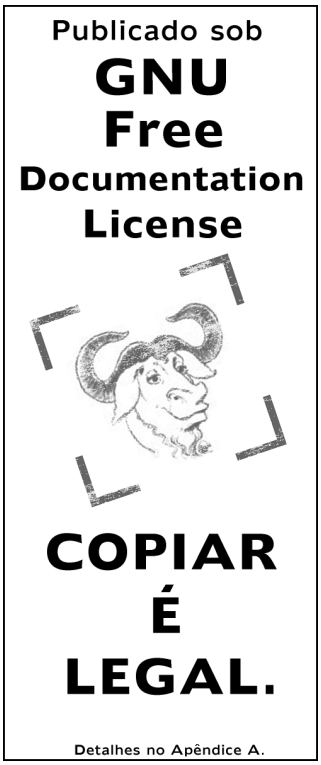
Fudeba Assembler

Manual e Referência da Arquitetura MSX



Felipe Bergo
bergo@seul.org
Novembro, 2002

Sumário

Introdução

O *Fudeba Assembler* é o componente do sistema FOCA[1] responsável por traduzir mnemônicos de estrutura relativamente rígida e invariável em arquivos executáveis.
O **fa** é direcionado à arquitetura MSX, baseada no processador Zilog Z80 de 8 bits, popular no Brasil, Japão, Espanha, Holanda e alguns outros países na década de 1980.
O **fa** é parte do projeto FOCA, que tem o objetivo de disponibilizar gratuitamente uma plataforma de desenvolvimento para o sistema operacional Uzix[2], e disponibilizar o código-fonte de um compilador relativamente simples.
Este documento explica a sintaxe de uso do **fa** , a sintaxe de seus arquivos de entrada e os formatos de saída. Mas além de ser uma referência de uso do **fa** , este livro reune descrições dos diversos componentes de hardware e software do MSX: BIOS, BDOS, Uzix, MSX-DOS, PPI, VDP, PSG e MegaRAM. Os dados contidos neste livro resumem (e em alguns casos corrigem) dados presentes em livros publicados nas décadas de 1980 e 1990 que hoje são verdadeiras raridades.
Além de resumir o conteúdo de vários livros, documentamos boa parte do sistema operacional Uzix, que surpreendeu a comunidade MSX ao implementar multi-tarefa (!), suporte a TCP/IP (!!) e acesso à web (!!!) em um microcomputador de 8 bits com 128 KB de RAM.
Espero que este livro seja útil e que forneça aos novatos (ou esquecidos) informações suficientes para começar (ou retornar) a programar para a arquitetura MSX.

Agradecimentos

Agradeço aos usuários que contribuiram para a realização e aperfeiçoamento desta obra, seja fornecendo informações, emprestando livros de referência, revisando o texto ou ainda fazendo comentários e críticas ao projeto:

Adriano Camargo Rodrigues da Cunha, Ricardo Bittencourt, Marco Antônio Simon Dal Poz, Daniel Caetano, Dante Nishida e Giovanni dos Reis Nunes.

Agradeço aos leitores da lista de discussão MSXBR-L pelo suporte, ao Pink Floyd pela inspiração e à seção de Agradecimentos pela referência recursiva.
Boa leitura!

[1] Fudeba: Otimizador, Compilador e Assembler
[2] http://uzix.sourceforge.net

Capítulo 1

Arquivos de Entrada e Uso do Fudeba Assembler

Programa Exemplo

Os arquivos de entrada do **fa** são arquivos texto simples, processados linha por linha. Linhas são separadas pelo caractere ASCII 10 (hexadecimal 0A), mas o caractere ASCII 13 (hexadecimal 0D) é ignorado se encontrado, permitindo o uso de arquivos gerados em plataformas erráticas como o MS-DOS.
Cada linha no arquivo de entrada contém uma operação a ser interpretada pelo **fa** . As operações são divididas em dois grupos: *diretivas* e *instruções*.
Linhas vazias são permitidas, bem como comentários e rótulos.
Caso o caractere ; (ponto-e-vírgula) ocorra em uma linha, todos os caracteres do ponto-e-vírgula ao final da linha serão ignorados (comentários).
Instruções são aquelas aceitas pelo processador Z80.
Diretivas mudam o comportamento do **fa** , geram dados ou repetem macros de código previamente definidas. Diretivas são sempre iniciadas por um ponto e estão descritas no Capítulo 3.
Cada linha, independentemente de ter instrução, diretiva ou não ter operação, pode associar o endereço de montagem atual a um label (rótulo).
O exemplo abaixo é um arquivo assembly válido (os números de linha não fazem parte do arquivo), que analisaremos a seguir.

```
 1    ; Oi, Mundo MSX!
      ; estas duas linhas sao comentarios

     .header basic
 5   .org    $c000

     ; rotinas do MSX BIOS usadas pelo programa
     .def INIT32 $6f    ; muda tela para screen 1
     .def CHPUT  $a2    ; imprime caractere na tela, %a = caractere
10   .def POSIT  $c6    ; posiciona cursor %h = coluna, %l = linha

     INICIO:
              call INIT32
              ld   %de, MENSAGEM
```

```
15              ld   %ix, POSX      ; (ix) = coluna (ix+1) = linha
     LOOP:      ld   %h, (%ix)
                ld   %l, (%ix+1)
                call POSIT
                inc (%ix)           ; incrementa coluna
20              ld %a, (%de)        ; carrega proximo caractere
                inc %de
                cp 0                ; fim da string ?
                jr z, FIM
                call CHPUT
25              jr LOOP
     FIM:       ld %l, (%ix+1)
                .repeat 2
                inc %l
                .endr
30              ld %h, 0
                call POSIT
                ret                 ; retorna ao basic

     ; dados
35   MENSAGEM: .dz "Oi, Mundo MSX!" ; dz eh uma string tipo "C"
     POSX:     .db 1 1               ; variaveis POSX e POSY
```

INICIO, LOOP, FIM, MENSAGEM e POSX são *labels*. Note que INICIO está definida numa linha vazia, LOOP numa linha com instrução e MENSAGEM numa linha com diretiva.
call, ld, inc são instruções do processador Z80, que serão descritas com maior detalhe no Capítulo 2.
.header, .def, .dz são diretivas, descritas no Capítulo 3.

Nomes, Regras e Limites

Cada linha no arquivo de entrada não deve ultrapassar 255 caracteres. Linhas mais longas causarão erro na montagem.
Para representar números podem ser usadas 3 bases: 10, 2 e 16 (decimal, binário e hexadecimal, respectivamente). O padrão é decimal. Para indicar que um número é hexadecimal ele deve ser precedido por $. Os caracteres permitidos em números hexadecimais são 0-9, A-F e a-f. Números binários devem ser precedidos por #. Exemplos: 27 $1b #11011. É possível também representar o valor ASCII de um caractere colocando-o entre apóstrofos: 'a'. Alguns *escapes* são permitidos para representar caracteres especiais:

Escape	**Valor**
\0	Nil (ASCII 0)
\t	Tab (ASCII 9)
\n	Quebra de linha (ASCII 10)
\r	Retorno de carro (ASCII 13)
\'	Apóstrofo
\"	Aspas
\\	Barra invertida

O **fa** aceita alguns tipos pré-definidos de tokens como partes de seus comandos, a lista abaixo descreve os tipos utilizados e as regras para cada tipo, tal como caracteres permitidos e comprimento máximo.

Tipo	Uso	Caracteres	Limites
byte	constantes numéricas de 8 bits	números: +, - (menos), 0-9 a-f A-F, $, # , ou qualquer caractere (ou escape) entre apóstrofos.	-128 a 255
num	constantes numéricas de 16 bits	idem	-32768 a 65535
endereço	constantes numéricas positivas de 16 bits	idem	0 a 65535
identificador	identificadores (labels e macros)	o primeiro caractere precisa ser uma letra ou underscore (A-Z,a-z,_). Os demais caracteres podem ser dígitos (0-9) também.	63 caracteres de comprimento
string	constantes string	cadeias de caracteres entre aspas ou apóstrofos, admitem escapes	n/a

Maiúsculas e Minúsculas

Nomes de labels e de macros (o tipo *identificador*) são sensíveis à diferença entre minúsculas e maiúsculas, ou seja: LOOP, loop e LooP são diferentes.
Diretivas precisam estar necessariamente em minúsculas.
Constantes caractere ('a') e strings diferenciam minúsculas de maiúsculas como seria de se esperar.
O restante da entrada é insensível a diferenças de maiúsculas e minúsculas: mnemônicos de instruções, nomes de registradores e flags de condição, constantes numéricas hexadecimais.

1.4 Sintaxe de Chamada do FA

O **fa** existe em duas versões independentes: o Fudeba Assembler original (fa), que é bem limitado mas roda nativamente no Uzix, e o XFA (eXtended FA, xfa), ainda em desenvolvimento, que será compatível com o FA porém destinado a ser um cross-assembler para ser executado em arquiteturas de maior porte (PCs, workstations de 32 e 64 bits), oferecendo mensagens de erro menos resumidas e a opção de gerar mapfiles com temporização e endereçamento do código gerado.

1.4.1 Sintaxe do FA

A sintaxe de linha de comando do FA é

```
fa [-q] [-t] [-o arquivo] arquivo1 [arquivo2 ...]
```

A opção -q faz o **fa** operar em modo silencioso, suprimindo algumas mensagens de diagnóstico no console.
A opção -t existe para depurar a conexão entre o fa e o fadb (o processo que serve de banco de dados de labels e macros, usado para ultrapassar o limite de 32 KB por processo no Uzix). Esta opção não tem uso a não ser que você esteja modificando o **fa** ou depurando o Uzix.
A opção -o seleciona o nome do arquivo de saída. Se for omitida, a saída do assembler é gravada em `a.out`.

1.4.2 Sintaxe do XFA

Quando estiver disponível, o XFA aceitará todas as opções do **fa** , e mais a opção `-m`:

```
xfa [-q] [-t] [-m arquivo] [-o arquivo] arquivo1 [arquivo2 ...]
```

A opção `-m` solicita a geração de um mapfile no arquivo especificado. As outras opções têm o mesmo significado que no **fa** .

Capítulo 2

Instruções Z80

Neste capítulo usaremos a seguinte sintaxe para indicar os registradores e operandos permitidos em cada instrução:

reg8 qualquer um entre A, B, C, D, E, H e L.

reg16 BC, DE ou HL.

IX/IY/HL IX, IY ou HL. Note que os registradores índice IX e IY, quando usados como referência indireta (entre parênteses), permitem um deslocamento no formato $IX + inc$, com inc entre -128 e 127.

imm8 valor imediato de 8 bits (0 a 255 ou -128 a 127).

imm16 valor imediato de 16 bits (0 a 65535 ou -32768 a 32767).

ender endereço, seja como **imm16**ou nome de label.

relatender endereço relativo, seja como **imm8**ou nome de label.

cond Condição para a execução da instrução, baseada no registrador de flags: C, NC, Z, NZ, PE, PO, M ou P.

No **fa** os registradores podem estar tanto em maiúsculas como minúsculas, e devem ser precedidos pelo caractere % (porcento).
Cada instrução lista um ou mais diagramas representando o efeito da instrução sobre o registrador de flags (F). Quando sintaxes diversas afetarem os flags de forma diferente, o diagrama de flags indica o efeito do bloco de sintaxe logo acima. Os bits 3 e 5 do registrador F não são utilizados. Para os demais, é usada a seguinte notação:

S	Z		HC		P/O	N	C
⋆	?		1		p	0	-
7	6	5	4	3	2	1	0

Símbolo	Significado
$\star$	O valor do flag é resultado da operação.
?	Não está definido o que ocorre com o flag.
1	O flag é setado.
0	O flag é zerado.
-	O flag não é modificado.
o	O flag P/O é setado em caso de overflow.
p	O flag P/O é setado com a paridade.

