efeitos bastante bons.

Um último ponto relativo a este programa: o leitor terá notado que a margem do visor passa a negro. Isto deve-se ao facto de termos mantido os bits zero, um e dois iguais a zero, o que produz uma cor de margem negra.

O nosso programa funciona perfeitamente, mas é um tanto desnecessário porque existe uma rotina em ROM que faz a mesma coisa e é muito fácil de usar. Encontra-se a partir do endereço 03B5, e para usá-la basta carregar o número de ciclos em DE e a duração destes, ou a tonalidade, em HL. Experimente por exemplo o seguinte programa:


```
7000  21  00  03   LD     HL,0300
7003  11  00  02   LD     DE,0200
7006  CD  B5  03   CALL   03B5
7009  C9           RET
```

Conseguimos assim um modo muito mais rápido de obter um simples BEEP! Mas interessa-nos algo mais do que um simples BEEP dado que também o podemos obter em Basic; vejamos então como é possível obter sons mais interessantes.

O programa que se segue utiliza a rotina em ROM para produzir um BEEP cada vez mais agudo:

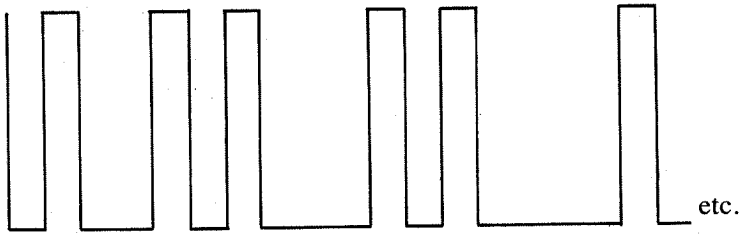
```
7000  21  00  10   LD     HL,1000
7003  11  20  00   LD     DE,0020
7006  ED  52       SBC    HL,DE
7008  11  01  00   LD     DE,0001
700B  E5           PUSH   HL
700C  CD  B5  03   CALL   03B5
700F  E1           POP    HL
7010  7C           LD     A,H
7011  A7           AND    A
7012  20  EF       JR     NZ,7003
7014  C9           RET
```

O programa começa por carregar 1000 (hexadecimal) em HL; é este o valor inicial do BEEP. Em seguida carrega 0200 (hexadecimal) no par de registos DE, indicando a quantidade que será diminuída de HL de cada vez. Em seguida subtrai-se DE a HL de modo a reduzir o conteúdo deste. Carrega-se depois em DE um valor que determina o tempo durante o qual o computador faz soar cada tom, antes de passar para outro mais agudo. Antes de chamarmos a rotina em ROM, guardamos o conteúdo de HL no stack. Depois de a ROM ter feito o seu trabalho trazemos este valor de volta a HL, verificando em seguida se o registo H é igual a zero. Se for, o programa termina; se não, passa ao tom seguinte. O conjunto destes tons produz um som cada vez mais agudo que pode ser usado em disparos laser. A rotina é bastante versátil; experimente usar outros valores tanto para o "passo" (valor que é diminuído a HL) como para o valor inicial de HL ou para a duração de cada tom. Através da experiência conseguirá obter

uma grande variedade de sons; se os experimentar recorrendo a um ciclo Basic verificará que podem ser bastante eficazes.

Que poderemos dizer, por exemplo, sobre os motores de automóveis? Bem, o som por eles produzido tem um nome especial: "ruído"... Talvez não seja uma designação muito técnica, mas indica o som produzido por um gravador de cassete, que é bastante agudo, ou o som grave dos potentes motores de qualquer nave espacial. As pessoas que já utilizaram sintetizadores conhecem estes diversos tipos de som, mas aqui manteremos a simplicidade das coisas interessando-nos apenas pelo bom e velho "ruído".

O ruído é de facto formado por impulsos espaçados aleatoriamente, e portanto é muito fácil simulá-lo. Necessitamos antes de mais de uma fonte de dados aleatórios, e em vez de escrever um programa complicado de geração de números aleatórios usaremos todos os códigos da ROM como dados. Isto significa que nos repetiremos um pouco quando terminarem os dados, mas como o ruído não é das coisas mais agradáveis é improvável que o leitor note a repetição. O motor de um navio produz um ruído grave, pelo que teremos de espaçar os impulsos usando um ciclo B. Utilizaremos HL para indicar o byte em ROM, e DE para contar o número de ciclos necessários. Vejamos qual será a aparência gráfica do nosso ruído:



E passemos à listagem do programa:

```
7000  F3          DI
7001  21  00  00  LD      HL,0000
7004  11  00  20  LD      DE,2000
7007  7E          LD      A,(HL)
7008  D3  FE      OUT     (FE),A
700A  06  FF      LD      B,FF
700C  10  FE      DJNZ    700C
700E  1B          DEC     DE
700F  23          INC     HL
7010  7B          LD      A,E
7011  B2          OR      D
7012  20  F3      JR      NZ,7007
7014  FB          EI
7015  C9          RET
```

Experimente o programa usando o carregador Basic habitual. Notará que o programa produz ainda cores bastante diferentes na margem do visor, o que pode ser bastante interessante. Como poderemos eliminar este efeito? Sugestão: use AND.

E como poderemos obter explosões? Bem, a técnica é bastante semelhante. Usaremos o mesmo ruído, mas fá-lo-emos evoluir de uma tonalidade aguda para uma tonalidade grave. Para o conseguirmos, tornamos cada vez maior a pausa entre cada ruído. Vejamos um programa para este efeito:

```
7000  21  00  00  LD      HL,0000
7003  0E  00      LD      C,00
7005  16  20      LD      D,20
7007  7E          LD      A,(HL)
7006  E6  18      AND     18
700A  D3  FE      OUT     (FE),A
700C  41          LD      B,C
700D  10  FE      DJNZ    700D
700F  23          INC     HL
7010  15          DEC     D
7011  20  F4      JR      NZ,7007
7013  0C          INC     C
7014  20  EF      JR      NZ,7005
7016  C9          RET
```

Neste programa utilizamos o registo C como um contador que torna o ciclo B cada vez mais demorado. Podemos tornar o efeito mais lento aumentando o valor inicial de D. Se pelo

contrário se pretende obter uma explosão mais rápida, reduz-se o valor incial do registo D. Usando um efeito de explosão muito rápido, chamado por um ciclo Basic aleatoriamente, podemos obter o efeito de trovoada e relâmpagos. Se se substituir INC C por DEC C (Código 0D), obtem-se um som progressivamente mais agudo, muito eficaz para uso num jogo quando algo surge no visor. Neste programa encontra-se ainda a solução para a minha última pergunta, como libertar-se de uma margem multi-colorida; utilizamos a instrução AND 18 para garantir que só os bits três e quatro são alterados.

Chegamos assim ao final do Capítulo VI, e espero que o leitor tenha obtido algumas ideias que possa utilizar a obtenção de efeitos sonoros, além de compreender o modo como os computadores comunicam com o mundo exterior. No Apêndice D apresenta-se uma lista completa das diversas instruções IN e OUT, juntamente com as respectivas definições. Se quiser experimentar o uso de ports extra, poderá adquirir os que existem no comércio. Alguns possuem linhas de entrada e saída separadas, enquanto outros utilizam circuitos integrados mais complexos e são programáveis. Em geral, para uso doméstico, quanto maior for o número de linhas de entrada/saída melhor. Por outro lado, procure aprender a usar bem as instruções IN e OUT em Basic, pois se souber fazê-lo não terá dificuldade em utilizá-las em código-máquina.

VII

POSSO INTERROMPER?

Neste capítulo começaremos por estudar as interrupções, e depois dedicaremos alguma atenção à ROM. As interrupções não têm grande utilidade directa no Spectrum, mas ajudam-nos a compreender um pouco o funcionamento da máquina.

A ROM do computador necessita de executar certas tarefas a intervalos regulares. Estas tarefas variam de computador para computador, mas uma delas consiste sempre em actualizar um qualquer tipo de "relógio" ou contador. Este contador permite uma temporização rigorosa. O modo como o computador realiza estas tarefas consiste na execução de interrupções.

Uma interrupção é um sinal enviado à CPU por uma entidade externa a ela, por exemplo o circuito lógico. Este sinal indica à CPU que deve parar o que está a fazer e dedicar-se a outra tarefa. Quando acaba a nova execução, ou mais exactamente quando deixa de responder à interrupção, continua o que estava a fazer anteriormente.

Existem dois tipos de interrupção. Um deles, a interrupção "maskable", é aquela que a CPU pode ignorar quando necessário; o outro, a interrupção "non-maskable", não se encontra nessas condições. No caso do ZX Spectrum é produzida uma interrupção "maskable" em cada 1/50 de segundo; o circuito lógico envia então um impulso para um perne chamado INT da CPU Z80 A.

Quando este sinal é recebido, é chamada a subrotina existente em ROM a partir do endereço 0038. Vejamos o programa:

Rotina de interrupção

```
003A  2A  78  5C   LD    HL,(5C78)
003D  23          INC   HL
003E  22  78  5C   LD    (5C78),HL
0041  7C          LD    A,H
0042  B5          OR    L
0043  20  03      JR      NZ,0048
0045  FD  34  40   INC   (IY+40)
0048  C5          PUSH  BC
0049  D5          PUSH  DE
004A  CD  BF  02   CALL  02BF
004D  D1          POP   DE
004E  C1          POP   BC
004F  E1          POP   HL
0050  F1          POP   AF
0051  FB          EI
0052  C9          RET
0053  E1          POP   HL
0054  6E          LD    L,(HL)
```

Primeiramente passa-se o conteúdo dos registos AF e HL para o stack. É necessário fazê-lo a fim de não corromper os valores neles contidos, dado que se interrompeu uma rotina e quando a interrupção tiver terminado a máquina voltará a ela como se nada tivesse acontecido. O passo seguinte consiste em carregar em HL os dois bytes menos significativos da variável de sistema FRAMES. Este valor em dois bytes é então incrementado, realizando-se um teste para verificar se chegou a zero. Este teste utiliza a instrução "OR", já descrita no Capítulo IV. Carregando H em A e executando a instrução OR sobre L, se HL for igual a zero o mesmo acontecerá ao resultado no Acumulador. Se HL não for zero, é anulada a instrução que manda incrementar o byte mais significativo de FRAMES (INC IY+40). Usa-se INC (IY+40) porque no registo IV encontra-se sempre o endereço base 5C3A. Este endereço é o de início da área de variáveis de sistema, o que permite à ROM consultar estas utilizando o valor em IY e um deslocamento em vez de recorrer ao par de registos HL. Por exemplo, se quisermos utilizar HL em vez de IY será necessário utilizar as seguintes instruções:


```
PUSH HL
LD HL,5C3A
INC (HL)
POP HL
```

Em vez disto, basta neste caso:

```
INC (IY+40)
```

Em seguida guarda-se o conteúdo dos outros registos no stack, antes de chamar a subrotina que executa uma leitura do teclado na variável de sistema K-SCAN, determinando em seguida a última tecla premida, guardada em LAST K. Em seguida devolvem-se aos registos os seus valores iniciais, através de instruções POP.

Finalmente, existe uma instrução EI antes do retorno.

A instrução EI é "irmã" da DI. Como ainda não encontrámos estas instruções, vamos agora referir-nos a elas.

A função destas instruções é bastante simples. Já mencionei o facto de existirem dois tipos de interrupção: "mascarável" e "não-mascarável" (que pode ser ou não desprezada). O Spectrum utiliza uma intervenção "maskable" para actualização do seu relógio interno e para leitura do teclado. Quando é executada uma interrupção deste tipo, a CPU interrompe automaticamente a aceitação de interrupções, ou seja, ignora qualquer nova interrupção. Evita assim que a interrupção seja interrompida... Como se pendurasse à porta um letreiro "Não incomode!". Quando termina a execução da interrupção "tira o letreiro" usando a instrução EI. EI significa portanto "Enable Interrupt" (Permitir Interrupção), e nada mais do que isso. É utilizada porque de outro modo as interrupções seriam definitivamente ignoradas. E sem interrupções o Spectrum nunca verificará sequer se o leitor carregou em alguma tecla...

Existem outros momentos em que a ROM não desejará ser incomodada. Quando carrega um programa de cassette, por exemplo. Se o circuito lógico "interrompe" a CPU no momento em que está a carregar um byte, por exemplo, quando a interrupção tiver terminado o gravador pode estar já a enviar para a máquina o byte seguinte.

Para evitar estas perturbações podemos portanto utilizar a

instrução DI, que significa "Disable Interrups" (impedir interrupções). Como já dissemos, esta instrução serve muito simplesmente para dizer à CPU que deve ignorar as interrupções "maskable" até "aviso em contrário", por exemplo através de uma instrução EI. A propósito, sempre que a CPU é ligada aceita interrupções "maskable".

Modo de interrupção

Mas ainda não foi dito que a CPU não passa necessariamente para as subrotinas em 0038 hexadecimal quando é interrompida. Passa a essas subrotinas apenas porque se encontra no "modo de interrupção 1". Que são então os "modos de interrupção"? Antes do mais convém dizer que para o utilizador do Spectrum estes modos de interrupção são em grande parte irrelevantes porque os outros só têm interesse para o utilizador que goste de ligar à máquina hardware próprio. A razão porque menciono os modos de interrupção é que um deles (a saber, o modo 2) utiliza o registo I, que provavelmente é desconhecido do leitor.

Neste ponto sou forçado a entrar um pouco no desagradável mundo do hardware a fim de explicar os modos de interrupção, pelo que peço ao leitor alguma paciência. Vou tentar ser o mais simples possível.

O modo de interrupção zero

As interrupções são muitas vezes utilizadas como uma espécie de "Olá", de cumprimento entre computadores falando entre si. Obviamente as máquinas não falam, e preferem em vez disso enviar mensagens serialmente (do mesmo modo que se guarda um programa numa cassette) ou em paralelo, ou seja, enviando um byte inteiro (oito bits) de cada vez. Antes de um computador "falar" com outro, necessita de chamar a atenção do segundo; isto é feito executando interrupções (note porém que os computadores inimigos, como o BBC e o Spectrum, se ignoram aristocraticamente usando DI!). Mas que acontecerá se existir ainda uma outra interrupção, executada 50 vezes por segundo, para actualizar um contador, por exemplo? É necessário neste caso que o computador possa dizer ao seu "camarada": "Olá, sou o outro computador, podemos falar um pouco?", a fim de que a CPU do

segundo não o confunda com o relógio que tem no pulso... Um modo de conseguir isto consiste em recorrer ao modo de interrupção zero.

O que acontece quando a máquina trabalha neste modo é que quando a interrupção é feita o dispositivo que interrompe tem tempo para "dizer" um byte inteiro à CPU. Quando a CPU recebe este byte executa-o muito simplesmente, como faz com qualquer outra instrução. Tendo em conta que as instruções RST têm um byte de comprimento, o computador pode "dizer" RST 28, por exemplo. Em 0028 poderia encontrar-se uma rotina capaz de receber a mensagem do outro computador. Por outro lado, se o circuito lógico interrompe nesse momento, indicando que é chegado o momento de acertar o relógio, pode dizer RST 38, levando a CPU a passar à execução da rotina em 0038 hexadecimal, que actualiza o contador. Recorrendo a estas directivas RST em modo zero um "dispositivo" pode portanto indicar à CPU a rotina que deve executar.

O modo de interrupção um

É este o modo de interrupção em que o computador se encontra quando é ligado. Se for produzida uma interrupção, é chamada a subrotina em 0038 hexadecimal. O Spectrum utiliza este modo de interrupção do modo já indicado.

O modo de interrupção dois

É aqui que se utiliza o registo I. O nome completo deste registo é "IV", ou "vector de interrupção". "Vector" é um termo técnico que designa um indicador de uma tabela. Para cada dispositivo capaz de produzir interrupções existe um endereço de subrotina apropriado numa tabela guardada em memória.

Tabela de interrupções

8000	10	00
8002	00	70
...	...	...

Para determinar qual a subrotina a chamar, é formado um

indicador (''pointer'') quando a CPU é interrompida. Este indicador apontará para dois bytes de dados existentes na tabela em causa. O indicador é formado considerando o registo I como o byte mais significativo e o byte para onde o dispositivo ''envia'' a CPU como byte menos significativo. Se utilizarmos a tabela acima, por exemplo, se o registo I contém 80 e o dispositivo envia um zero, será chamada a rotina 0010. Consideremos esta questão por fases a fim de a tornar mais fácil de compreender:

1. É produzido um pedido de interrupção.
2. A CPU recebe um byte do dispositivo que originou o pedido de interrupção.
3. Usando o registo I como byte mais significativo e o byte enviado como byte menos significativo, a CPU constrói um indicador ou vector.
4. No endereço indicado pelo vector encontraremos o byte menos significativo e o mais significativo.
5. Estes dois bytes são então usados para chamamento da subrotina desejada.

Para alterar o modo de interrupção em que o computador se encontra usamos as instruções IM0 e IM2. Quando a máquina é ligada a ROM cumpre uma rotina que inicializa todo o sistema, executando entre outras uma instrução IM1. Vejamos os códigos de operações das instruções relativas a modos de interrupção:

IM0	ED 46
IM1	ED 56
IM2	ED 5E

Interrupções ''non-maskable''

Como já mencionei anteriormente, existem dois tipos de interrupções. A NMI (interrupção ''não-mascarável'') será estudada agora. Dado não poder ser ignorada (é esse o sentido da expressão ''non-maskable''), esta interrupção só é de facto ignorada quando está a ser executada outra interrupção. Quando é produzida uma NMI a CPU passa a executar a subrotina que se inicia em 0066 hexadecimal, independentemente do modo em que se encontra.

Infelizmente a rotina NMI em 0066 parece ter um erro... Vejamos a sua listagem.

```
>Z0066
0066  F5          PUSH   AF
0067  E5          PUSH   HL
0068  2A  B0  5C  LD     HL,(5CB0)
006B  7C          LD     A,H
006C  B5          OR     L
006D  20  01      JR       NZ,0070
006F  E9          JP       (HL)
0070  E1          POP    HL
0071  F1          POP    AF
0072  ED  45      RETN
```

O primeiro aspecto a notar é que a rotina em causa termina pela instrução RETN. Esta instrução é usada em vez de RET para terminar a execução da rotina ''non-maskable''. Mas o aspecto mais importante é o erro aparente. Se estudarmos o programa, notamos que inicialmente é passado para o stack o conteúdo dos pares de registos AF e HL. Em seguida é carregado em HL o conteúdo dos endereços 5CB0 e 5CB1. Se o leitor estiver disposto a investigar estes endereços, constatará que se trata dos dois bytes livres na área das variáveis de sistema. A questão adensa-se... Em seguida é executada uma instrução ''OR 0'' sobre HL, usando o método já descrito no Capítulo IV. A instrução seguinte deveria ser JR Z,0070, mas em vez disso é JR NZ,0070. A questão é que se tratasse de uma JR Z, se esses bytes não fossem zero seria realizado um salto para o endereço indicado em HL. Deste modo seria possível relacionar uma tecla com a NMI de tal modo que se esses bytes contivessem o endereço de uma rotina apropriada seria possível por exemplo escapar a um ''crash''.