E os 6 flags do Z80 são:

Flag	Bit	Descrição
S	7	Sinal (1=negativo, 0=positivo)
Z	6	Zero (1=Zero, 0=Não-Zero)
HC	4	Half-Carry: Carry (vai-um) do bit 3 para o bit 4 em operações de soma ou borrow (empréstimo) do bit 4 para o bit 3 em operações de subtração.
P/O	2	Paridade ou Overflow, depende da instrução.
N	1	Subtract Flag, indica se a última instrução foi subtração (1) ou soma (0).
C	0	Carry (vai-um)

O **fa** não monta as instruções não documentadas do Z80, descritas em [You01] e [RL87]. Os efeitos sobre os flags foram obtidos a partir de [Lev79], [RL87] e [Zil01]. Seria de se esperar que este último, publicado pelo fabricante do Z80, com data de revisão em 2001 (o Z80 foi lançado em 1976!), fosse a referência oficial, mas está repleto de erros e deve ser consultado com cautela. As tabelas de opcodes ao final deste capítulo foram construídas a partir de [RL87]. O conteúdo de alguns flags em algumas instruções são descritos como indefinidos em [Zil01], mas [You01] testou a maioria das instruções e conseguiu chegar a alguma descrição para o Z80. Neste manual mantemos a descrição indefinida, pois CPUs mais novas que sejam "backward-compatible" com o Z80 podem tratar os flags indefinidos de forma diferente.

2.1 Listagem Alfabética Descritiva

ADC *ADC dest, src*

■ Adiciona usando valor do carry flag.

▶ $dest \leftarrow dest + src + CF$

▷ ADC %A , **reg8/imm8**
▷ ADC %A , (%IX /%IY /%HL)
▷ ADC %HL , **reg16**/%SP

S	Z		HC		P/O	N	C
$\star$	$\star$		$\star$		o	0	$\star$
7	6	5	4	3	2	1	0

ADD *ADD dest, src*

■ Adição.

► $dest \leftarrow dest + src$

▷ ADD %A , **reg8/imm8**

▷ ADD %A , (%IX /%IY /%HL)

S	Z		HC		P/O	N	C
⋆	⋆		⋆		o	0	⋆
7	6	5	4	3	2	1	0

▷ ADD %IX , %BC /%DE /%IX /%SP

▷ ADD %IY , %BC /%DE /%IY /%SP

▷ ADD %HL , **reg16**/%SP

S	Z		HC		P/O	N	C
-	-		⋆		o	0	⋆
7	6	5	4	3	2	1	0

AND *AND src*

■ Operação E lógico.

► $\%A \leftarrow \%A \wedge src$

▷ AND **reg8/imm8**

▷ AND (%HL /%IX /%IY)

S	Z		HC		P/O	N	C
⋆	⋆		1		p	0	0
7	6	5	4	3	2	1	0

BIT *BIT n, src*

■ Testa um bit, resultado na ZF (Zero Flag). [You01] descreve o resultado dos flags S e P/V baseado em experimentos, mas [Zil01] descreve ambos como indefinidos.

► $ZF \leftarrow \overline{bit_n(src)}$

▷ BIT bitn, **reg8**/(%HL /%IX /%IY)

S	Z		HC		P/O	N	C
?	⋆		1		?	0	-
7	6	5	4	3	2	1	0

CALL

CALL [cond ,] dest

■ Chama subrotina.

► $\%SP \leftarrow \%SP - 2$
$(\%SP) \leftarrow PC + 3$
$PC \leftarrow dest$

▷ CALL **ender**

▷ CALL **cond**, **ender**

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

CCF

CCF

■ Complementa (inverte) o carry flag (CF). O carry anterior é copiado no flag HC.

► $CF \leftarrow \overline{CF}$

▷ CCF

S	Z		HC		P/O	N	C
-	-		⋆		-	0	⋆
7	6	5	4	3	2	1	0

CP

CP src

■ Compara valor com o acumulador e ajusta flags de acordo com o resultado.

► $\%F \leftarrow \varphi(\%A, src)$

▷ CP **reg8**/**imm8**/(%HL /%IX /%IY)

S	Z		HC		P/O	N	C
⋆	⋆		⋆		o	1	⋆
7	6	5	4	3	2	1	0

CPD *CPD*

■ Equivale a CP %A , (%HL) , DEC %HL , DEC %BC .

► $\%F \leftarrow \varphi(\%A, (\%HL))$
$\%HL \leftarrow \%HL - 1$
$\%BC \leftarrow \%BC - 1$

▷ CPD

S	Z		HC		P/O	N	C
⋆	⋆		⋆		⋆	1	-
7	6	5	4	3	2	1	0

CPDR *CPDR*

■ Repete CPD até que %BC seja zero.

► *enquanto* $\%BC \neq 0$ {
$\%F \leftarrow \varphi(\%A, (\%HL))$
$\%HL \leftarrow \%HL - 1$
$\%BC \leftarrow \%BC - 1$
}

▷ CPDR

S	Z		HC		P/O	N	C
⋆	⋆		⋆		⋆	1	-
7	6	5	4	3	2	1	0

CPI *CPI*

■ Equivale a CP %A , (%HL) , INC %HL , DEC %BC .

► $\%F \leftarrow \varphi(\%A, (\%HL))$
$\%HL \leftarrow \%HL + 1$
$\%BC \leftarrow \%BC - 1$

▷ CPI

S	Z		HC		P/O	N	C
⋆	⋆		⋆		⋆	1	-
7	6	5	4	3	2	1	0

CPIR *CPIR*

■ Repete CPI até que %BC seja zero.

► *enquanto* $\%BC \neq 0$ {
$\%F \leftarrow \varphi(\%A, (\%HL))$
$\%HL \leftarrow \%HL + 1$
$\%BC \leftarrow \%BC - 1$
}

▷ CPIR

S	Z		HC		P/O	N	C
⋆	⋆		⋆		⋆	1	-
7	6	5	4	3	2	1	0

CPL *CPL*

■ Complementa o acumulador.

► $\%A \leftarrow \overline{\%A}$

▷ CPL

S	Z		HC		P/O	N	C
-	-		1		-	1	-
7	6	5	4	3	2	1	0

DAA *DAA*

■ Ajusta o acumulador para que seus nibbles representem os dígitos decimais de seu valor inicial.

► $\%A_{0-3} \leftarrow \%A \mod 10$
$\%A_{4-7} \leftarrow \%A \mathit{idiv} 10$

▷ DAA

S	Z		HC		P/O	N	C
⋆	⋆		⋆		p	-	⋆
7	6	5	4	3	2	1	0

DEC *DEC dest*

■ Decrementa valor em um.

► $dest \leftarrow dest - 1$

▷ DEC **reg8**/(%IX /%IY /%HL)

S	Z		HC		P/O	N	C
⋆	⋆		⋆		o	1	-
7	6	5	4	3	2	1	0

▷ DEC **reg16**/%IX /%IY /%SP

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

DI *DI*

■ Desabilita interrupções.

▷ DI

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

DJNZ *DJNZ dest*

■ Decrementa %B e salta para destino se %B não for zero.

► $\%B \leftarrow \%B - 1$
se $\%B \neq 0$ {
$PC \leftarrow dest$
}

▷ DJNZ relatender

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

EI *EI*

■ Habilita interrupções.

▷ EI

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

EX *EX dest, src*

■ Troca valores de registradores. A sintaxe EX %AF troca o par %AF com os registradores-sombra A'F'. No **fa** , ao contrário de vários outros assemblers, a sintaxe é apenas EX %AF em vez de EX %AF , %A'F'.

▶ $tmp \leftarrow src$
$src \leftarrow dest$
$dest \leftarrow tmp$

▷ EX %DE , %HL

▷ EX (%SP), %HL /%IX /%IY

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

▷ EX %AF

S	Z		HC		P/O	N	C
★	★		★		★	★	★
7	6	5	4	3	2	1	0

EXX *EXX*

■ Troca os valores dos registradores %B , %C , %D , %E , %H , %L com seus registradores-sombra.

► $troca(\%B, \%B')$
$troca(\%C, \%C')$
$troca(\%D, \%D')$
$troca(\%E, \%E')$
$troca(\%H, \%H')$
$troca(\%L, \%L')$
onde $troca(\alpha, \beta)$ é $tmp \leftarrow \beta$, $\beta \leftarrow \alpha$, $\alpha \leftarrow tmp$.

▷ EXX

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

HALT *HALT*

■ Pára o processamento do Z80.

► ∞ ;-)

▷ HALT

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

IM *IM n*

■ Muda o modo de interrupção do Z80. Existem 3 modos: 0, 1 e 2. No modo 0 o dispositivo deve fornecer (via pinagem do Z80) uma instrução RST para tratamento da interrupção, no modo 1 as interruções são sempre tratadas pelo RST $38 e no modo 2 o tratador de interrupção está em um endereço cujo MSB é o valor do registrador especial %I e o LSB é fornecido pelo dispositivo de forma semelhante ao modo 0.

▷ IM **imm8**

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

IN

IN dest, src

■ Lê um byte na porta de I/O src para dest.

► $dest \leftarrow IO(src)$

▷ IN %A , **imm8**

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

▷ IN **reg8**, (%C)

S	Z		HC		P/O	N	C
⋆	⋆		0		p	0	-
7	6	5	4	3	2	1	0

INC

INC dest

■ Incrementa valor em um.

► $dest \leftarrow dest + 1$

▷ INC **reg8**/(%IX /%IY /%HL)

S	Z		HC		P/O	N	C
⋆	⋆		⋆		o	0	-
7	6	5	4	3	2	1	0

▷ INC **reg16**/%IX /%IY /%SP

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

IND

IND

■ Lê byte em porta de I/O, decrementa %HL e %B .

► $(\%HL) \leftarrow IO((\%C))$
$\%HL \leftarrow \%HL - 1$
$\%B \leftarrow \%B - 1$

▷ IND

S	Z		HC		P/O	N	C
?	⋆		?		?	1	-
7	6	5	4	3	2	1	0

INDR *INDR*

■ Repete IND até que %B seja zero.

► *enquanto* $\%B \neq 0$ {
$(\%HL) \leftarrow IO((\%C))$
$\%HL \leftarrow \%HL - 1$
$\%B \leftarrow \%B - 1$
}

▷ INDR

S	Z		HC		P/O	N	C
?	1		?		?	1	-
7	6	5	4	3	2	1	0

INI *INI*

■ Lê byte em porta de I/O, incrementa %HL e decrementa %B .

► $(\%HL) \leftarrow IO((\%C))$
$\%HL \leftarrow \%HL + 1$
$\%B \leftarrow \%B - 1$

▷ INI

S	Z		HC		P/O	N	C
?	⋆		?		?	1	-
7	6	5	4	3	2	1	0

INIR *INIR*

■ Repete INI até que %B seja zero.

► *enquanto* $\%B \neq 0$ {
$(\%HL) \leftarrow IO((\%C))$
$\%HL \leftarrow \%HL + 1$
$\%B \leftarrow \%B - 1$
}

▷ INIR

S	Z		HC		P/O	N	C
?	1		?		?	1	-
7	6	5	4	3	2	1	0

JP

JP [cond,] dest

■ Transfere a execução para dest. Usa endereçamento absoluto. O modo (%IX/Y) não permite deslocamento em %IX/Y .

► $PC \leftarrow dest$

▷ JP ender

▷ JP (%HL /%IX /%IY)

▷ JP cond, ender

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

JR

JR [cond,] dest

■ Transfere a execução para dest. Usa endereçamento relativo. Apenas as condições C, NC, Z e NZ podem ser usadas.

► $PC \leftarrow dest$

▷ JR relatender

▷ JR cond, relatender

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

LD *LD dest, src*

■ Transfere src para dest.

► $dest \leftarrow src$

▷ LD **reg8**, **reg8**
▷ LD **reg8**, **imm8**
▷ LD **reg16**/%SP /%IX /%IY , **imm16**/ender
▷ LD (%HL /%IX /%IY), **imm8**/**reg8**
▷ LD (**reg16**), %A
▷ LD %A , (**reg16**/%IX /%IY)
▷ LD **reg8**, (%HL /%IX /%IY)
▷ LD %A /**reg16**/%IX /%IY /%SP , (ender)
▷ LD (ender), %A /**reg16**/%IX /%IY /%SP
▷ LD %SP , %HL /%IX /%IY
▷ LD %I /%R , %A

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

▷ LD %A , %I /%R

S	Z		HC		P/O	N	C
⋆	⋆		0		⋆	0	-
7	6	5	4	3	2	1	0

LDD *LDD*

■ Carrega conteúdo de %HL para %DE , decrementa %DE , %HL e %BC .