Mas tal como se encontra, esta rotina só produz o salto no caso de os bytes serem zero, o que a transforma num modo praticamente inútil de repor o sistema operativo nas suas condições iniciais, como se encontra ao ser ligado. Se o ''erro'' não existisse, estaria aberta a porta para a obtenção de teclas de funções programáveis e muitas outras aplicações. Enfim, esperemos ter melhor sorte da próxima vez...

APÊNDICES

A: Tabela de códigos ASCII em decimal e hexadecimal e mnemónicas Z80.
B: Mnemónicas Z80 e flags.
C: Variáveis de sistema do ZX Spectrum e sua explicação.
D: Definição breve das mnemónicas Z80.

APÊNDICE A

TABELA DE CÓDIGOS ASCII EM DECIMAL E HEXADECIMAL. MNEMÓNICAS Z80

Este apêndice inclui todos os números entre 0 e 255 ou 00 e FF em hexadecimal, os caracteres ASCII correspondentes e as mnemónicas Z80 que lhes estão associadas. Essas últimas encontram-se listadas em três colunas; na primeira, encontram-se as mnemónicas que não utilizam prefixo; na segunda, as que possuem o prefixo CB; finalmente, as que utilizam o prefixo ED. As instruções que envolvem os registos de indexação IX e IY possuem códigos hexadecimais correspondentes às HL equivalentes prefixadas por DD no caso de IX e FD no caso de IY.

HEX	DECIMAL	CARACTER	Z80 (s/ prefixo)	PREFIXO CB	PREFIXO ED
00	0	Não usados	NOP	RLC B	
01	1		LD BC,nn	RLC C	
02	2		LD (BC),A	RLC D	
03	3		INC BC	RLC E	
04	4		INC B	RLC H	
05	5		DEC B	RLC L	
06	6	Virg. impressão	LD B,n	RLC (HL)	
07	7	Mont. de linha	RLCA	RLC A	
08	8	Cursor esq.	EX AF,AF'	RRC B	
09	9	Cursor direita	ADD HL,BC	RRC C	
0A	10	Cursor p/ baixo	LD A,(BC)	RRC D	
0B	11	Cursor p/ cima	DEC BC	RRC E	
0C	12	Apagar caract.	INC C	RRC H	
0D	13	Entrar linha	DEC C	RRC L	

HEX	DECIMAL	CARACTER	Z80 (s/ prefixo)	PREFIXO CB	PREFIXO ED
0E	14	Número	LD C,n	RRC (HL)	
0F	15	Não usado	RRCA	RRC A	
10	16	Comando INK	DJNZ dis	RL B	
11	17	Com. PAPER	LD DE,nn	RL C	
12	18	Com. FLAH	LD(DE),A	RL D	
13	19	Com. BRIGHT	INC DE	RL E	
14	20	Com. INVERSE	INC D	RL H	
15	21	Com. OVER	DEC D	RL L	
16	22	Com. AT	LD D,n	RL (HL)	
17	23	Com. TAB	RLA	RL A	
18	24		JR Dis	RR B	
19	25		ADD HL,DE	RR C	
1A	26		LD A,(DE)	RR D	
1B	27	Não usados	DEC DE	RR E	
1C	28		INC E	RR H	
1D	29		DEC E	RR L	
1E	30		LD E,n	RR (HL)	
1F	31		RRA	RR A	
20	32	espaço	JR NZ,dis	SLA B	
21	33	!	LD HL,nn	SLA C	
22	34	"	LD(nn),HL	SLA D	
23	35		INC HL	SLA E	
24	36	$	INC H	SLA H	
25	37	%	DEC H	SLA L	
26	38	&	LD H,n	SLA (HL)	
27	39	'	DAA	SLA A	
28	40	(	JR Z,dis	SRA B	
29	41	)	ADD HL,HL	SRA C	
2A	42	*	LD HL,(nn)	SRA D	
2B	43	+	DEC HL	SRA E	
2C	44	,	INC L	SRA H	
2D	45	−	DEC L	SRA L	
2E	46	.	LD L,n	SRA (HL)	
2F	47	/	CPL	SRA A	
30	48	0	JR NC,dis		
31	49	1	LD SP,nn		
32	50	2	LD (nn),A		
33	51	3	INC SP		
34	52	4	INC (HL)		
35	53	5	DEC (HL)		
36	54	6	LD (HL),n		
37	55	7	SCF		
38	56	8	JR C,dis	SRL B	
39	57	9	ADD HL,SP	SRL C	
3A	58	:	LD A,(nn)	SRL D	
3B	59	;	DEC SP	SRL E	
3C	60	<	INC A	SRL H	
3D	61	=	DEC A	SRL L	

HEX	DECIMAL	CARACTER	Z80 (s/ prefixo)	PREFIXO CB	PREFIXO ED
3E	62	>	LD A,n	SRL (HL)	
3F	63	?	CCF	SRL A	
40	64	@	LD B,B	BIT 0,B	IN B,(C)
41	65	A	LD B,C	BIT 0,C	OUT (C),B
42	66	B	LD B,D	BIT 0,D	SBC HL,BC
43	67	C	LD B,E	BIT 0,E	LD (nn),BC
44	68	D	LD B,H	BIT 0,H	NEG
45	69	E	LD, B,L	BIT 0,1	RETN
46	70	F	LD B,(HL)	BIT 0,(HL)	IM 0
47	71	G	LD B,A	BIT 0,A	LD I,A
48	72	H	LD C,B	BIT 1,B	IN C,(C)
49	73	I	LD C,C	BIT 1,C	OUT (C),C
4A	74	J	LD C,D	BIT 1,D	ADC HL,BC
4B	75	K	LD C,E	BIT 1,E	LD BC,(nn)
4C	76	L	LD C,H	BIT 1,H	
4D	77	M	LD C,L	BIT 1,L	RETI
4E	78	N	LD C,(HL)	BIT 1,(HL)	
4F	79	O	LD C,A	BIT 1,A	LD R,A
50	80	P	LD D,B	BIT 2,B	IN D,(C)
51	81	Q	LD D,C	BIT 2,C	OUT (C),D
52	82	R	LD D,D	BIT 2,D	SBC HL,DE
53	83	S	LD D,E	BIT 2,E	LD (nn),DE
54	84	T	LD D,H	BIT 2,H	
55	85	U	LD D,L	BIT 2,L	
56	86	V	LD D,(HL)	BIT 2,(HL)	IM 1
57	87	W	LD D,A	BIT 2,A	LD A,I
58	88	X	LD E,B	BIT 3,B	IN E,(C)
59	89	Y	LD E,C	BIT 3,C	OUT (C),E
5A	90	Z	LD E,D	BIT 3,D	ADC HL,DE
5B	91	[	LD E,E	BIT 3,E	LD DE,(nn)
5C	92	/	LD E,H	BIT 3,H	
5D	93	]	LD E,L	BIT 3,L	
5E	94	↑	LD E,(HL)	BIT 3,(HL)	IM 2
5F	95	−	LD E,A	BIT 3,A	LD A,R
60	96	£	LD H,B	BIT 4,B	IN H,(C)
61	97	a	LD H,C	BIT 4,C	OUT (C),H
62	98	b	LD H,D	BIT 4,D	SBC HL,HL
63	99	c	LD H,E	BIT 4,E	LD (nn),HL
64	100	d	LD H,H	BIT 4,H	
65	101	e	LD H,L	BIT 4,L	
66	102	f	LD H,(HL)	BIT 4,(HL)	
67	103	g	LD H,A	BIT 4,A	RRD
68	104	h	LD L,B	BIT 5,B	IN L,(C)
69	105	i	LD L,C	BIT 5,C	OUT (C),L
6A	106	j	LD L,D	BIT 5,D	ADC HL,HL
6B	107	k	LD L,E	BIT 5,E	LD HL,(nn)
6C	108	l	LD L,H	BIT 5,H	
6D	109	m	LDL,L	BIT 5,L	

HEX	DECIMAL	CARACTER	Z80 (s/ prefixo)	PREFIXO CB	PREFIXO ED
6E	110	n	LD L,(HL)	BIT 5,(HL)	
6F	111	o	LD L,A	BIT 5,A	RLD
70	112	p	LD (HL),B	BIT 6,B	IN F,(C)
71	113	q	LD (HL),C	BIT 6,D	
72	114	r	LD (HL),D	BIT 6,D	SBC HL,SP
73	115	s	LD (HL),E	BIT 6,E	LD (nn),SP
74	116	t	LD (HL),H	BIT 6,H	
75	117	u	LD (HL),L	BIT 6,L	
76	118	v	HALT	BIT 6,(HL)	
77	119	w	LD (HL),A	BIT 6,A	
78	120	x	LD A,B	BIT 7,B	IN A,(C)
79	121	y	LD A,C	BIT 7,C	OUT (C),A
7A	122	z	LD A,D	BIT 7,D	ADC HL,SP
7B	123	{	LD A,E	BIT 7,E	LD SP,(nn)
7C	124	|	LD A,H	BIT 7,H	
7D	125	}	LD A,L	BIT 7,L	
7E	126	-	LF A,(HL)	BIT 7,(HL)	
7F	127	©	LD A,A	BIT 7,A	
80	128	■	ADD A,B	RES 0,B	
81	129	■	ADD A,C	RES 0,C	
82	130	■	ADD A,D	RES 0,D	
83	131	■	ADD A,E	RES 0,E	
84	132	■	ADD A,H	RES 0,H	
85	133	■	ADD A,L	RES 0,L	
86	134	■	ADD A,(HL)	RES 0,(HL)	
87	135	■	ADD A,A	RES 0,A	
88	136	■	ADC A,B	RES 1,B	
89	137	■	ADC A,C	RES 1,C	
8A	138	■	ADC A,D	RES 1,D	
8B	139	■	ADC A,E	RES 1,E	
8C	140	■	ADC A,H	RES 1,H	
8D	141	■	ADC A,L	RES 1,L	
8E	142	■	ADC A,(HL)	RES 1,(HL)	
8F	143	■	ADC A,A	RES 1,A	
90	144	(a)	SUB B	RES 2,B	
91	145	(b)	SUB C	RES 2,C	
92	146	(c)	SUB D	RES 2,D	
93	147	(d)	SUB E	RES 2,E	
94	148	(e)	SUB H	RES 2,H	
95	149	(f)	SUB L	RES 2,L	
96	150	(g)	SUB (HL)	RES 2,(HL)	
97	151	(h)	SUB A	RES 2,A	
98	152	(i)	SBC A,B	RES 3,B	
99	153	(j)	SBC A,C	RES 3,C	
9A	154	(k)	SBC A,D	RES 3,D	
9B	155	(l)	SBC A,E	RES 3,E	
9C	156	(m)	SBC A,H		
9D	157	(n)	SBC A,L	RES 3,L	