► $(\%DE) \leftarrow (\%HL)$
$\%DE \leftarrow \%DE - 1$
$\%HL \leftarrow \%HL - 1$
$\%BC \leftarrow \%BC - 1$

▷ LDD

S	Z		HC		P/O	N	C
-	-		0		⋆	0	-
7	6	5	4	3	2	1	0

LDDR *LDDR*

■ Repete LDD até que %BC seja zero.

► *enquanto* $\%BC \neq 0$ {
$(\%DE) \leftarrow (\%HL)$
$\%DE \leftarrow \%DE - 1$
$\%HL \leftarrow \%HL - 1$
$\%BC \leftarrow \%BC - 1$
}

▷ LDDR

S	Z		HC		P/O	N	C
-	-		0		0	0	-
7	6	5	4	3	2	1	0

LDI *LDI*

■ Carrega conteúdo de %HL para %DE , incrementa %DE , %HL , decrementa %BC .

► $(\%DE) \leftarrow (\%HL)$
$\%DE \leftarrow \%DE + 1$
$\%HL \leftarrow \%HL + 1$
$\%BC \leftarrow \%BC - 1$

▷ LDI

S	Z		HC		P/O	N	C
-	-		0		⋆	0	-
7	6	5	4	3	2	1	0

LDIR *LDIR*

■ Repete LDI até que %BC seja zero.

► *enquanto* $\%BC \neq 0$ {
$(\%DE) \leftarrow (\%HL)$
$\%DE \leftarrow \%DE + 1$
$\%HL \leftarrow \%HL + 1$
$\%BC \leftarrow \%BC - 1$
}

▷ LDIR

S	Z		HC		P/O	N	C
-	-		0		0	0	-
7	6	5	4	3	2	1	0

NEG *NEG*

■ Inverte o sinal do acumulador.

► $\%A \leftarrow -\%A$

▷ NEG

S	Z		HC		P/O	N	C
⋆	⋆		⋆		o	1	⋆
7	6	5	4	3	2	1	0

NOP *NOP*

■ Não faz nada.

▷ NOP

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

OR *OR src*

■ Operação OU lógico.

► $\%A \leftarrow \%A \vee src$

▷ OR **reg8/imm8**

▷ OR (%HL /%IX /%IY)

S	Z		HC		P/O	N	C
⋆	⋆		1		p	0	0
7	6	5	4	3	2	1	0

OTDR *OTDR*

■ Repete OUTD até que %B seja zero.

► *enquanto* $\%B \neq 0$ {
$IO((\%C)) \leftarrow (\%HL)$
$\%HL \leftarrow \%HL - 1$
$\%B \leftarrow \%B - 1$
}

▷ OTDR

S	Z		HC		P/O	N	C
?	1		?		?	1	-
7	6	5	4	3	2	1	0

OTIR *OTIR*

■ Repete OUTI até que %B seja zero.

► *enquanto* $\%B \neq 0$ {
$IO((\%C)) \leftarrow (\%HL)$
$\%HL \leftarrow \%HL + 1$
$\%B \leftarrow \%B - 1$
}

▷ OTIR

S	Z		HC		P/O	N	C
?	1		?		?	1	-
7	6	5	4	3	2	1	0

OUT *OUT dest, src*

■ Escreve src na porta de I/O dest.

► $IO(dest) \leftarrow src$

▷ OUT **imm8**, %A

▷ OUT (%C), **reg8**

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

OUTD *OUTD*

■ Escreve (%HL) na porta (%C), decrementa %HL , decrementa %B .

► $IO((\%C)) \leftarrow (\%HL)$
$\%HL \leftarrow \%HL - 1$
$\%B \leftarrow \%B - 1$

▷ OUTD

S	Z		HC		P/O	N	C
?	★		?		?	1	-
7	6	5	4	3	2	1	0

OUTI *OUTI*

■ Escreve (%HL) na porta (%C), incrementa %HL , decrementa %B .

► $IO((\%C)) \leftarrow (\%HL)$
$\%HL \leftarrow \%HL + 1$
$\%B \leftarrow \%B - 1$

▷ OUTI

S	Z		HC		P/O	N	C
?	★		?		?	1	-
7	6	5	4	3	2	1	0

POP *POP dest*

■ Desempilha para dest.

► $dest \leftarrow (\%SP)$
$\%SP \leftarrow \%SP + 2$

▷ POP **reg16**/%AF /%IX /%IY

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

PUSH *PUSH src*

■ Empilha src.

► $\%SP \leftarrow \%SP - 2$
$(\%SP) \leftarrow src$

▷ PUSH **reg16**/%AF /%IX /%IY

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

RDL *RDL*

■ Sinônimo para RLD.

RES *RES n, dest*

■ Reseta bit n em dest.

► $bit_n(dest) \leftarrow 0$

▷ RES bitn, **reg8**/(%HL /%IX /%IY)

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

RET *RET [cond]*

■ Retorna de sub-rotina.

► $PC \leftarrow (\%SP)$
$\%SP \leftarrow \%SP + 2$

▷ RET

▷ RET cond

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

RETI *RETI*

■ Retorna de tratador de interrupção mascarável. Semelhante a RET mas comunica ao dispositivo que o tratamento da interrupção terminou. RETI não habilita as interrupções, em geral é necessária uma instrução EI antes de RETI em tratadores de interrupção.

▶ $PC \leftarrow (\%SP)$
$\%SP \leftarrow \%SP + 2$

▷ RETI

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

RETN *RETN*

■ Retorna de tratador de interrupção não-mascarável (NMI). Restaura o estado das interrupções mascaráveis (EI ou DI) para o valor anterior à ocorrência da NMI.

▶ $PC \leftarrow (\%SP)$
$\%SP \leftarrow \%SP + 2$

▷ RETN

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

RL *RL dest*

■ Rotação à esquerda com carry.

▶ ⊙Veja Figura 2.1

▷ RL **reg8**/(%HL /%IX /%IY)

S	Z		HC		P/O	N	C
⋆	⋆		0		p	0	⋆
7	6	5	4	3	2	1	0

RLA *RLA*

■ RL sobre o acumulador.

► ⊙Veja Figura 2.1

▷ RLA

S	Z		HC		P/O	N	C
-	-		0		-	0	⋆
7	6	5	4	3	2	1	0

RLC *RLC dest*

■ Rotação à esquerda sem carry.

► ⊙Veja Figura 2.1

▷ RLC **reg8**/(%HL /%IX /%IY)

S	Z		HC		P/O	N	C
⋆	⋆		0		p	0	⋆
7	6	5	4	3	2	1	0

RLCA *RLCA*

■ RLC sobre o acumulador.

► ⊙Veja Figura 2.1

▷ RLCA

S	Z		HC		P/O	N	C
-	-		0		-	0	⋆
7	6	5	4	3	2	1	0

RLD *RLD*

■ Rotação decimal (de nibbles) à esquerda.

► $tmp \leftarrow \%A_{0-3}$
$\%A_{0-3} \leftarrow (\%HL)_{4-7}$
$(\%HL)_{4-7} \leftarrow (\%HL)_{0-3}$
$(\%HL)_{0-3} \leftarrow tmp$

▷ RLD

S	Z		HC		P/O	N	C
⋆	⋆		0		p	0	-
7	6	5	4	3	2	1	0

RR *RR dest*

■ Rotação à direita com carry.

► ⊙Veja Figura 2.1

▷ RR **reg8**/(%HL /%IX /%IY)

S	Z		HC		P/O	N	C
⋆	⋆		0		p	0	⋆
7	6	5	4	3	2	1	0

RRA *RRA*

■ RR sobre o acumulador.

► ⊙Veja Figura 2.1

▷ RRA

S	Z		HC		P/O	N	C
-	-		0		-	0	⋆
7	6	5	4	3	2	1	0

RRC *RRC dest*

■ Rotação à direita sem carry.

► ⊙Veja Figura 2.1

▷ RRC **reg8**/(%HL /%IX /%IY)

S	Z		HC		P/O	N	C
⋆	⋆		0		p	0	⋆
7	6	5	4	3	2	1	0

RRCA *RRCA*

■ RRC sobre o acumulador.

► ⊙Veja Figura 2.1

▷ RRCA

S	Z		HC		P/O	N	C
-	-		0		-	0	⋆
7	6	5	4	3	2	1	0

RRD *RRD*

■ Rotação decimal (de nibbles) à direita.

► $tmp \leftarrow \%A_{0-3}$
$\%A_{0-3} \leftarrow (\%HL)_{0-3}$
$(\%HL)_{0-3} \leftarrow (\%HL)_{4-7}$
$(\%HL)_{4-7} \leftarrow tmp$

▷ RRD

S	Z		HC		P/O	N	C
⋆	⋆		0		p	0	-
7	6	5	4	3	2	1	0

RST *RST dest*

■ Chama subrotina de sistema. Semelhante a CALL mas ocupa apenas um byte.

► $\%SP \leftarrow \%SP - 2$
$(\%SP) \leftarrow PC + 1$
$PC \leftarrow dest$

▷ RST $0/$8/$10/$18/$20/$28/$30/$38

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

SBC *SBC dest, src*

■ Subtração com Carry

► $dest \leftarrow dest - src - CF$

▷ SBC %A , **reg8/imm8**/(%HL /%IX /%IY)

▷ SBC %HL , **reg16**/%SP

S	Z		HC		P/O	N	C
★	★		★		O	1	★
7	6	5	4	3	2	1	0

SCF *SCF*

■ Seta o carry flag (CF).

► $CF \leftarrow 1$

▷ SCF

S	Z		HC		P/O	N	C
-	-		0		-	0	1
7	6	5	4	3	2	1	0

SET *SET n, dest*

■ Seta bit n em dest.

► $bit_n(dest) \leftarrow 1$

▷ SET bitn, **reg8**/(%HL /%IX /%IY)

S	Z		HC		P/O	N	C
-	-		-		-	-	-
7	6	5	4	3	2	1	0

SLA *SLA dest*

■ Deslocamento aritmético à esquerda.

► ⊙Veja Figura 2.1

▷ SLA **reg8**/(%HL /%IX /%IY)

S	Z		HC		P/O	N	C
⋆	⋆		0		p	0	⋆
7	6	5	4	3	2	1	0

SRA *SRA dest*

■ Deslocamento aritmético à direita.

► ⊙Veja Figura 2.1

▷ SRA **reg8**/(%HL /%IX /%IY)

S	Z		HC		P/O	N	C
⋆	⋆		0		p	0	⋆
7	6	5	4	3	2	1	0

SRL *SRL dest*

■ Deslocamento lógico à direita.

► ⊙Veja Figura 2.1

▷ SRL **reg8**/(%HL /%IX /%IY)

S	Z		HC		P/O	N	C
0	⋆		0		p	0	⋆
7	6	5	4	3	2	1	0

SUB

SUB dest, src

■ Subtração

► $dest \leftarrow dest - src$

▷ SUB %A , **reg8/imm8**/(%HL /%IX /%IY)

S	Z		HC		P/O	N	C
⋆	⋆		⋆		o	1	⋆
7	6	5	4	3	2	1	0

XOR

XOR src

■ Operação OU-exclusivo lógico.