Gráficos do utilizador

HEX	DECIMAL	CARACTER	Z80 (s/ prefixo)	PREFIXO CB	PREFIXO ED
9E	158	(o)	SBC A,(HL)	RES 3,(HL)	
9F	159	(p)	SBC A,A	RES 3,A	
A0	160	(q)	AND B	RES 4,B	LDI
A1	161	(r)	AND C	RES 4,C	CPI
A2	162	(s)	AND D	RES 4,D	INI
A3	163	(t)	AND E	RES 4,E	OUTI
A4	164	(u)	AND H	RES 4,H	
A5	165	RND	AND L	RES 4,L	
A6	166	INKEY$	AND (HL)	RES 4,(HL)	
A7	167	PI	AND A	RES 4,A	
A8	168	FN	XOR B	RES 5,B	LDD
A9	169	POINT	XOR C	RES 5,C	CPD
AA	170	SCREEN$	XOR D	RES 5,D	IND
AB	171	ATTR	XOR E	RES 5,E	OUTD
AC	172	AT	XOR H	RES 5,H	
AD	173	TAB	XOR L	RES 5,L	
AE	174	VAL$	XOR (HL)	RES 5,(HL)	
AF	175	CODE	XOR A	RES 5,A	
B0	176	VAL	OR B	RES 6,B	LDIR
B1	177	LEN	OR C	RES 6,C	CPIR
B2	178	SIN	OR D	RES 6,D	INIR
B3	179	COS	OR E	RES 6,E	OTIR
B4	180	TAN	OR H	RES 6,H	
B5	181	ASN	OR L	RES 6,L	
B6	182	ACS	OR (HL)	RES 6,(HL)	
B7	183	ATN	OR A	RES 6,A	
B8	184	LN	CP B	RES 7,B	LDDR
B9	185	EXP	CP C	RES 7,C	CPDR
BA	186	INT	CP D	RES 7,D	INDR
BB	187	SQR	CP E	RES 7,E	OTDR
BC	188	SGN	CP H	RES 7,H	
BD	189	ABS	CP L	RES 7,L	
BE	190	PEEK	CP (HL)	RES 7,(HL)	
BF	191	IN	CP A	RES 7,A	
C0	192	USR	RET NZ	SET 0,B	
C1	193	STR$	POP BC	SET 0,C	
C2	194	CHR$	JP NZ,nn	SET 0,D	
C3	195	NOT	JP nn	SET 0,E	
C4	196	BIN	CALL NZ,nn	SET 0,H	
C5	197	OR	PUSH BC	SET 0,L	
C6	198	AND	ADD A,n	SET 0,(HL)	
C7	199	> =	RST 0	SET 0,A	
C8	200	< =	RET Z	SET 1,B	
C9	201	<>	RET	SET 1,C	
CA	202	LINE	JP Z,nn	SET 1,D	
CB	203	THEN		SET 1,E	
CC	204	TO	CALL Z,nn	SET 1,H	
CD	205	STEP	CALL nn	SET 1,L	

Gráficos do utilizador

HEX	DECIMAL	CARACTER	Z80 (s/ prefixo)	PREFIXO CB	PREFIXO ED
CE	206	DEF FN	ADC A,n	SET 1,(HL)	
CF	207	CAT	RST 8	SET 1,A	
D0	208	FORMAT	RET NC	SET 2,B	
D1	209	MOVE	POP DE	SET 2,C	
D2	210	ERASE	JP NC,nn	SET 2,D	
D3	211	OPEN	OUT (n),A	SET 2,E	
D4	212	CLOSE	CALL NC,nn	SET 2,H	
D5	213	MERGE	PUSH DE	SET 2,L	
D6	214	VERIFY	SUB n	SET 2,(HL)	
D7	215	BEEP	RST 10	SET 2,A	
D8	216	CIRCLE	RET C	SET 4,B	
D9	217	INK	EXX	SET 3,C	
DA	218	PAPER	JP C,nn	SET 3,D	
DB	219	FLASH	IN A,(n)	SET 3,R	
DC	220	BRIGHT	CALL C,nn	SET 3,H	
DD	221	INVERSE	PREFIXA INSTRUÇÕES USANDO IX	SET 3,L	
DE	222	OVER	SBC A,n	SET 3,(HL)	
DF	223	OUT	RST 18	SET 3,A	
E0	224	LPRINT	RET PO	SET 4,B	
E1	225	LLIST	POP HL	SET 4,C	
E2	226	STOP	JP PO,nn	SET 4,D	
E3	227	READ	EX (SP),HL	SET 4,E	
E4	228	DATA	CALL PO,nn	SET 4,H	
E5	229	RESTORE	PUSH HL	SET 4,L	
E6	230	NEW	AND n	SET 4,(HL)	
E7	231	BORDER	RST 20	SET 4,A	
E8	232	CONTINUE	RET PE	SET 5,B	
E9	233	DIM	JP (HL)	SET 5,C	
EA	234	REM	JP PE,nn	SET 5,D	
EB	235	FOR	EX DE, HL	SET 5,E	
EC	236	GO TO	CALL PE,nn	SET 5,H	
ED	237	GO SUB		SET 5,L	
EE	238	INPUT	XOR n	SET 5,(HL)	
EF	239	LOAD	RST 28	SET 5,A	
F0	240	LIST	RET P	SET 6,B	
F1	241	LET	POP AF	SET 6,C	
F2	242	PAUSE	JP P,nn	SET 6,D	
F3	243	NEXT	DI	SET 6,E	
F4	244	POKE	CALL P,nn	SET 6,H	
F5	245	PRINT	PUSH AF	SET 6,L	
F6	246	PLOT	OR n	SET 6,(HL)	
F7	247	RUN	RST 30	SET 6,A	
F8	248	SAVE	RET M	SET 7,B	
F9	249	RANDOMIZE	LD SP,HL	SET 7,C	
FA	250	IF	JP M,nn	SET 7,D	
FB	251	CLS	EI	SET 7,E	
FC	252	DRAW	CALL M,nn	SET 7,H	
FD	253	CLEAR	PREFIXA INSTRUÇÕES USANDO IY	SET 7,L	
FE	254	RETURN	CP n	SET 7,(HL)	
FF	255	COPY	RST 38	SET 7,A	

APÊNDICE B

FLAGS

Neste apêndice pode-se observar como as instruções afectam as flags. São listadas todas as instruções Z80, apresentando à frente o respectivo efeito sobre as flags. Só se indicam as flags importantes, a saber a flag Carry, a flag Paridade/Overflow, a flag Zero e a Flag Sinal, não tendo as outras qualquer utilidade directa para o programador uma vez que não desempenham qualquer papel na tomada de decisões (não são consideradas como condições nas instruções JR, JP, CALL e RET). Nesta tabela serão usados alguns símbolos:

Nas mnemónicas:	nn	Número de um só byte
	nnnn	Número de dois bytes
	r	Registo de um só byte (A, B, C, D, E, H, L)
	d	Registo de dois bytes
	c	Condição
	dis	Deslocamento calculado em complemento para dois
Nas flags:	0	A flag é passada a zero
	1	A flag é passada a 1
	.	A flag não é afectada
	R	A flag é alterada em função do resultado

?	A flag tem um valor aleatório
B	A flag é passada a 1 se o registo B ou o par de registos BC (dependendo da instrução) contiverem zero no final da operação

INSTRUÇÕES (mnemónicas)	FLAGS S	Z	P	C
ADC A,r	R	R	R	R
ADC HL,d	R	R	R	R
ADD A,r	R	R	R	R
ADD HL,d	.	.	.	R
ADD IX,d	.	.	.	R
ADD IY,d	.	.	.	R
AND r	R	R	R	0
BIT b,r	?	R	?	.
CALL nnnn	.	.	.	.
CALL c,nnnn	.	.	.	.
CCF	.	.	.	R
CP r	R	R	R	R
CPI	R	B	R	.
CPD	R	B	R	.
CPIR	R	B	R	.
CPDR	R	B	R	.
CPL	.	.	.	.
DAA	R	R	R	R
DEC r	R	R	R	.
DEC d	.	.	.	.
DI	.	.	.	.
DJNZ dis	.	B	.	.
EI	.	.	.	.
EX AF,AF'	.	.	.	.
EX DE,HL	.	.	.	.
EX (SP),HL	.	.	.	.
EX (SP),IX	.	.	.	.
EX (SP),IY	.	.	.	.