► $\%A \leftarrow \%A \oplus src$

▷ XOR **reg8/imm8**

▷ XOR (%HL /%IX /%IY)

S	Z		HC		P/O	N	C
⋆	⋆		1		p	0	0
7	6	5	4	3	2	1	0

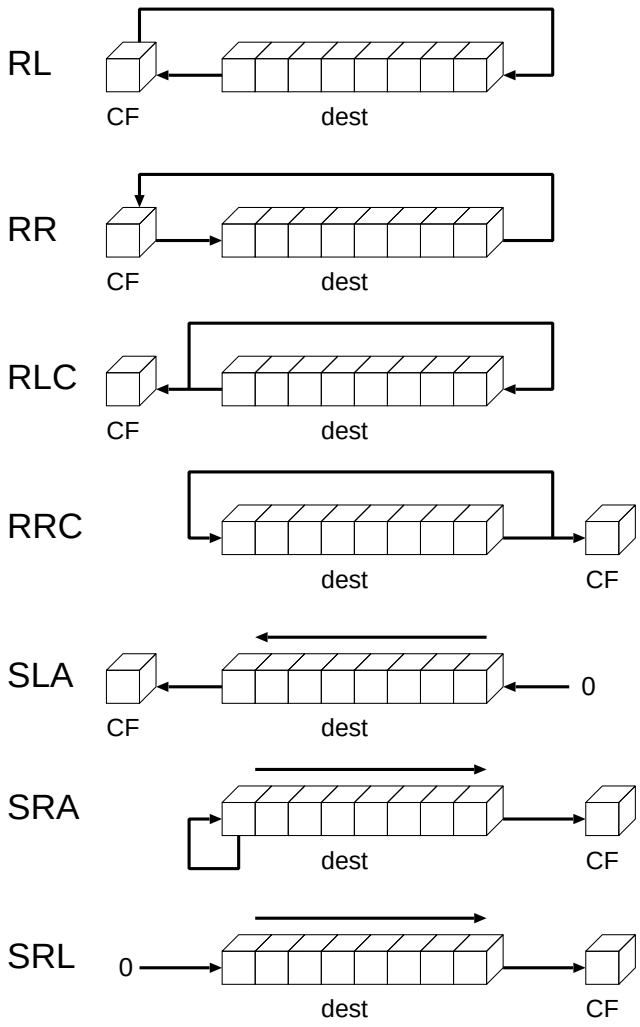


Figura 2.1: Instruções de Deslocamento e Rotação

2.2 Temporização das Instruções

A Tabela 2.1 (abaixo) lista o tamanho de cada instrução montada (em bytes), o número de ciclos de CPU necessários para execução, número de wait states no MSX (implementações de MSX baseadas em "clones" do Z80 podem não corresponder aos valores listados) e número total de ciclos para execução, considerando os wait-states.
Algumas instruções (em geral as condicionais) não têm número de ciclos fixo. Nestes casos, o menor número indica o comportamento se a ação condicional não for executada.

Tabela 2.1: Temporização das Instruções do Z80

Instrução	**Tamanho**	**Ciclos**	**Wait-States**	**Total**
ADC %A , (%HL)	1	7	1	8

⟶

Tabela 2.1: (cont.)

Instrução	**Tamanho**	**Ciclos**	**Wait-States**	**Total**
ADC %A , (%IX/Y + ss)	3	19	2	21
ADC %A , **imm8**	2	7	1	8
ADC %A , **reg8**	1	4	1	5
ADC %HL , **reg16**/ %SP	2	15	2	17
ADD %A , (%HL)	1	7	1	8
ADD %A , (%IX/Y + ss)	3	19	2	21
ADD %A , **imm8**	2	7	1	8
ADD %A , **reg8**	1	4	1	5
ADD %HL , **reg16**/ %SP	1	11	1	12
ADD %IX/Y , %BC /%DE /src	2	15	2	17
AND (%HL)	1	7	1	8
AND (%IX/Y + ss)	3	19	2	21
AND **imm8**	2	7	1	8
AND **reg8**	1	4	1	5
BIT b, (%HL)	2	12	2	14
BIT b, (%IX/Y + ss)	4	20	3	23
BIT b, **reg8**	2	8	2	10
CALL **ender**	3	17	1	18
CALL cond, **ender**	3	17/10	1	18/11
CCF	1	4	1	5
CP (%HL)	1	7	1	8
CP (%IX/Y + ss)	3	19	2	21
CP **imm8**	2	7	1	8
CP **reg8**	1	4	1	5
CPD	2	16	2	18
CPDR	2	21/21/16	2	23/23/18
CPI	2	16	2	18
CPIR	2	21/21/16	2	23/23/18
CPL	1	4	1	5
DAA	1	4	1	5
DEC (%HL)	1	11	1	12
DEC (%IX/Y + ss)	3	23	2	25
DEC **reg8**	1	4	1	5
DEC **reg16**/ %SP	1	6	1	7
DEC %IX/Y	2	10	2	12
DI	1	4	1	5
DJNZ ss	2	13/8	1	14/9
EI	1	4	1	5
EX (%SP) , %HL	1	19	1	20
EX (%SP) , %IX/Y	2	23	2	25
EX %AF	1	4	1	5
EX %DE , %HL	1	4	1	5
EXX	1	4	1	5
HALT	1	4	1	5
IM n	2	8	2	10
IN A, **imm8**	2	11	1	12

⟶

Tabela 2.1: (cont.)

Instrução	**Tamanho**	**Ciclos**	**Wait-States**	**Total**
IN **reg8**, (%C)	2	12	2	14
INC (%HL)	1	11	1	12
INC (%IX/Y + ss)	3	23	2	25
INC **reg8**	1	4	1	5
INC **reg16**/ %SP	1	6	1	7
INC %IX/Y	2	10	2	12
IND	2	16	2	18
INDR	2	21/21/16	2	23/23/18
INI	2	16	2	18
INIR	2	21/21/16	2	23/23/18
JP (%HL)	1	4	1	5
JP (%IX/Y)	2	8	2	10
JP **ender**	3	10	1	11
JP COND, **ender**	3	10	1	11
JR ss	2	12	1	13
JR COND, ss	2	12/7	1	13/8
LD (**ender**), %A	3	13	1	14
LD (**ender**), %HL	3	16	1	17
LD (**ender**), %IX/Y	4	20	2	22
LD (**reg16**), %A	1	7	1	8
LD (%HL), **imm8**	2	10	1	11
LD (%HL), **reg8**	1	7	1	8
LD (%IX/Y + ss), **imm8**	4	19	2	21
LD (%IX/Y + ss), **reg8**	3	19	2	21
LD %A , (**ender**)	3	13	1	14
LD %A , (**reg16**)	1	7	1	8
LD %A , %I / %R	2	9	2	11
LD **reg8**, (%HL)	1	7	1	8
LD **reg8**, (%IX/Y + ss)	3	19	2	21
LD **reg8**, **imm8**	2	7	1	8
LD %I /%R , %A	2	9	2	11
LD **reg8**, **reg8**	1	4	1	5
LD %BC /%DE /%SP , (**ender**)	4	20	1(2)	21(22)
LD **reg16**/%SP , **imm16**	3	10	1	11
LD %HL , (**ender**)	3	16	1	17
LD %IX/Y , (**ender**)	4	20	2	22
LD %IX/Y , **imm16**	4	14	2	16
LD %SP , %HL	1	6	1	7
LD %SP , %IX/Y	2	10	2	12
LDD	2	16	2	18
LDDR	2	21/21/16	2	23/23/18
LDI	2	16	2	18
LDIR	2	21/21/16	2	23/23/18
NEG	2	8	2	10
NOP	1	4	1	5
OR (%HL)	1	7	1	8

⟶

Tabela 2.1: (cont.)

Instrução	**Tamanho**	**Ciclos**	**Wait-States**	**Total**
OR (%IX/Y + ss)	3	19	2	21
OR **imm8**	2	7	1	8
OR **reg8**	1	4	1	5
OTDR	2	21/21/16	2	23/23/18
OTIR	2	21/21/16	2	23/23/18
OUT **imm8**, %A	2	11	1	12
OUT (%C), **reg8**	2	12	2	14
OUTD	2	21/21/16	2	23/23/18
OUTI	2	21/21/16	2	23/23/18
POP **reg16**/%AF	1	10	1	11
POP %IX/Y	2	14	2	16
PUSH **reg16**/%AF	1	11	1	12
PUSH %IX/Y	2	15	2	17
RES b, (%HL)	2	15	2	17
RES b, (%IX/Y + ss)	4	23	3	26
RES b, **reg8**	2	8	2	10
RET	1	10	1	11
RET COND	1	11/5	1	12/6
RETI	2	14	2	16
RETN	2	14	2	16
RL (%HL)	2	15	2	17
RL (%IX/Y + ss)	4	23	3	26
RL **reg8**	2	8	2	10
RLA	1	4	1	5
RLC (%HL)	2	15	2	17
RLC (%IX/Y + ss)	4	23	3	26
RLC **reg8**	2	8	2	10
RLCA	1	4	1	5
RLD	2	18	2	20
RR (%HL)	2	15	2	17
RR (%IX/Y + ss)	4	23	3	26
RR **reg8**	2	8	2	10
RRA	1	4	1	5
RRC (%HL)	2	15	2	17
RRC (%IX/Y + ss)	4	23	3	26
RRC **reg8**	2	8	2	10
RRCA	1	4	1	5
RRD	2	18	2	20
RST nn	1	11	1	12
SBC %A , (%HL)	1	7	1	8
SBC %A , (%IX/Y + ss)	3	19	2	21
SBC %A , **imm8**	2	7	1	8
SBC %A , **reg8**	1	4	1	5
SBC %HL , **reg16**/%SP	2	15	2	17
SET b, (%HL)	2	15	2	17
SET b, (%IX/Y + ss)	4	23	3	26

⟶

Tabela 2.1: (cont.)

Instrução	**Tamanho**	**Ciclos**	**Wait-States**	**Total**
SET b, **reg8**	2	8	2	10
SLA (%HL)	2	15	2	17
SLA (%IX/Y + ss)	4	23	3	26
SLA **reg8**	2	8	2	10
SRA (%HL)	2	15	2	17
SRA (%IX/Y + ss)	4	23	3	26
SRA **reg8**	2	8	2	10
SRL (%HL)	2	15	2	17
SRL (%IX/Y + ss)	4	23	3	26
SRL **reg8**	2	8	2	10
SUB %A , (%HL)	1	7	1	8
SUB %A , (%IX/Y + ss)	3	19	2	21
SUB %A , **imm8**	2	7	1	8
SUB %A , **reg8**	1	4	1	5
XOR (%HL)	1	7	1	8
XOR (%IX/Y + ss)	3	19	2	21
XOR **imm8**	2	7	1	8
XOR **reg8**	1	4	1	5

Tabela 2.1: Temporização das Instruções do Z80

2.3 Mapeamento dos Opcodes

A sintaxe das instruções nas tabelas desta seção foi modificada para reduzir o tamanho da tabela. Os parâmetros *ss*, *tt* e *qq* são o segundo, terceiro e quarto bytes da instrução, respectivamente. *ttss* indica que o terceiro byte contem a parte mais significativa do argumento de 16 bits.

	0	1	2	3	4	5	6	7
0	NOP	LD bc, ttss	LD (bc),a	INC bc	INC b	DEC b	LD b,ss	RLCA
1	DJNZ ss	LD de, ttss	LD (de),a	INC de	INC d	DEC d	LD d,ss	RLA
2	JR nz,ss	LD hl, ttss	LD (ttss),hl	INC hl	INC h	DEC h	LD h,ss	DAA
3	JR nc,ss	LD sp, ttss	LD (ttss),a	INC sp	INC (hl)	DEC (hl)	LD (hl),ss	SCF
4	LD b,b	LD b,c	LD b,d	LD b,e	LD b,h	LD b,l	LD b,(hl)	LD b,a
5	LD d,b	LD d,c	LD d,d	LD d,e	LD d,h	LD d,l	LD d,(hl)	LD d,a
6	LD h,b	LD h,c	LD h,d	LD h,e	LD h,h	LD h,l	LD h,(hl)	LD h,a
7	LD (hl),b	LD (hl),c	LD (hl),d	LD (hl),e	LD (hl),h	LD (hl),l	HALT	LD (hl),a
8	ADD a,b	ADD a,c	ADD a,d	ADD a,e	ADD a,h	ADD a,l	ADD a,(hl)	ADD a,a
9	SUB a,b	SUB a,c	SUB a,d	SUB a,e	SUB a,h	SUB a,l	SUB a,(hl)	SUB a,a
A	AND b	AND c	AND d	AND e	AND h	AND l	AND (hl)	AND a
B	OR b	OR c	OR d	OR e	OR h	OR l	OR (hl)	OR a
C	RET nz	POP bc	JP nz,ttss	JP ttss	CALL nz,ttss	PUSH bc	ADD a,ss	RST 0
D	RET nc	POP de	JP nc,ttss	OUT ss,a	CALL nc,ttss	PUSH de	SUB a,ss	RST 16
E	RET po	POP hl	JP po,ttss	EX (sp),hl	CALL po,ttss	PUSH hl	AND ss	RST 32
F	RET p	POP af	JP p,ttss	DI	CALL p,ttss	PUSH af	OR ss	RST 48