INSTRUÇÕES (mnemónicas)	FLAGS S	Z	P	C
EXX	.	.	.	.
HALT	.	.	.	.
IM 0	.	.	.	.
IM 1	.	.	.	.
IM 2	.	.	.	.
INC r	R	R	R	.
INC d	.	.	.	.
IN A,(nn)	.	.	.	.
IN r,(C)	R	R	R	.
INI	?	B	?	.
IND	?	B	?	.
INIR	?	1	?	.
INDR	?	1	?	.
JP nnnn	.	.	.	.
JP c,nnnn	.	.	.	.
JP (HL)	.	.	.	.
JP (IX)	.	.	.	.
JP (IY)	.	.	.	.
JR dis	.	.	.	.
JR c,dis	.	.	.	.
LD (d),A	.	.	.	.
LD A,(d)	.	.	.	.
LD A,R	R	R	R	.
LD A,I	R	R	R	.
LD I,A	.	.	.	.
LD R,A	.	.	.	.
LD SP,HL	.	.	.	.
LD SP,IX	.	.	.	.
LD SP,IY	.	.	.	.
LD r,r	.	.	.	.
LD r,nn	.	.	.	.
LD d,nnnn	.	.	.	.
LD A,(nnnn)	.	.	.	.
LD (nnnn),A	.	.	.	.
LD d,(nnnn)	.	.	.	.
LD (nnnn),d	.	.	.	.

INSTRUÇÕES (mnemónicas)	FLAGS S	Z	P	C
LDI	.	.	B	.
LDD	.	.	B	.
LDIR	.	.	0	.
LDDR	.	.	0	.
NEG	R	R	R	R
NOP	.	.	.	.
OR r	R	R	R	0
OUT (nn),A	.	.	.	.
OUT (C),r	.	.	.	.
OUTI	?	B	?	.
OUTD	?	B	?	.
OTIR	?	1	?	.
OTDR	?	1	?	.
POP AF	R	R	R	R
POP d	.	.	.	.
PUSH AF	.	.	.	.
PUSH d	.	.	.	.
RES b,r	.	.	.	.
RET	.	.	.	.
RET c	.	.	.	.
RETN	.	.	.	.
RETI	.	.	.	.
RLA	.	.	.	R
RLCA	.	.	.	R
RRA	.	.	.	R
RRCA	.	.	.	R
RL r	R	R	R	R
RLC r	R	R	R	R
RR r	R	R	R	R
RRC r	R	R	R	R
RRD	R	R	R	.
RST 00	.	.	.	.
RST 08	.	.	.	.
RST 10	.	.	.	.
RST 18	.	.	.	.
RST 20	.	.	.	.

INSTRUÇÕES (mnemónicas)	FLAGS S	Z	P	C
RST 28	.	.	.	.
RST 30	.	.	.	.
RST 38	.	.	.	.
SBC A,r	R	R	R	R
SBC HL,d	R	R	R	R
SCF	.	.	.	1
SET b,r	.	.	.	.
SLA r	R	R	R	R
SRA r	R	R	R	R
SRL r	R	R	R	R
SUB r	R	R	R	R
XOR r	R	R	R	0

Nota: Nos casos em que se indica "r" nas mnemónicas, esta letra representa não só os registos de um só byte como ainda (HL), (IX + dis), (IY + dis) e dados directos "nn" quando aplicável.

APÊNDICE C

VARIÁVEIS DE SISTEMA

Este apêndice apresenta uma lista de todas as variáveis de sistema e explica-as um pouco melhor do que o Manual Sinclair. Estas explicações serão melhor compreendidas por aqueles que conhecerem o mecanismo da ROM.

A indicação "X" antes do número de bytes de cada variável significa que esta só deverá ser utilizada se o leitor compreender os efeitos possíveis da alteração.

VARIÁVEIS DE SISTEMA

Bytes	*Endereço em hex.*	*Etiqueta*	*Função*
8	5C00	KSTATE	Esta variável consiste em oito byts, cada um deles contendo informações sobre a tecla premida, como seja o período de repetição, o seu código em modo "extended", etc.
1	5C08	LAST K	Esta variável contém sempre o código da última tecla premida, tendo em conta o modo. Só é alterada quando se carrega numa nova tecla. A repetição automática afecta a variável. Passando-se a zero pode-se verificar se foi premida alguma tecla.

Bytes	*Endereço em hex.*	*Etiqueta*	*Função*
1	5C09	REPDEL	Tempo (em 50 ciclos por segundo) durante o qual é necessário carregar numa tecla para repetir. Possui inicialmente o valor 23 hexadecimal.
1	5C0A	REPPER	Atraso (em 50 ciclos por segundo) entre as sucessivas repetições de uma tecla. Inicialmente contém o valor 05 hexadecimal. Pode-se diminuir para um mínimo de 01 hexadecimal a fim de tornar mais rápida a repetição.
2	5C0B	DEFADD	Endereço dos argumentos da função definida pelo utilizador no caso de estar a ser avaliada alguma.
1	5C0D	K DATA	Quando é escrito directamente no teclado um código de comando, por exemplo tecla shift em modo "extended" e tecla 1 (INK azul), o segundo byte, ou seja a cor concreta ou o uso ou não de FLASH, é guardado aqui enquanto o código INK, PAPER, FLASH ou INVERSE é impresso. Depois disto a ROM consulta este byte para impressão a seguir ao código de comando.
2	5C0E	TVDATA	Esta variável é usada pela rotina de impressão para guardar AT, TAB e os comandos de cor dirigidos para a televisão.
X38	5C10	STRMS	Esta variável é usada para guardar os deslocamentos em CHANS. Para um total de 19 ficheiros, 16 do utilizador e três da máquina, existe um deslocamento. Quando este é somado a CHANS, aponta para um endereço que constitui o início da rotina de tratamento desse ficheiro.
2	5CB6	CHARS	Indica o endereço do conjunto de caracteres menos 100 hexadecimal (incluindo no conjunto de caracteres todos os que possuem códigos entre 32 — espaço — e 127 — copyright — inclusive). A variável contém normalmente o endereço 4C00 (conjunto de caracteres em 4D00) mas este valor pode ser alterado levando-a a apontar para um novo conjunto de caracteres, definido pelo utilizador.
1	5C38	RASP	Som produzido pelo computador quando se introduzem códigos de comando de cor ilegais, ou quando a linha de programa em montagem ultrapassa um comprimento de 23 linhas. Para alterar este comprimento, modifique o conteúdo desta variável.
1	5C39	PIP	Nesta variável encontra-se definida a duração do som produzido pelas teclas ao serem premidas. Pode aumentar o seu valor a fim de tornar mais audível o som produzido.
1	5C3A	ERR NR	Indica o código das mensagens, menos um. Inicialmente em FF hexadecimal. Se for alterada por um programa Basic, quando este programa termina a máquina imprime a mensagem de erro desejada.

Bytes	Endereço em hex.	Etiqueta	Função
X1	5C3B	FLAGS	Este byte contém as flags que controlam o sistema Basic.
X1	5C3C	TVFLAG	Flags associadas à impressão no visor e na impressora.
X2	5C3D	ERR SP	Esta variável aponta para um elemento do stack da máquina. Quando ocorre um erro, é para este endereço que a execução salta depois de o stack ser limpo por RST 08. Alterando este elemento, é possível escrever novas rotinas de tratamento de erros.
2	5CBF	LIST SP	Esta variável aponta para o endereço de retorno do stack da máquina, para o qual salta a execução após uma listagem automática.
1	5C41	MODE	Esta variável especifica o cursor em uso — K, L, C, E ou G.
2	5C42	NEWPPC	Linha para onde deve saltar a execução. Usada nas instruções GOTO e GOSUB.
1	5C44	NSPPC	Número da declaração da linha para onde deve saltar a execução. Alterando primeiro a variável NEWPPC e depois NSPPC força-se um salto para uma determinada declaração de uma linha.
2	5C45	PPC	Número de linha onde se encontra a instrução actualmente em execução.

Bytes	Endereço em hex.	Etiqueta	Função
1	5C47	SUBPPC	Número da instrução actualmente em execução.
1	5C48	BORDCR	Esta variável contém os atributos da janela inferior do visor. Os bits zero a dois correspondem à cor INK, e os bits três a cinco à cor de PAPER/ /BORDER. Os bits FLASH e BRIGHT não são usados.
2	5C49	E PPC	Número de linha onde se encontra o cursor.
X2	5C4B	VARS	Esta variável indica o início da área onde estão guardadas as variáveis durante a execução de um programa Basic.
2	5C4D	DEST	Endereço da variável em atribuição.
X2	5C4F	CHANS	Indica a tabela de endereços usada por STRMS.
X2	5C51	CURCHL	Indica o endereço da tabela anterior (de endereços de tratamento de ficheiros) que está a ser usado pela rotina de tratamento.
X2	5C53	PROG	Esta variável indica o início da área onde se encontra o programa Basic.
X2	5C55	NXTLIN	Indica o endereço da linha do programa Basic que se segue à que está em execução.
X2	5C57	DATADD	Aponta para o separador final do último elemento DATA. Se não existir DATA no programa, aponta para 80

Bytes	*Endereço em hex.*	*Etiqueta*	*Função*
			hexadecimal, no fim das informações de canal.
X2	5C59	E LINE	Endereço da ordem que está a ser introduzida no teclado.
2	5C5B	K CUR	Endereço do cursor no interior da linha avaliada.
X2	5C5D	CHADD	Endereço do caracter a interpretar em seguida.
2	5C5F	X PTR	Endereço do caracter que se encontra depois do indicador "?".
X2	5C61	WORKSP	Endereço da área de trabalho temporária.
X2	5C63	STKBOT	Endereço inferior do stack de cálculo.
X2	5C65	STKEND	Endereço do início da memória livre.
1	5C67	BREG	Registo de cálculo usado para diversos fins (contagem).
2	5C68	MEM	Endereço da área usada para as seis memórias de cálculo (normalmente MEMBOT, mas nem sempre).
1	5C6A	FLAGS2	Mais flags.
X1	5C6B	DF SZ	Número de linhas (incluindo uma em branco) que constituem a janela do visor.
2	5C6C	STOP	Número de linha superior em listagem automática.