Tabela 2.2: Opcodes simples, LSN (Least Significant Nibble) 0-7

	8	9	A	B	C	D	E	F
0	EX af	ADD hl,bc	LD a,(bc)	DEC bc	INC c	DEC c	LD c,ss	RRCA
1	JR ss	ADD hl,de	LD a,(de)	DEC de	INC e	DEC e	LD e,ss	RRA
2	JR z,ss	ADD hl,hl	LD hl,(ttss)	DEC hl	INC l	DEC l	LD l,ss	CPL
3	JR c,ss	ADD hl,sp	LD a,(ttss)	DEC sp	INC a	DEC a	LD a,ss	CCF
4	LD c,b	LD c,c	LD c,d	LD c,e	LD c,h	LD c,l	LD c,(hl)	LD c,a
5	LD e,b	LD e,c	LD e,d	LD e,e	LD e,h	LD e,l	LD e,(hl)	LD e,a
6	LD l,b	LD l,c	LD l,d	LD l,e	LD l,h	LD l,l	LD l,(hl)	LD l,a
7	LD a,b	LD a,c	LD a,d	LD a,e	LD a,h	LD a,l	LD a,(hl)	LD a,a
8	ADC a,b	ADC a,c	ADC a,d	ADC a,e	ADC a,h	ADC a,l	ADC a,(hl)	ADC a,a
9	SBC a,b	SBC a,c	SBC a,d	SBC a,e	SBC a,h	SBC a,l	SBC a,(hl)	SBC a,a
A	XOR b	XOR c	XOR d	XOR e	XOR h	XOR l	XOR (hl)	XOR a
B	CP b	CP c	CP d	CP e	CP h	CP l	CP (hl)	CP a
C	RET z	RET	JP z,ttss	**Grupo CB**	CALL z,ttss	CALL ttss	ADC a,ss	RST 8
D	RET c	EXX	JP c,ttss	IN a,ss	CALL c,ttss	**Grupo DD**	SBC a,ss	RST 24
E	RET pe	JP (hl)	JP pe,ttss	EX de,hl	CALL pe,ttss	**Grupo ED**	XOR ss	RST 40
F	RET m	LD sp,hl	JP m,ttss	EI	CALL m,ttss	**Grupo FD**	CP ss	RST 56

Tabela 2.3: Opcodes simples, LSN (Least Significant Nibble) 8-F

2.3.1 Grupo CB

A partir do grupo CB existem opcodes não atribuídos, que em princípio não deveriam gerar instruções legais. Alguns autores testaram os opcodes "secretos" e descobriram um comportamento regular. Estas instruções não-oficiais estão marcadas com ⋆. O **fa** não assembla estas instruções.

	0	1	2	3	4	5	6	7
0	RLC b	RLC c	RLC d	RLC e	RLC h	RLC l	RLC (hl)	RLC a
1	RL b	RL c	RL d	RL e	RL h	RL l	RL (hl)	RL a
2	SLA b	SLA c	SLA d	SLA e	SLA h	SLA l	SLA (hl)	SLA a
3	⋆SLI b	⋆SLI c	⋆SLI d	⋆SLI e	⋆SLI h	⋆SLI l	⋆SLI (hl)	⋆SLI a
4	BIT 0,b	BIT 0,c	BIT 0,d	BIT 0,e	BIT 0,h	BIT 0,l	BIT 0,(hl)	BIT 0,a
5	BIT 2,b	BIT 2,c	BIT 2,d	BIT 2,e	BIT 2,h	BIT 2,l	BIT 2,(hl)	BIT 2,a
6	BIT 4,b	BIT 4,c	BIT 4,d	BIT 4,e	BIT 4,h	BIT 4,l	BIT 4,(hl)	BIT 4,a
7	BIT 6,b	BIT 6,c	BIT 6,d	BIT 6,e	BIT 6,h	BIT 6,l	BIT 6,(hl)	BIT 6,a
8	RES 0,b	RES 0,c	RES 0,d	RES 0,e	RES 0,h	RES 0,l	RES 0,(hl)	RES 0,a
9	RES 2,b	RES 2,c	RES 2,d	RES 2,e	RES 2,h	RES 2,l	RES 2,(hl)	RES 2,a
A	RES 4,b	RES 4,c	RES 4,d	RES 4,e	RES 4,h	RES 4,l	RES 4,(hl)	RES 4,a
B	RES 6,b	RES 6,c	RES 6,d	RES 6,e	RES 6,h	RES 6,l	RES 6,(hl)	RES 6,a
C	SET 0,b	SET 0,c	SET 0,d	SET 0,e	SET 0,h	SET 0,l	SET 0,(hl)	SET 0,a
D	SET 2,b	SET 2,c	SET 2,d	SET 2,e	SET 2,h	SET 2,l	SET 2,(hl)	SET 2,a
E	SET 4,b	SET 4,c	SET 4,d	SET 4,e	SET 4,h	SET 4,l	SET 4,(hl)	SET 4,a
F	SET 6,b	SET 6,c	SET 6,d	SET 6,e	SET 6,h	SET 6,l	SET 6,(hl)	SET 6,a

Tabela 2.4: Opcodes do grupo CB, LSN 0-7

	8	9	A	B	C	D	E	F
0	RRC b	RRC c	RRC d	RRC e	RRC h	RRC l	RRC (hl)	RRC a
1	RR b	RR c	RR d	RR e	RR h	RR l	RR (hl)	RR a
2	SRA b	SRA c	SRA d	SRA e	SRA h	SRA l	SRA (hl)	SRA a
3	SRL b	SRL c	SRL d	SRL e	SRL h	SRL l	SRL (hl)	SRL a
4	BIT 1,b	BIT 1,c	BIT 1,d	BIT 1,e	BIT 1,h	BIT 1,l	BIT 1,(hl)	BIT 1,a
5	BIT 3,b	BIT 3,c	BIT 3,d	BIT 3,e	BIT 3,h	BIT 3,l	BIT 3,(hl)	BIT 3,a
6	BIT 5,b	BIT 5,c	BIT 5,d	BIT 5,e	BIT 5,h	BIT 5,l	BIT 5,(hl)	BIT 5,a
7	BIT 7,b	BIT 7,c	BIT 7,d	BIT 7,e	BIT 7,h	BIT 7,l	BIT 7,(hl)	BIT 7,a
8	RES 1,b	RES 1,c	RES 1,d	RES 1,e	RES 1,h	RES 1,l	RES 1,(hl)	RES 1,a
9	RES 3,b	RES 3,c	RES 3,d	RES 3,e	RES 3,h	RES 3,l	RES 3,(hl)	RES 3,a
A	RES 5,b	RES 5,c	RES 5,d	RES 5,e	RES 5,h	RES 5,l	RES 5,(hl)	RES 5,a
B	RES 7,b	RES 7,c	RES 7,d	RES 7,e	RES 7,h	RES 7,l	RES 7,(hl)	RES 7,a
C	SET 1,b	SET 1,c	SET 1,d	SET 1,e	SET 1,h	SET 1,l	SET 1,(hl)	SET 1,a
D	SET 3,b	SET 3,c	SET 3,d	SET 3,e	SET 3,h	SET 3,l	SET 3,(hl)	SET 3,a
E	SET 5,b	SET 5,c	SET 5,d	SET 5,e	SET 5,h	SET 5,l	SET 5,(hl)	SET 5,a
F	SET 7,b	SET 7,c	SET 7,d	SET 7,e	SET 7,h	SET 7,l	SET 7,(hl)	SET 7,a

Tabela 2.5: Opcodes do grupo CB, LSN 8-F

2.3.2 Grupo ED

	0	1	2	3	4	5	6	7
4	IN b,(c)	OUT (c),b	SBC hl,bc	LD (qqtt),bc	NEG	RETN	IM 0	LD i,a
5	IN d,(c)	OUT (c),d	SBC hl,de	LD (qqtt),de			IM 1	LD a,i
6	IN h,(c)	OUT (c),h	SBC hl,hl					RRD
7			SBC hl,sp	LD (qqtt),sp				
A	LDI	CPI	INI	OUTI				
B	LDIR	CPIR	INIR	OTIR				

Tabela 2.6: Opcodes do grupo ED, LSN 0-7

	8	9	A	B	C	D	E	F
4	IN c,(c)	OUT (c),c	ADC hl,bc	LD bc,(qqtt)		RETI		LD r,a
5	IN e,(c)	OUT (c),e	ADC hl,de	LD de,(qqtt)			IM 2	LD a,r
6	IN l,(c)	OUT (c),l	ADC hl,hl					RLD
7	IN a,(c)	OUT (c),a	ADC hl,sp	LD sp,(qqtt)				
A	LDD	CPD	IND	OUTD				
B	LDDR	CPDR	INDR	OTDR				

Tabela 2.7: Opcodes do grupo ED, LSN 8-F

2.3.3 Grupos DD e FD

Os prefixos DD e FD substituem o par HL por IX (DD) e IY (FD) em diversas instruções. Listamos apenas a tabela do grupo DD, mas as instruções podem ter o par IX substituído por IY apenas trocando o prefixo DD por FD.

	0	1	2	3	4	5
2		LD ix, qqtt	LD (qqtt),ix	INC ix		
3					INC (ix+tt)	DEC (ix+tt)
7	LD (ix+tt),b	LD (ix+tt),c	LD (ix+tt),d	LD (ix+tt),e	LD (ix+tt),h	LD (ix+tt),l
E		POP ix		EX (sp),ix		PUSH ix

Tabela 2.8: Grupo DD, LSN (Least Significant Nibble) 0-5

	6	7	9	A	B	E
0			ADD ix,bc			
1			ADD ix,de			
2			ADD ix,ix	LD ix,(qqtt)	DEC ix	
3	LD (ix+tt), qq		ADD ix,sp			
4	LD b,(ix+tt)					LD c,(ix+tt)
5	LD d,(ix+tt)					LD e,(ix+tt)
6	LD h,(ix+tt)					LD l,(ix+tt)
7		LD (ix+tt),a				LD a,(ix+tt)
8	ADD a,(ix+tt)					ADC a,(ix+tt)
9	SUB a,(ix+tt)					SBC a,(ix+tt)
A	AND (ix+tt)					XOR a,(ix+tt)
B	OR (ix+tt)					CP (ix+tt)
C					Grupo CB	
E			JP (ix)			
F			LD sp,ix			

Tabela 2.9: Grupo DD, LSN (Least Significant Nibble) 6-E

Os grupos DD e FD podem ser combinados com todas as instruções com LSN 6 e E do grupo CB (Tabelas 2.4 e 2.5) tomando a forma de instruções de 4 bytes. Exemplo: `DD CB tt 46` é a instrução BIT 0,$(\%IX + tt)$.
A instrução não documentada SLI (Shift Left Inverted) opera de forma semelhante a SLA, porém insere 1 no bit mais baixo, em vez de 0. Os autores que pesquisaram os buracos nas tabelas de opcodes detectaram centenas de novas instruções, usando registradores esdrúxulos como HX e HY.

Capítulo 3

Diretivas

O **fa** aceita comandos especiais precedidos por um ponto (.) cujo objetivo não é gerar uma instrução válida do Z80, mas inserir dados ou modificar o estado do assembler. Neste capítulo listamos em ordem alfabética todas as diretivas aceitas.

.align *.align imm16*

SINTAXE: **.align** *num*

DESCRIÇÃO: Preenche com zeros todas as posições entre o endereço atual e o prómixo múltiplo do alinhamento. No exemplo abaixo a instrução *ld %hl, $40ff* é montada no endereço $c100.

EXEMPLO:

```
1   .org $c000
            ld %a, $3e
    .align $100
            ld %hl, $40ff
```

.append *.append nomearq*

SINTAXE: **.append** *nomearq*

DESCRIÇÃO: Assembla um arquivo de entrada ao final do conjunto de arquivos entrada atual. Útil para incluir bibliotecas de funções pré-definidas. Se o arquivo indicado já estiver na lista de arquivos a serem concatenados, este não é lido duas vezes, portanto não há problema em assemblar com *fa arq1.s arq2.s* mesmo que ambos *arq1.s* e *arq2.s* contenham *.append arq3.s*.
O arquivo é procurado: no diretório corrente e, caso o *nomearq* não seja um caminho absoluto (iniciado em /), é procurado também nos seguintes diretórios, nesta ordem: /include/asm , /usr/include/asm , /usr/local/include/asm , /foca/asm, /opt/foca/asm. *No futuro será permitido adicionar outros diretórios ao caminho de busca através de opções de linha de comando.*.

.da — *.da label*

SINTAXE: **.da** *identificador*

DESCRIÇÃO: Monta o valor de um label a partir da posição atual.

.db — *.db imm8 [imm8 ...]*

SINTAXE: **.db** *byte [byte ...]*

DESCRIÇÃO: Monta um ou mais bytes de dados a partir da posição atual. Os parâmetros podem estar em formato numérico (decimal, hexadecimal ou binário) ou como caractere entre apóstrofos.

EXEMPLO:

```
1   ; monta 11 bytes
    .db 'H' 'e' 'l' 'l' 'o' '\n' 1 4 $7e #00110110 'a'
```

.define — *.define label imm16*

SINTAXE: **.define** *label endereço*
SINTAXE: **.def** *label endereço*

DESCRIÇÃO: Declara um label com o endereço dado.

EXEMPLO:

```
1   .def INIT32 $6f
    .def CHPUT  $00a2

    call INIT32
5   ld %a, 'H';
    call CHPUT
```

.ds — *.ds string*

SINTAXE: **.ds** *string*

DESCRIÇÃO: Monta uma string ASCII de dados a partir do endereço atual. .ds não termina a string com delimitador algum. Para gerar strings terminadas em zero (como na linguagem C) use a diretiva *.dz*.

EXEMPLO:

```
1   .ds "Tudo que sobe\ntem que descer"
    .ds 'Tanto aspas como apostrofos sao aceitos para'
    .ds ' delimitar, mas voce precisa fechar a string'
    .ds " com o mesmo delimitador usado para abri-la"
```

.dw — *.dw imm16 [imm16 ...]*

SINTAXE: **.dw** *num [num ...]*

DESCRIÇÃO: Monta uma ou mais palavras de dados de 16 bits em formato little-endian (LSB no endereço mais baixo, MSB no endereço mais alto).

.dz — *.dz string*

SINTAXE: **.dz** *string*

DESCRIÇÃO: Monta uma string de dados ASCII a partir do endereço atual, e adiciona um byte 0 ao final, como usado pela linguagem C. Note que algumas rotinas CP/M esperam terminadores diferentes (um $, por exemplo) para strings.

.empty — *.empty imm16*

SINTAXE: **.empty** *num*

DESCRIÇÃO: Pula um número dado de bytes, preenchendo-os com zeros.

.endm — *.endm*

SINTAXE: **.endm**

DESCRIÇÃO: Termina a definição de uma macro iniciada com *.macro*.

.endr — *.endr*

SINTAXE: **.endr**

DESCRIÇÃO: Termina um bloco de repetição iniciado com *.repeat* ou *.rept*.

.forget — *.forget prefixo*

SINTAXE: **.forget** *prefixo_de_label*

DESCRIÇÃO: Torna labels (rótulos) inválidos a partir do ponto da diretiva, permitindo que os nomes de labels possam ser reutilizados. Todas os labels iniciados com o prefixo dado são afetados.

EXEMPLO:

```
1   inicio:  ld %de, $e000
             ld %hl, $d000
             ld %b, $60
```

```
    meuloop: ld %a, (%de)
 5           ld (%hl), %a
             inc %hl
             inc %de
    meufim:  djnz meuloop

10  .forget meu ; meuloop e meufim sao invalidas
                ; daqui em diante

    rotina2: ld %a, (%dhl)
             cp $55
15           jr z, meufim
             jr inicio ; a label inicio ainda eh valida
    meufim:  ret       ; eh permitido redefinir meufim pois
                       ; ela havia sido esquecida com .forget
```

.header

.header tipo

SINTAXE: **.header** *basic* | *msxdos* | *uzix* | *raw*

DESCRIÇÃO: Define o tipo de cabeçalho binário a ser gerado e, em alguns dos casos, a origem (*.origin*). Deve ser especificado antes de qualquer instrução ou diretiva que avance o contador de programa. Idealmente deve estar na primeira linha do arquivo ou logo abaixo das linhas de comentário iniciais.

basic Gera um arquivo binário carregável através do comando *BLOAD* do MSX-BASIC. O cabeçalho tem 7 bytes. Não ajusta a origem, que precisa estar após $8000 , de preferência próxima de $c000 para evitar sobreposições com programas BASIC e variáveis de sistema.

msxdos Gera um arquivo .COM compatível com MSX-DOS (CP/M). A origem é ajustada para $100, não há cabeçalho, o primeiro byte do arquivo já é a primeira instrução.

uzix Gera um binário compatível com o sistema operacional UZIX. A origem é ajustada para $110 e há um cabeçalho de 16 bytes, mas é recomendado usar o *stub* genérico para programas UZIX disponibilizado com o **fa** , que inicializa as variáveis do processe adequadamente e chama o rótulo *main*.

raw Apenas escreve as instruções no arquivo de saída sem qualquer cuidado especial. Semelhante ao modo *msxdos*, porém sem ajustar a origem.

.include

.include nomearq

SINTAXE: **.include** *nomearq*

■■ Descrição: Monta o arquivo de entrada indicado a partir do ponto atual, semelhante à diretiva #include da linguagem C.
O arquivo é procurado: no diretório corrente e, caso o *nomearq* não seja um caminho absoluto (iniciado em /), é procurado também nos seguintes diretórios, nesta ordem: /include/asm , /usr/include/asm , /usr/local/include/asm , /foca/asm, /opt/foca/asm. *No futuro será permitido adicionar outros diretórios ao caminho de busca através de opções de linha de comando.*.

.origin *.origin imm16*

■ Sintaxe: **.origin** *endereço*
Sintaxe: **.org** *endereço*

■■ Descrição: Ajusta a origem (endereço de montagem) para o endereço dado. Se utilizada em um ponto do arquivo onde algum dado ou instrução já tenha sido montado, preenche com zeros o espaço entre o contador de programa atual e a nova origem. Só é possível redefinir a origem para a "frente".

.macro *.macro ident*

■ Sintaxe: **.macro** *identificador*

■■ Descrição: Inicia a definição de uma macro. Macros são definidas internamente ao assembler e expandidas com a diretiva *.x*. Macros permitem que várias instruções sejam abreviadas com apenas uma linha.
Os nomes de macro são *case-sensitive*, ou seja, maiúsculas são diferentes de minúsculas. Os nomes de macro estão sujeitos às mesmas regras sintáticas dos nomes de labels, mas labels e macros estão em espaços diferentes: é permitido ter uma macro e uma label com o mesmo nome.
Toda e qualquer label definida dentro de uma definição de macro é local à macro e pode ser referenciada apenas dentro da própria macro. (Mas as macros podem usar labels definidas externamente sem problema, desde que a label esteja disponível no contexto em que a macro estiver sendo expandida.
Definições de macro são terminadas com a diretiva *endm*. Macros podem ter no máximo 1024 bytes (cada) em seu corpo, sendo que cada linha da macro tem os comentários removidos antes de ser armazenada.

■■■ Exemplo:

```
1   .macro inc4hl ; 4x inc %hl
      .repeat 4
        inc %hl
      .endr
5   .endm

    .macro pushall ; empilha todos os registradores
      push %af
      push %bc
10    push %de
      push %hl
```

```
      push %ix
      push %iy
    .endm

    .macro popall ; desfaz pushall
      pop %iy
      pop %ix
      pop %hl
      pop %de
      pop %bc
      pop %af
    .endm

    .def rotina_obscura $015b

          .x pushall          ; salva registradores
          call rotina_obscura
          .x popall           ; recupera

          ld %b, 64
          ld %a, $c0
    loop: ld (%hl), %a
          [inc4hl]   ; sintaxe alternativa para .x inc4hl
          dec %a
          djnz loop
          ret
```

.repeat *.repeat imm8*

SINTAXE: **.repeat** *num*
SINTAXE: **.rept** *num*

DESCRIÇÃO: Inicia um bloco de repetição. Note que a repetição é feita pelo assembler em tempo de montagem, e não pela CPU em tempo de execução do programa. O bloco é terminado pela diretiva *.endr*. Não é permitido aninhar blocos de repetição.

EXEMPLO:

```
.repeat 4
    inc %hl
.endr

; o codigo acima e o codigo abaixo geram as mesma instrucoes
; no arquivo de saida

inc %hl
inc %hl
inc %hl
inc %hl
```

.x *.x ident*

SINTAXE: **.x** *nome_de_macro*
SINTAXE: *[nome_de_macro]*

DESCRIÇÃO: Expande a macro indicada no ponto atual do programa. A sintaxe alternativa permite expandir macro escrevendo seu nome entre colchetes. Veja exemplo na descrição da diretiva *.macro*.

Capítulo 4

Programando para Uzix

Uzix é um sistema operacional para o MSX que implementa as funcionalidades do AT&T Unix Version 7. O Uzix é multi-tarefa, multi-usuário e oferece infra-estrutura de rede (TCP/IP). O **fa** oferece include files e cabeçalhos que facilitam a programação em ambiente Uzix.
Este capítulo toma como base o Uzix 0.2.0, mas notas a respeito de versões anteriores serão feitas quando necessário.

4.1 Multi-Tarefa no Uzix

O Uzix implementa multi-tarefa preemptiva, mas a arquitetura MSX não possui um modo protegido, e mesmo que tivesse, acessar o hardware via chamadas de sistema seria excessivamente lento para periféricos como o VDP. Portanto, o funcionamento correto da multi-tarefa depende dos processos não se sabotarem nem sabotarem o sistema.
A multi-tarefa preemptiva consiste em manter uma tabela com os dados dos processos (em que bloco(s) da mapper eles estão, seus descritores de arquivo abertos, o estado dos registradores quando o processo "perdeu" a CPU pela última vez) e dar a CPU a um processo por vez por um determinado tempo (este tempo é denominado *quantum*. Quando o tempo de CPU do processo acaba, o kernel do Uzix (através do seu manipulador de interrupções) salva o estado do processo atual (todo o banco de registradores e o PC), chaveia a expansão de memória[1] para mapear o próximo processo, restaura os registradores do processo que está recebendo a CPU e transfere o controle a ele. Toda multi-tarefa é baseada na idéia de chavear a CPU entre vários processos rapidamente, criando a ilusão de que todos os processos "executam ao mesmo tempo".
"Preemptiva" significa que o kernel decide quando o processo já "usou demais a CPU", e não o processo (isto seria multi-tarefa cooperativa). Este comportamento (preemptivo) apenas é possível no Uzix porque o manipulador de interrupções do kernel está sempre sendo chamado de forma transparente e pode "contar" há quanto tempo o processo atual detem a posse da CPU.
Os problemas são:

- Se o processo desabilitar as interrupções (instrução DI), o kernel nunca vai tomar a CPU dele, e a multi-tarefa vai para o vinagre; Não execute DI nos seus programas.
- Se o processo acessa o hardware diretamente (realizando INs e OUTs diretamente ou mudando o modo de vídeo via BIOS, por exemplo), o kernel não tem como detectar os acessos e "reverter" o estado ao chavear o processo. Com isso, um processo que use o modo gráfico rodando concomitantemente com um processo que use o vídeo em modo texto causará desordem, pois poderá acessar o

[1] Em geral a Memory Mapper, mas houve uma versão do Uzix 1 que usava a MegaRAM como banco de processos.

vídeo pensando que está em modo gráfico quando está em modo texto e vice-versa para o processo em modo texto. A única forma segura de garantir inter-operabilidade é usar apenas as chamadas de sistema do Uzix para realizar I/O, mas o Uzix não provê drivers do som e suas funções de vídeo se resumem a escrever bytes no terminal;

- Como os processos não estão impedidos de realizar INs, OUTs, ler memória fora de seu espaço de endereçamento, etc., eles podem alterar porções da mapper que contêm dados do Uzix ou de outros processos, corrompendo o sistema.

Os problemas acima não são falhas do Uzix, são consequências diretas da inexistência de um modo protegido no MSX. Não há como resolver esses problemas sem a adição de hardware especializado. É possível amenizar o problema provendo drivers (chamadas de sistema ou módulos padronizados) para acesso aos periféricos, porém esta solução é lenta, consome memória e consome tempo do desenvolvedor do sistema, portanto a "solução" resolve um problema e cria três problemas novos.

4.2 Modelo de memória

A Figura 4.1 mostra o layout de memória de um processo no Uzix. Toda vez que um processo ganha um quantum de CPU, sua imagem é mapeada a partir do endereço $0000.