Bytes	*Endereço em hex.*	*Etiqueta*	*Função*
2	5C6E	OLDPPC	Número de linha para onde salta a ordem CONTINUE.
1	5C72	STRLEN	Comprimento do destino (tipo cadeia) em atribuição.
2	5C74	T ADDR	Endereço do elemento seguinte da tabela de sintaxe. Na ROM existe uma extensa tabela que define o local onde se encontra a rotina correspondente a cada ordem e o modo de obter a informação necessária.
2	5C76	SEED	Origem da função RND. É esta a variável sobre a qual actua a função RANDOMIZE.
3	5C78	FRAMES	Contador de imagens com três bytes, incrementado em cada ciclo da alimentação (50 ciclos por segundo). Consulte o capítulo 18 do Manual Sinclair.
2	5C7B	UDG	Endereço do primeiro caracter gráfico definido pelo utilizador. Recorde que quando se altera a RAMTOP para incluir um programa em código-máquina, a área de UDG's não é afectada. Se colocar código nesta área, corromperá certamente quaisquer caracteres gráficos que tenha definido.
1	5C7D	COORDS	Usada para armazenamento temporário da coordenada X quando são realizados cálculos para definição da posição de PLOT.

Bytes	*Endereço em hex.*	*Etiqueta*	*Função*
1	5C7E		Como acima, mas para a coordenada Y.
1	5C7F	P POSN	Número da posição de impressão (32 colunas).
1	5C80	PR CC	Byte menos significativo do endereço da posição que se segue para LPRINT AT (no buffer da impressora).
1	5C81		Não usada.
2	5C82	ECHO E	Número de coluna (32) e de linha (24) do final do buffer da impressora.
2	5C84	DFCC	Endereço da posição de PRINT da fiada superior do caracter no ficheiro de imagem. Pode ser dirigida para outra posição.
2	5C86	DFCCL	Como DFCC, para a parte inferior do visor.
X1	5C88	S POSN	Número de coluna (32) para a posição de PRINT.
X1	5C89		Número de coluna (24) para a posição de PRINT.
X2	5C8A	SPOSNL	Como SPOSN, para a janela inferior.
1	5C8C	SCR CT	Conta os "scrolls" da imagem; contém mais um do que o número de "scrolls" realizados antes de a imagem parar imprimindo a pergunta "scroll?". Se for regularmente colocado aqui o valor 255, a imagem rolará indefinidamente, sem parar nunca.
1	5C8D	ATTR P	Atributos permanentes (definidos por

Bytes	*Endereço em hex.*	*Etiqueta*	*Função*
			declarações INK, PAPER, etc., globais).
1	5C8E	MASK P	Usada para atributos transparentes. Qualquer bit igual a um indica que o atributo correspondente não é retirado de ATTR P mas do valor que se encontra já no visor.
1	5C8F	ATTR T	Atributos temporários em uso (por exemplo definidos em instruções PLOT, PRINT, DRAW).
1	5C90	MASK T	Como MASK P, mas temporário.
1	5C91	P FLAG	Mais flags.
30	5C92	MEMBOT	É nesta área que a unidade de cálculo aritmético pode guardar até seis diferentes números em notação de vírgula flutuante (cinco bytes cada), utilizando "memórias" especiais (ver variável MEM).
2	5CB0	INTERR	Ex-vector de interrupção, não usado devido às características da programação da ROM. Consultar o capítulo "Posso Interromper?". Pode ser usada sem restrições pelo programador.
2	5CB2	RAMTOP	Esta variável contém o endereço do último byte da área Basic.
2	5CB4	PRAMT	Endereço do último byte físico da RAM.

APÊNDICE D

MNEMÓNICAS Z80 E SUA EXPLICAÇÃO

Este apêndice fornece uma breve descrição de cada uma das ordens aceites pelo processador Z80. Refere-se ao tipo de funcionamento que provocam e à correpondente actuação sobre as flags. Não são indicados os respectivos códigos hexadecimais, mas é possível encontrá-los no apêndice A, onde se encontram listadas todas as mnemónicas Z80 juntamente com os respectivos códigos de operação e os códigos de operação e ASCII correspondentes usados pelo sistema operativo do Spectrum.

Nas explicações que se seguem serão usadas algumas abreviaturas:

r = Registo de um só byte

A, B, C, D, E, F, H, L = Registos simples do ZX Spectrum.

nnnn = Dado directo, de dois bytes.

dl = Registos de dois bytes: BC, DE, HL ou SP.

d2 = Registos de dois bytes: BC, DE, HL, IX, IY e SP.

x = Número de ordem de um bit, compreendido entre 0 e 7.

dis = deslocamento em número de bytes, segundo a convenção de ''complemento para dois''.

res = valor hexadecimal num só byte: 00, 08, 10, 18, 20, 28, 30 ou 38.

Apresentamos em seguida as instruções de código-máquina ordenadas alfabeticamente:

ADC A,r : Soma o registo r ao Acumulador, e soma igualmente a flag carry ao resultado. Exceptuando ADC A,A, o conteúdo do registo operando mantém-se. A flag N é passada a zero pela instrução ADC, e as restantes flags reflectem o estado final do Acumulador.

ADC A,(HL)
ADC A,(IX+dis)
ADC A,(IY+dis)
ADC A,nn : Idênticas a ADC A,r, mas somando ao Acumulador os dados indicados por (HL), (IX+dis) ou (IY+dis), ou dados directos.

ADC HL,dl : Realiza uma soma dupla, acrescentando primeiro o valor da flag "carry" ao bit menos significativo do registo L, e somando em seguida o par de registos dl (BC, DE, HL, SP) a HL. A flag N será passada a zero e as outras flags reflectirão o estado de HL.

ADD A,r : Realiza uma soma simples num único byte, adicionando r ao Acumulador. ADD passa a zero a flag N, e as outras flags reflectirão o estado do Acumulador.

ADD A,(HL)
ADD A,(IX+dis)
ADD A,(IY+dis)
ADD A,nn : Exactamente como em ADD A,r, excepto que neste caso se soma a A o conteúdo do endereço indicado por HL, IX+dis e IY+dis, ou um dado directo formado por dois bytes.

ADD HL,dl
ADD IX,dl
ADD IY,dl : Realiza uma soma a dois bytes sobre o conteúdo dos pares de registos HL, IX ou IY respectivamente. Soma-se a estes um par de registos dl (HL, BC, DE, SP), o qual se mantém inalterável no final da operação.

AND r : Realiza uma operação lógica AND sobre o conteúdo do Acumulador. Os bits do registo r são comparados com os correspondentes do Acumulador, e quando ambos são iguais a um o resultado é também 1, guardado no Acumulador; em todos os outros casos o bit em avaliação é passado ou mantido ao valor zero no Acumulador. O registo operando, r, não é afectado por estas operações. As flags são afectadas em função do resultado no Acumulador.

AND (HL)
AND (IX+dis)
AND (IY+dis)
AND nn : É executada uma operação lógica AND sobre o Acumulador. Os dados indicados por HL, IX+dis, IY+dis, ou os dados directos na forma nn são comparados com o conteúdo do Acumulador, ficando neste o resultado da operação lógica AND executada bit a bit. O operando não é afectado em caso algum. As flags são afectadas em função do resultado no Acumulador.

BIT x,r
BIT x,(HL)
BIT x,(IX+dis)
BIT x,(IY+dis) : Esta instrução verifica o bit x (onde x varia entre zero e sete) do registo r, (HL), (IX+dis) ou (IY+dis). Se o bit é zero, a flag Zero é passada a 1. Se o bit é um, a flag Zero é passada a zero. A flag C não é afectada, a flag H é passada a um e a flag N a zero. O estado das flags S e P/V não pode ser previsto.

CALL nnnn : Provoca um salto para uma subrotina que se inicia no endereço nnnn. O endereço da instrução seguinte é guardado no stack, mantendo-se aí até ser atingida uma instrução RET. Nenhuma das flags é afectada por esta instrução.

CALL C,nnnn
CALL M,nnnn
CALL NC,nnnn
CALL NZ,nnnn
CALL P,nnnn
CALL PE,nnnn
CALL PO,nnnn
CALL Z,nnnn

: Estas ordens actuam exactamente do mesmo modo que uma CALL incondicional, sendo no entanto ignoradas quando a condição não é satisfeita. As flags não são afectadas.

CCF

: O estado da flag Carry é invertido. A flag N é passada a zero, mas as outras flags não são alteradas.

CP r

: Esta instrução subtrai o registo r ao Acumulador, mas não altera o conteúdo de qualquer destes. As flags são porém modificadas em função do resultado.

CP (HL)
CP (IX+dis)
CP (IY+dis)
CP nn

: Estas instruções comportam-se exactamente como CP r, mas comparando agora ou dados indicados por HL, IX+dis e IY+dis, ou dados directos, com o conteúdo do registo A, em vez de a comparação ser feita com um registo simples. As flags reflectem o resultado da operação, que não altera o conteúdo dos registos envolvidos.

CPD

: O conteúdo da posição de memória indicada por HL é comparado com o conteúdo do Acumulador. A flag N é passada a 1, e as outras flags reflectem o resultado da operação excepto a flag C, que não é afectada. O registo BC, actuando como contador, é então decrementado tal como o par de registos HL, apontando assim para a posição seguinte (inferior) em memória. Se o par de registos BC passa a zero a flag P/V é passada igualmente para zero; no caso

contrário mantém-se a um.

CPDR

: Actua exactamente do mesmo modo que CPD, mas depois de HL ter sido decrementado juntamente com BC, se este não for zero repete a operação. A repetição não se verificará se BC for igual a zero ou se o valor do Acumulador for igual ao da posição de memória indicada por HL.

CPI

: Esta instrução actua exactamente como a CPD, incrementando no entanto o registo HL em vez de o decrementar.

CPIR

: Esta instrução actua exactamente como a CPDR, exceptuando o facto de o registo HL ser incrementado ao terminar a operação, em vez de decrementado.

CPL

: Complementa o conteúdo do Acumulador. Os bits iguais a um são passados para zero e os iguais a zero são passados a um. Todas as flags da CPU são mantidas no seu estado anterior, excepto as flags H e N, passadas a um.

DAA

: Esta instrução altera o valor do Acumulador, passando-o de um valor binário para dois algarismos decimais em codificação binária. Os quatro bits mais significativos reflectem o algarismo das "dezenas", e os quatro menos significativos reflectem o algarismo das "unidades".

EI

: Esta instrução ("enable interrupt") ordena à CPU que aceite interrupções "exclusivas". Depois de ser recebida uma interrupção, não é aceite qualquer

outra até ser recebida outra instrução EI.

EX AF,AF' : Esta ordem de um só byte troca os valores em AF e AF'. As flags reflectem o estado de A depois da troca.

EXX : Esta instrução substitui os pares de registos BC, DE e HL pelos registos correspondentes do conjunto alternativo.

EX DE,HL : Esta instrução troca entre si os valores dos pares de registos DE e HL.

EX (SP),HL : O último valor colocado no stack é trocado pelo conteúdo de HL. Tanto o valor de SP como as flags não são afectados por esta instrução.

EX (SP),IX
EX (SP),IY : Semelhantes a EX (SP),HL, estas duas instruções trocam pelo último valor do stack o conteúdo dos registos IX e IY. As flags não são alteradas.

HALT : Esta istrução leva a CPU a interromper todas as operações, mantendo-se neste estado até receber um sinal de interrupção ou "reset". A memória dinâmica é "refrescada" porque o processador processa de facto repetidamente a instrução NOP para esse efeito. Esta instrução não altera quaisquer registos ou flags.

IM 0 : Esta instrução ordena à CPU que se mantenha no modo de interrupção zero. Quando recebe uma interrupção, a CPU permite que um dispositivo externo convenientemente concebido e accionado force a passagem de uma instrução para o canal de dados. A CPU executará

depois essa instrução. As flags não são afectadas por qualquer ordem IM.

IM 1 : Esta instrução coloca a CPU no modo de interrupção 1. Quando recebe uma interrupção, a CPU executa uma RESTART para o endereço 38 hexadecimal. As flags não são afectadas.

IM 2 : Esta interrupção passa a CPU ao modo de interrupção 2. A CPU, quando "interrompida", saltará para um endereço formado, considerando o conteúdo do registo vector de interrupção (IV ou apenas I) como equivalente aos oito bits mais significativos e a informação colocada no canal de dados como equivalendo aos oito bits menos significativos.

IN r,(C) : Esta ordem leva a CPU a ler o valor no port C, colocando-o no registo r. As flags não são afectadas.

IN A,nn : Actua exactamente da mesma maneira que a instrução anterior, mas usa agora o dado directo nn e o registo A em vez do registo r.

DEC r : Esta instrução decrementa o registo r. Não afecta a flag C. A flag N é passada a zero. As outras flags reflectem o valor resultante no registo r.

DEC (HL)
DEC (IX+dis)
DEC (IY+dis) : Esta ordem decrementa o conteúdo da posição de memória indicada por HL, IX+dis ou IY+dis. As flags são afectadas do modo já indicado para DEC r.

DEC d2 : Esta instrução decrementa o par de regis-

tos d2 (BC, DE, HL, SP, IX e IX). Não afecta qualquer das flags.

DI : Este comando instroi a CPU para não aceitar uma interrupção mascarável.

DJNZ : Esta ordem decrementa o registo B. Se B não for igual a zero é realizado um salto relativo, usando o deslocamento e a complementação para dois; o deslocamento é somado ao Contador de Programas (registo PC).

INC r : Esta instrução incrementa o registo r. A flag "carry" não é afectada, a flag N é passada a zero e as outras reflectem o valor resultante do registo r.

INC HL
INC (IX+dis)
INC (IY+dis)

Esta ordem incrementa o conteúdo da posição indicada por HL, IX+dis e IY+dis. As flags são afectadas do modo descrito anteriormente para INC r.

INC d2 : Esta instrução incrementa o par de registos d2 (BC, DE, HL, SP, IX ou IY). Não afecta as flags.

IND : Esta ordem leva à aceitação dos dados provenientes do "port" de entrada especificado pelo registo C. Os dados são transferidos para a posição de memória indicada por HL; este registo é em seguida decrementado. O registo B, servindo como contador, é também decrementado e se passar a zero a flag Z assume o valor 1. O estado das flags S, M e P/V não pode ser previsto. A flag N é sempre passada a 1 por esta instrução, e a flag "carry" não é afectada.

INDR : Esta instrução actua exactamente como a anteriormente descrita, mas se B não for zero no final da operação esta é repetida. Isto significa que no final da execução as flags serão afectadas do modo indicado, mas a flag Z estará obrigatoriamente em 1.

INI : Esta instrução tem exactamente o mesmo efeito que IND, mas o par de registos HL é incrementado em vez de decrementado.

INIR : Esta instrução actua exactamente como INDR, mas o par de registos HL é incrementado em vez de decrementado.

JP (HL) : Esta instrução provoca um salto para o endereço especificado pelo par de registos HL.

JP (IX)
JP (IY) : Esta instrução provoca um salto para o endereço especificado pelo registo de indexação IX ou IY respectivamente. As flags não são afectadas por estas instruções.

JP nnnn : Esta instrução provoca um salto directo para o endereço nnnn. As flags não são alteradas por ela.

JP C,nnnn
JP M,nnnn
JP NC,nnnn
JP NZ,nnnn
JP P,nnnn
JP PE,nnnn
JP PO,nnnn
JP Z,nnnn : Estas instruções são ignoradas pela CPU a menos que a condição seja satisfeita. Se o for, é executado um salto directo para o endereço nnnn. As flags não são alteradas por estas instruções.

JR dis : Esta instrução provoca um salto relativo. O endereço de destino é formado usando o byte de deslocamento e a convenção de complementação para dois. O deslocamento é calculado a partir do endereço da posição seguinte. JR dis deixa as flags no seu estado anterior.

JR C,dis
JR NC,dis
JR NZ,dis
JR Z,dis : Estas instruções actuam do mesmo modo que JR dis, sendo no entanto ignoradas pela CPU a menos que a condição seja satisfeita.

LD (nnnn),A
LD (nnnn),d2
LD (BC),A
LD (DE),A
LD (HL),r
LD (HL),nn
LD (IX+dis),r
LD (IX+dis),nn
LD (IY+dis),r
LD (IY+dis),nn
LD A,(nnnn)
LD A,(BC)
LD A,(DE)
LD r,(HL)
LD r,(IX+dis)
LD r,(IY+dis)
LD r,(r)
LD r,nn
LD d2,(nnnn)
LD d2,nnnn
LD A,I
LD A,R
LD R,A
LD SP,HL
LD SP,IX
LD SP,IY : A instrução LD possui uma grande quantidade de combinações. Listamos aqui todas as suas formas possíveis. Copia simplesmente um valor do dado que se encontra à direita, que pode ser o conteúdo de um dado endereço ou de um registo simples ou duplo, ou um dado directo, para o destino indicado do lado esquerdo, que pode consistir num registo ou par de registos ou numa posição de memória. Nem todas as combinações são no entanto possíveis, pelo que o leitor deverá verificar, quando programa, se a sintaxe que está a usar se encontra indicada no Apêndice A, a fim de ter a certeza de que não está a escrever um programa impossível...

LDD : O conteúdo da posição de memória indicada pelo par de registos HL é transferido para a posição indicada pelo par de registos DE. DE e HL são em seguida decrementados. BC é decrementado, e se passa a zero a flag P/V será igualmente passada a zero; senão será colocada a 1. As flags H e N passam a zero, mas as outras são mantidas no seu estado anterior.

LDDR : Esta instrução actua exactamente do mesmo modo que LDD, mas se BC não for zero no final da operação esta é repetida. Isto significa que no final a flag P/V será zero, sendo as outras flags afectadas no modo indicado anteriormente.

LDI : Esta instrução actua exactamente como LDD, excepto que HL e DE são incrementados em vez de decrementados. As flags comportam-se exactamente do mesmo modo.

LDIR : Esta instrução comporta-se exactamente como LDDR, mas neste caso os registos HL e DE são incrementados em vez de decrementados.

NEG : Esta instrução "nega" o conteúdo do Acumulador de acordo com a convenção de complementação para dois. Em primeiro lugar todos os bits são complementados ("invertidos"), ou seja, os que são zero passam a um e vice-versa. Em seguida soma-se um ao resultado. As flags S e Z reflectem o resultado da operação, e a flag P/V é passada a 1 se o Acumulador começou, e portanto ter-

minou, com 80 hexadecimal. Se não passa a 0. A flag C passa a 0 se o Acumulador começar e terminar com 0 hexadecimal. Se não é passada a 1. A flag N é sempre passada a 1.

NOP : Esta instrução leva simplesmente a CPU a perder tempo, nada mais fazendo além de incrementar o Contador de Programa para a posição seguinte. Nenhuma das flags é afectada.

OR r : O registo de um único byte r é usado numa operação lógica OR executada sobre o conteúdo do Acumulador. Os bits deste são comparados um a um com os equivalentes do registo r. Se qualquer dos bits ou ambos forem um, o bit correspondente do Acumulador é passado a 1; se não é passado a zero. O registo r não é afectado. A flag C passa a 0, o mesmo acontecendo com a flag N. A flag H passa sempre a 1 e as outras reflectem o resultado da operação.

OR (HL)
OR (IX+dis)
OR (IY+dis)
OR nn : Do mesmo modo já descrito para a instrução anterior, o conteúdo da posição de memória indicado por HL, IX+dis, IY+dis ou um dado directo nn são usados numa operação OR sobre o Acumulador.