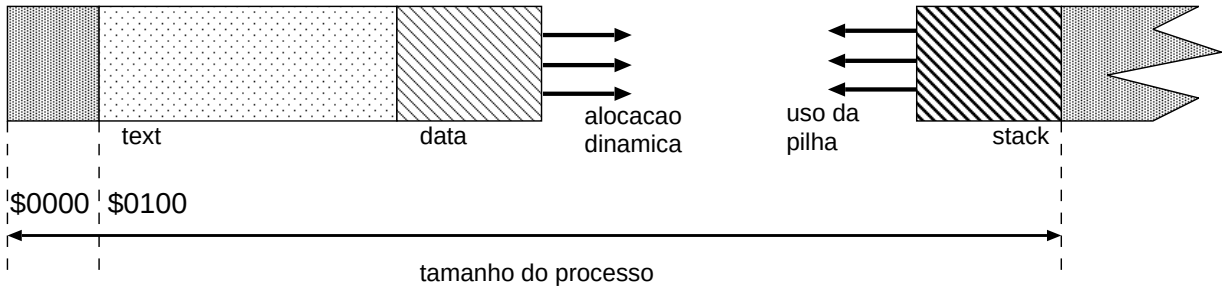


Figura 4.1: Mapa de Memória de um Processo

A região $0000-$00FF contém dados de controle do processo, que não devem ser acesados diretamente pela aplicação.
O segmento *text* contém as instruções do processo. Não há proteção de memória no MSX, portanto o processo pode se auto-modificar, embora isso seja extremamente não-recomendado.
Os 16 bytes entre $0100-$010f formam o cabeçalho do processo. A instrução em $0100 precisa ser um JP $0110 (c3 10 01) para que o Uzix reconheça o arquivo como executável. Os 13 bytes restantes contêm flags e são preenchidos com ponteiros para variáveis de ambiente e parâmetros de linha de comando. Assim, o programa "de verdade" começa em $0110. Alguma inicialização é recomendada (o **fa** oferece um cabeçalho padrão em arquivos de stub, como veremos à frente).
A região de dados compreende o heap e a pilha. O heap cresce à medida em que o processo realiza chamadas *brk* ao sistema. Em programas C essas chamadas são materializadas pela família *malloc()*. A pilha cresce à medida em que vai sendo usada. Uma operação que tipicamente consome pilha é uma chamada recursiva, em que vários contextos são salvos na pilha. Se a pilha e a região de dados se encontrarem, o resultado é desastroso. Como o MSX não possui segmentos de memória protegidos, não é possível isolar os segmentos de heap e pilha.
A comprimento de $0000 ao fundo da pilha é o comprimento do processo. Até o Uzix 1.0 todos os processos recebiam 32 KB. A partir do Uzix 0.2.0 (versão beta que resultará no Uzix 2.0) Os processos podem receber

"slots" de 16, 32 ou 48 KB de memória. Dois bits no cabeçalho do executável indicam ao Uzix o tipo de slot que deve ser alocado. O processo pode requerer um slot de 16KB, 32KB, 48KB ou "automático". No modo automático é alocado o menor slot que comporte o processo. O processo não pode "crescer" além do slot que é alocado no momento de sua execução.

4.3 Exemplo: Oi Mundo!

Nesta seção apresentamos um programa-exemplo extremamente original: ele escreve uma mensagem no console (saída-padrão) e sai com código 0 (sucesso).

```
1
    .header uzix
    .include uzix.stub

5   _main:                          ;  o stub chama _main
            ld %hl, 10
            push %hl                ;  comprimento da string
            ld %bc,string
            push %bc                ;  endereco
10          ld %de,1
            push %de                ;  descritor 1 (stdout)
            ld %hl,36               ;
            push %hl                ;  syscall 36 (write)
            call 8                  ;  chama write(1,string,13)
15          pop %hl                 ;  desempilha syscall#
            pop %de                 ;  desempilha valor de retorno
            pop %bc
            pop %hl
            ret                     ;  retorna ao stub
20
    string: .dz "Oi Mundo!\n"
```

A diretiva *.header* diz ao **fa** para gerar um executável no formato Uzix. O arquivo de include *uzix.stub*, que será listado na próxima seção, é parte do **fa** e implementa uma inicialização padrão do processo, semelhante àquela necessária para programas na linguagem C. Tudo o que precisamos saber sobre o stub no momento é que ele chamará o label *_main* com uma instrução CALL e ao receber o controle de volta sairá com o valor do registrador DE como status.
Este programa apenas empilha os parâmetros para a chamada de sistema *write(int descritor, char *text, int comprimento)* (na ordem inversa, pois o Uzix segue a convenção de chamada do C), empilha o número da system call indicando que é um write (36) e chama a rotina no endereço $8, que é o ponto de entrada das chamadas de sistema do Uzix. O programa é responsável por desempilhar os parâmetros.

4.4 Stub

O **fa** inclui um stub para o Uzix, com uma inicialização padrão do processo. O stub é um arquivo assembly do **fa** , e está listado abaixo.

```
1   .define e_flags  $103
```

```
.define e_text   $104
.define e_data   $106
.define e_bss    $108
.define e_heap   $10a
.define e_stack  $10c
.define e_env    $10e

        ld (___stktop), %sp
; Clear BSS
        ld      %hl, (e_data)
        ex      %de, %hl
        ld      %hl, (e_bss)
        or      %a              ; CLC
        sbc     %hl, %de
        ld      %c,  %l
        ld      %b,  %h
        dec     %bc             ; BC = counter-1
        ld      %l,  %e
        ld      %h,  %d         ; HL = e_data
        inc     %de             ; DE = e_data+1
        ld      (%hl), 0
        ldir                    ; clear bss - always >= 10 bytes
        pop     %bc             ; drop retaddr
; now there is the next stack structure:
;       +4 envp
;       +2 argv
; %sp-> +0 argc
        ld      %ix, 0
        add     %ix, %sp
        ld      %l, (%ix+4)
        ld      %h, (%ix+5)
        ld      (_environ), %hl
        ld      %l, (%ix+2)
        ld      %h, (%ix+3)
        ld      (__argv), %hl
        ld      %l, (%ix+0)
        ld      %h, (%ix+1)
        ld      (__argc), %hl
_start1:
        call    _main
        pop     %bc
        pop     %bc
        pop     %bc
        ex      %de, %hl        ; exit arg in DE
_exit:
        push    %de
        ld      %hl, (___cleanup)
        ld      %a, %l
        or      %h
        call    nz, _indirect   ; (*__cleanup)(exitcode, ???)
```

```
            pop     %de
            jr      __exit          ; to kernel - arg in DE

_indirect:
            jp      (%hl)

__exit:
            ld      %hl, 11
            push    %de
            push    %hl
            call    8

; area de dados

etext:

;       variaveis de gerencia de memoria

___heapbase:    .da     ebss
___brklvl:      .da     ebss
___heaptop:     .da     ebss
___stktop:      .dw     0

edata:
__argc:         .empty 2
__argv:         .empty 2
_environ:       .empty 2
_errno:         .empty 2
___cleanup:     .empty 2

ebss:
```

4.5 Chamadas de Sistema

O Uzix implementa todas as chamadas de sistema do AT&T Unix 7, embora algumas variantes de *exec()* sejam providas apenas pela biblioteca C.
Uma chamada de sistema é realizada empilhando os parâmetros na ordem inversa da declaração, empilhando o número da chamada e então fazendo um CALL 8. É responsabilidade da aplicação desempilhar os parâmetros após o CALL. O valor de retorno, de 16 bits, é colocado no registrador DE. A única exceção é a chamada *lseek*, cujo valor de retorno é de 32 bits e é colocado em HL:DE (HL é a palavra mais significativa).
A tabela abaixo lista as chamadas diretas, seus parâmetros e número de chamada.

Tabela 4.1: Chamadas Diretas de Sistema.

#	args	syscall	Protótipo C
0	2	access	`int access(char *path, int mode)`
1	1	alarm	`int alarm(int secs)`

⟶

Tabela 4.1: (cont.)

#	args	syscall	Protótipo C
2	1	brk	`int brk(char *addr)`
3	1	chdir	`int chdir(char *path)`
4	2	chmod	`int chmod(char *path, int mode)`
5	3	chown	`int chown(char *path, int owner, int group)`
6	1	close	`int close(int fd)`
7	N	getset	`int getset(int operacao, ...)`
8	1	dup	`int dup(int oldd)`
9	2	dup2	`int dup2(int oldd, int newd)`
10	3	execve	`int execve(char *name, char **argv, char **envp)`
11	1	exit	`int exit(int status)`
12	0	fork	`int fork(void)`
13	2	fstat	`int fstat(int fd, void *buf)`
14	2	getfsys	`int getfsys(int dev, void *buf)`
15	N	ioctl	`int ioctl(int fd, int req, ...)`
16	2	kill	`int kill(int pid, int sig)`
17	2	link	`int link(char *oldname, char *newname)`
18	3	mknod	`int mknod(char *name, int mode, int dev)`
19	3	mount	`int mount(char *spec, char *dir, int rwflag)`
20	3	open	`int open(char *name, int flags, int mode)`
21	0	pause	`int pause(void)`
22	1	pipe	`int pipe(int *fd)`
23	3	read	`int read(int fd, void *buf, int bytes)`
24	1	sbrk	`int sbrk(int incr)`
25	3	lseek	`long lseek(int fd, long offset, int flag)`
26	2	signal	`int signal(char sig_num, void (*func)(int) )`
27	2	stat	`int stat(char *path, void *buf)`
28	1	stime	`int stime(int *tvec)`
29	0	sync	`int sync(void)`
30	1	time	`void time(int *t)`
31	1	times	`int times(struct tms *tvec)`
32	1	umount	`int umount(char *spec)`
33	1	unlink	`int unlink(char *path)`
34	2	utime	`int utime(char *path, struct utimbuf *buf)`
35	3	waitpid	`int waitpid(int pid, int *statloc, int options)`
36	3	write	`int write(int fd, void *buf, int nbytes)`
37	2	reboot	`int reboot(char p1, char p2)`
38	2	symlink	`int symlink(char *oldname, char *newname)`
39	1	chroot	`int chroot(char *path)`
40	2	mod_reg	`int mod_reg(int sig, int (*func)())`
41	1	mod_dereg	`int mod_dereg(int sig)`
42	4	mod_call	`int mod_call(int sig, int fnc, char *args, int argsz)`
43	4	mod_sendreply	`int mod_sendreply(int pid, int fnc, char *r, int rsz)`
44	3	mod_reply	`int mod_reply(int sig, int fcn, char *r)`

Tabela 4.1: Chamadas Diretas de Sistema

As chamadas relativas a módulos (mod_*) permitem a extensão do kernel do Uzix com novas funcionalidades. A mais comum é o módulo TCP/IP que provê funções de rede e conexão com a Internet.
A chamada *creat* é realizada como uma chamada para *open*:

Chamada	**Proxy**
`creat(char *path, int mode)`	`open(path, 0x301, mode)`[2]

Tabela 4.2: Chamadas Indiretas de Sistema

As chamadas restantes são aquelas que apenas lêm ou alteram o valor de uma variável do sistema, como *getpid()*. Estas são implementadas através da chamada 7, *getset*. A tabela seguinte lista as chamadas baseadas em *getset*[3].

Chamada	**Protótipo C**	**#**	**Proxy**
getpid	`getpid(void)`	0	`getset(0)`
getppid	`getppid(void)`	1	`getset(1)`
getuid	`getuid(void)`	2	`getset(2)`
setuid	`setuid(int uid)`	3	`getset(3,uid)`
geteuid	`geteuid(void)`	4	`getset(4)`
getgid	`getgid(void)`	5	`getset(5)`
setgid	`setgid(int gid)`	6	`getset(6,gid)`
getegid	`getegid(void)`	7	`getset(7)`
getprio	`getprio(void)`	8	`getset(8)`
setprio	`setprio(int pid, char prio)`	9	`getset(9,pid,prio)`
umask	`umask(int mask)`	10	`getset(10,mask)`
systrace	`systrace(int onoff)`	11	`getset(11,onoff)`

Tabela 4.3: Chamadas via *getset*

4.6 Módulos

O Uzix permite a extensão de serviços do sistema operacional através de módulos que rodam como processos normais. Um módulo é um conjunto de funções, que recebem uma sequência de no máximo 512 bytes como entrada e devolvem uma sequência também limitada a 512 bytes como resposta. Chamamos de *requisição* uma chamada de função de um módulo, e de *reply* a resposta de uma requisição.
Processos podem se registrar e desregistrar como módulos com as chamadas `mod_reg` e `mod_dereg`. Em `mod_reg` é passada a assinatura (sig) do módulo, que o identifica dentre quaisquer outros módulos, e um ponteiro para uma função de callback, que é chamada pelo kernel do Uzix quando há uma requisição para o módulo. Esta função de callback deve usar `mod_sendreply` para enviar a resposta à chamada (ver Figuras 4.2 e 4.3). A função de callback deve ser escrita de acordo com o protótipo C abaixo:

```
int callback(int fcn, int pid, char *data, int datasize);
```

[2] $0x301$ = O_CREAT|O_TRUNC|O_WRONLY
[3] os tipos de retorno, omitidos, são sempre int.