OTDR : De um modo semelhante ao descrito para INDR, são transferidos dados da posição de memória indicada por HL para o "port" especificado pelo registo C, actuando o registo B como "byte de contagem" e sendo HL decrementado depois de cada execução da operação.

OTIR : Esta instrução actua exactamente da mesma maneira que OTDR, mas neste caso o par de registos HL é incrementado.

OUT (C),r : Esta instrução envia o dado existente no registo r para o port de entrada/saída especificado pelo registo C. As flags não são alteradas.

OUT nn,A : Esta instrução é semelhante a OUT (C),r, exceptuando o facto de o dado directo nn especificar o port e o dado de uso ser obtido no registo A.

OUTD : Trata-se de uma instrução muito semelhante à ordem IND anteriormente descrita, mas no caso de OUTD o dado é enviado para o port a partir da posição especificada em HL.

OUTI : Esta instrução é semelhante a OUTD, mas após cada operação o par de registos HL é incrementado em vez de decrementado.

POP AF : O valor de dois bytes que se encontra no topo do stack é removido e guardado no par de registos AF. O "Stack Pointer", SP, é então incrementado duas vezes de modo a indicar o dado seguinte. As flags não são alteradas.

POP BC
POP DE
POP HL
POP IX
POP IY : Estas instruções actuam exactamente do mesmo modo que POP AF, mas o valor recolhido do stack é agora guardado nos pares de registos BC, DE, HL, IX e IY respectivamente.

PUSH AF : Esta instrução tem um efeito oposto a

POP AF. O valor guardado no par de registos AF é colocado no stack, sendo o "Stack Pointer" decrementado duas vezes. As flags não são afectadas por esta operação.

PUSH BC
PUSH DE
PUSH HL
PUSH IX
PUSH IY : Estas instruções actuam exactamente do mesmo modo que PUSH AF, excepto o facto de serem guardados no stack os valores contidos respectivamente nos pares de registos BC, DE, HL, IX e IY.

RES x,r : Esta instrução passa o bit x do registo de um só byte r para zero. As flags da CPU não são afectadas.

RES x,HL
RES x,(IX+dis)
RES x,(IY+dis)
respectivamente. : Estas instruções passam a zero o bit x do conteúdo da posição de memória indicada por HL, IX+dis e IY+dis

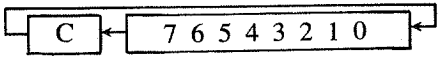
RET : O valor no topo do stack é removido e transferido para o Contador de Programa, PC. O Stack Pointer é incrementado duas vezes de modo a apontar para o dado seguinte. A execução do programa continua utilizando o novo valor contido em PC. As flags não são afectadas.

RET C
RET M
RET NC
RET NZ
RET P
RET PE
RET PO
RET Z : Estas instruções actuam exactamente do mesmo modo que a anterior ordem RET, mas são ignoradas pela CPU a menos que a condição seja satisfeita. As flags não são afectadas.

RET I : Estas duas instruções não serão necessárias ao leitor, mas são muito úteis para o tratamento de interrupções pelo Z80. No Spectrum, as interrupções são usadas para ler o teclado. Esta leitura é interrompida em certos momentos, por exemplo quando a impressora ou a interface de ligação ao gravador estão a ser usadas.

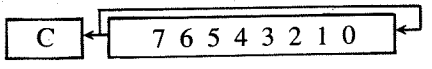
RET N

RL r
RL (HL)
RL (IX+dis)
RL (IY+dis) : Os bits do registo r ou da posição de memória indicada por HL, IX+dis ou IY+dis são rodados pelo bit de transporte, com se mostra na figura seguinte.



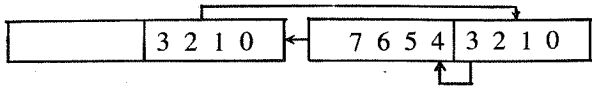
As flags H e N passam ao valor zero. As flags S, Z e P/V reflectem o resultado da operação. A flag P/V reflecte a paridade do registo ou posição de memória rodados.

RLC r
RLC (HL)
RLC (IX+dis)
RLC (IY+dis) : O registo r ou a posição de memória indicada por HL, IX+dis ou IY+dis respectivamente é rodado do modo indicado no diagrama seguinte. As flags comportam-se do mesmo modo que na instrução anterior.

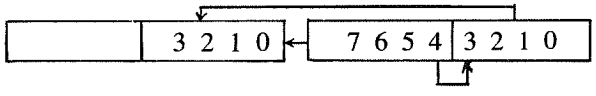


RLD : Esta instrução permite rodar um algarismo de quatro bits, decimal em codificação binária, na metade menos significativa do Acumulador, para a esquerda sobre dois algarismos em memória indi-

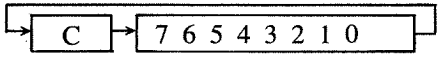
cados por HL. Esta operação é melhor esclarecida pelo seguinte diagrama:



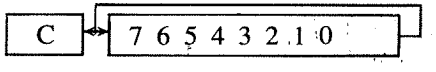
RRD : Esta instrução é bastante semelhante à anterior, rodando igualmente algarismos em codificação binária. Mas neste caso os algarismo são rodados para a direita como se mostra na figura seguinte:



RR R
RR (HL) : Estas instruções actuam de modo semelhante a RL, mas a rotação é agora realizada na direcção oposta como se poderá ver no diagrama:



RRC r
RRC (HL)
RRC (IX + dis)
RRC (IY + dis) : Estas instruções actuam de modo semelhante a RLC, mas a rotação é realizada na direcção oposta como se ilustra na figura seguinte:



RST res : Esta instrução constitui uma versão abreviatura da instrução CALL, restringida aos endereços 00, 08, 10, 18, 20, 28, 30 e 38. RST é abreviatura de RESTART. Estas instruções não afectam as flags.

SBCA,r : Esta instrução subtrai o registo r do Acumulador. O conteúdo da flag "carry" é subtraído ao bit menos significativo do Acumulador. A flag N é passada a 1, mas as outras flags não são alteradas.

SBC A,nn
SBC A,(HL)
SBC A,(IX + dis)
SBC A,(IY + dis) : Esta instrução actua do mesmo modo que SBC A,r, exceptuando o facto de ser subtraído o dado directo nn, ou o conteúdo da posição de memória indicada em HL, IX + dis e IY + dis respectivamente ao Acumulador, em vez do conteúdo de outro registo.

SBC HL,dl : Esta instrução realiza uma subtracção a dois bytes, com transporte. O par de registos dl é subtraído de HL, sendo a flag "carry" subtraída ao bit menos significativo de HL. A flag N é passada a 1. As flags restantes reflectem o estado de HL.

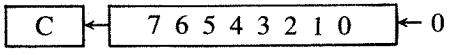
SCF : A flag "carry" é simplesmente passada a 1 por esta instrução. As flags H e N são passadas a zero, e as flags restantes são deixadas sem alteração.

SET x,r : Esta instrução passa a um o bit x do registo r. Nenhuma das flags da CPU é afectada.

SET x,(HL)
SET x,(IX + dis)
SET x,(IY + dis) : Estas instruções passam a um o bit x da posição de memória indicada por HL, IX + dis ou IY + dis respectivamente. Nenhuma das flags é afectada.

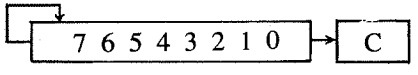
SLA r
SLA (HL)
SLA (IX + dis)
SLA (IY + dis)

: Estas instruções deslocam o registo r, ou o conteúdo das posições de memória indicadas por HL, IX + dis e IY + dis respectivamente, do modo indicado no diagrama seguinte. As flags comportam-se do modo já descrito para RL r.



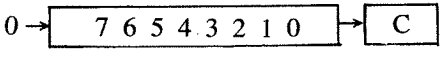
SRA r
SRA (HL)
SRA (IX + dis)
SRA (IY + dis)

: Estas instruções são semelhantes às SLA, mas o deslocamento é neste caso realizado no sentido contrário, como se ilustra em seguida:



SRL r
SRL (HL)
SRL (IX + dis)
SRL (IY + dis)

: Estas instruções deslocam o registo r ou a posição de memória indicada por HL, IX + dis ou IY + dis respectivamente para a esquerda, como se mostra na figura seguinte:



As flags são afectadas do modo indicado para RL.

SUB r

: Esta instrução realiza uma subtracção de um único byte. Subtrai simplesmente o conteúdo do registo r ao Acumulador. O registo r não é alterado, a flag N é passada a um e as restantes flags reflectem o estado em que fica o Acumulador.

SUB (HL)
SUB (IX + dis)

: Estas instruções actuam exactamente do mesmo modo que SUB (r), exceptuando

SUB (IY + dis)
SUB nn

o facto de se subtrair o conteúdo da posição de memória indicada por HL, IX + dis ou IY + dis respectivamente, ou um dado fornecido directamente ao Acumulador.

XOR r

: É realizada uma operação "OR exclusivo" entre o conteúdo de r e o conteúdo do Acumulador. Os bits são comparados um a um. Se o bit do Acumulador e o do registo r forem ambos zero ou um, o bit do Acumulador passa a zero. De outro modo passa a um. A flag "carry" passa a zero, tal como a flag N, a flag H é passada a um e as flags restantes reflectem o resultado no Acumulador.

XOR nn
XOR (HL)
XOR (IX + dis)
XOR (IY + dis)

: Estas instruções actuam do mesmo modo que no caso anterior, mas a operação OR exclusivo é realizada entre o Acumulador e um dado directo ou o conteúdo da posição de memória indicada por HL, IX + dis ou IY + dis respectivamente.

ÍNDICE

APÊNDICES

Fotocomposto por
Nova Força – Lisboa
e impresso na
Empresa Gráfica Feirense, Lda.
Vila da Feira
em 1984
para a Editorial Presença, Lda.