A função deve retornar 0 em caso de sucesso no enfileiramento (ou processamento imediato) da requisição ou os códigos de erro EINVFUNC ou ENOMEM em caso de falha (os valores numéricos dos códigos de erro estão listados no final deste capítulo).

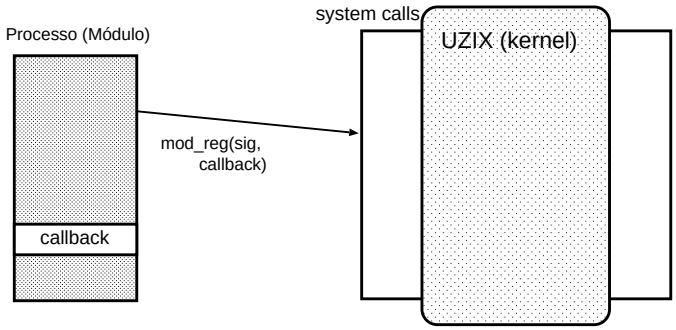


Figura 4.2: Registro de Módulo

Para *usar* um módulo, um processo deve realizar chamadas de funções com a chamada `mod_call` (que tem a assinatura do módulo, número da função dentro do módulo e os dados da chamada, que podem ter até 512 bytes de comprimento). Esta chamada não é bloqueante (retorna imediatamente ainda que a operação não tenha sido completada). Para verificar o resultado da operação, o processo deve chamar `mod_reply` até que esta retorne 0 (sucesso). Tanto na chamada como na resposta, parâmetros passados por referência (ponteiros) são linearizados (copiados) no buffer de mod_call antes da chamada.

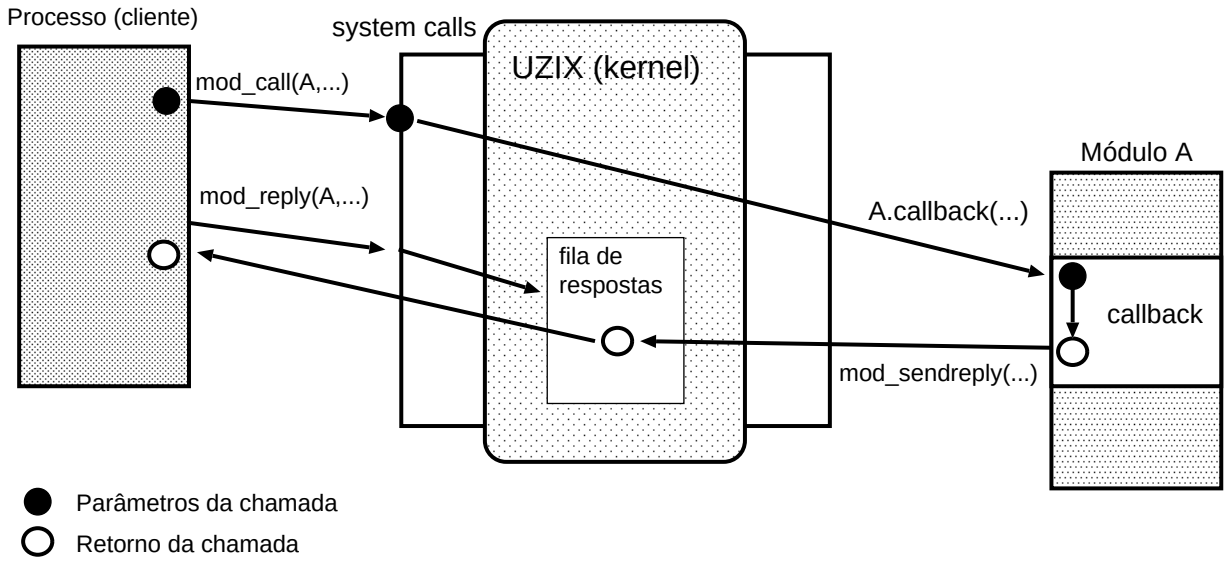


Figura 4.3: Chamada de função em módulo. A chamada mod_call retorna tão logo o callback do módulo retorne indicando a aceitação ou não da requisição. Quando o termina de processar a requisição, o módulo enfileira a resposta no kernel com a chamada mod_sendreply. O processo requisitante precisa perguntar ao kernel se já há resposta disponível com a chamada mod_reply até que a resposta esteja disponível.

As requisições e os replies são implementados com entrega imediata de requisições e uma fila interna (no

kernel) para replies. Como a maioria das chamadas de sistema, as chamadas `mod_*` retornam 0 em caso de sucesso, < 0 em caso de erro, ajustando `errno` com a identificação do erro ocorrido. O módulo pode recusar uma requisição (por não ter memória para armazená-la, por exemplo), neste caso `mod_call` retornará um valor negativo.

4.7 Módulo TCP/IP

O módulo TCP/IP implementa um subconjunto de IPv4 e permite que o Uzix se comunique com outros sistemas que suportem o protocolo.
A assinatura do módulo TCP/IP é $4950, e ele provê as funções listadas na Tabela 4.4.

Tabela 4.4: Chamadas do Módulo TCP/IP.

Chamada	**Protótipo C**	**FNC#**
ipconnect	`int ipconnect(char mode, ip_struct_t *ipstruct)`	1
ipgetc	`int ipgetc(uchar socknum)`	2
ipputc	`int ipputc(uchar socknum, uchar byte)`	3
ipwrite	`int ipwrite(uchar socknum, uchar *bytes, int len)`	4
ipread	`int ipread(uchar socknum, uchar *bytes, int len)`	5
ipclose	`int ipclose(uchar socknum)`	6
iplisten	`int iplisten(int aport, uchar protocol)`	7
ipaccept	`int ipaccept(ip_struct_t *ipstruct, int aport, uchar block)`	8
ping	`int ping(uchar *IP, unsigned long *unused, uint len)`	9
setsocktimeout	`int setsocktimeout(uchar socknum, uint timeout)`	10
ipunlisten	`int ipunlisten(int aport)`	11
ipgetpingreply	`icmpdata_t *ipgetpingreply(void)`	12
gettcpinfo	`tcpinfo_t *gettcpinfo(void)`	13
getsockinfo	`sockinfo_t *getsockinfo(uchar socknum)`	14

Tabela 4.4: Chamadas do Módulo TCP/IP.

Os tipos de dados usados são:

```
// numeros de protocolo (protocolo para iplisten)
ICMP_PROTOCOL = 1
TCP_PROTOCOL  = 6
UDP_PROTOCOL  = 17

// modos de abertura
TCP_ACTIVE_OPEN  = 255
TCP_PASSIVE_OPEN = 0

// protocolos (mode do ipconnect)
IPV4_TCP  = 1
IPV4_UDP  = 2
IPV4_ICMP = 3
```

```
15  // modos UDP
    UDPMODE_ASC   = 1
    UDPMODE_CKSUM = 2

    // codigos de erro
20  ECONTIMEOUT = $80
    ECONREFUSED = $81
    ENOPERM     = $82
    ENOPORT     = $83
    ENOROUTE    = $84
25  ENOSOCK     = $85
    ENOTIMP     = $86
    EPROT       = $87
    EPORTINUSE  = $88

30  // estados permitidos para sockstatus em sockinfo_t
    TCP_CLOSED       = $00
    TCP_LISTEN       = $01
    TCP_SYN_SENT     = $42
    TCP_SYN_RECEIVED = $43
35  TCP_ESTABLISHED  = $c4
    TCP_FIN_WAIT1    = $45
    TCP_FIN_WAIT2    = $46
    TCP_CLOSE_WAIT   = $87
    TCP_CLOSING      = $08
40  TCP_LAST_ACK     = $09
    TCP_TIMEWAIT     = $0a
    UDP_LISTEN       = $91
    UDP_ESTABLISHED  = $94

45  ip_struct_t = { uchar remote_ip[4],
                    uint remote_port,
                    uint local_port }

    icmpdata_t  = { uchar type,
50                  uchar icmpcode,
                    unsigned long unused,
                    uchar data[28],  /* pad para 64 bytes */
                    uint len;
                    uchar sourceIP[4],
55                  uchar ttl }

    tcpinfo_t   = { uchar IP[4],
                    uchar dns1ip[4],
                    uchar dns2ip[4],
60                  char datalink[5],
                    char domainname[DOMSIZE=128],
                    int used_sockets,
                    int avail_sockets,
                    int used_buffers,
```

```
65                  int avail_buffers,
                    int IP_chksum_errors }

    sockinfo_t  = { int localport,
                    int remoteport,
70                  uchar remote_ip[4],
                    char socketstatus, /* bit 7: permissao de escrita
                                          bit 6: estado de listen
                                          bits 3-0: estado */
                    char sockettype,   /* TCP = 1, UDP = 2 */
75                  char sockerr,      /* codigo de erro */
                    int pid }
```

4.7.1 Estabelecimento de conexão

Para estabelecer uma conexão é utilizada a chamada ipconnect. O conceito de conexão aqui se refere à criação de estruturas internas no kernel (semelhante à criação de sockets em sistemas Unix de maior porte, como BSD e Linux), portanto é necessário chamar ipconnect mesmo para transmitir pacotes UDP. ipconnect retorna o identificador da conexão.

4.7.2 Envio e recebimento de dados

Para enviar e receber dados são usadas as chamadas ipread, ipwrite, ipgetc e ipputc. A conexão deve ser fechada com ipclose. Todas as conexões de um processo são fechadas quando o processo termina.

4.7.3 Esperando uma conexão ou pacotes UDP

Para esperar uma conexão, deve-se chamar iplisten (equivalente ao *bind* de outros Unixes) para associar uma porta ao processo. O processo então deve aguardar uma conexão chamando ipaccept, que retornará o identificador a ser usado para ler e escrever da conexão TCP ou do pacote UDP. A chamada ipunlisten permite desassociar a porta do processo.

4.7.4 Ping

A operação de ping (protocolo ICMP) é realizada pela chamada ping, e o pacote de eco é recebido pela chamada ipgetpingreply (que retorna 0 caso tenha sucesso).

4.7.5 Outras Operações

As demais operações configuram o timeout para operações (setsocktimeout) e obtêm informações sobre conexões e sobre o módulo TCP/IP (getsockinfo e gettcpinfo).

Ainda existem duas chamadas declaradas com números de função 15 e 16, *rawclose* e *shutdown*, mas elas não estão declaradas nos cabeçalhos para usuários da biblioteca C e são chamadas não documentadas.

As operações de resolução de nomes (DNS) não estão implementadas no módulo TCP/IP, e estão disponíveis apenas em biblioteca C, ligada estaticamente com os programas que a utilizam, portanto não é possível utilizá-la em programas assembly (pelo menos não de forma fácil).

4.8 Códigos de erro

As chamadas de sistema do Uzix retornam um valor ≥ 0 em caso de sucesso e < 0 em caso de erro, e o código de erro é colocado na variável global (definida no stub dos programas Uzix) **errno**. Abaixo estão relacionados os possíveis códigos de erro.

```
 1  EPERM           1               /* 1 Operation not permitted */
    ENOENT          2               /* 2 No such file or directory */
    ESRCH           3       /*-*/   /* 3 No such process */
    EINTR           4               /* 4 Interrupted system call */
 5  EIO             5               /* 5 I/O error */
    ENXIO           6               /* 6 No such device or address */
    E2BIG           7               /* 7 Arg list too long */
    ENOEXEC         8               /* 8 Exec format error */
    EBADF           9               /* 9 Bad file number */
10  ECHILD          10              /* 10 No child processes */
    EAGAIN          11              /* 11 Try again */
    ENOMEM          12              /* 12 Out of memory */
    EACCES          13              /* 13 Permission denied */
    EFAULT          14              /* 14 Bad address */
15  ENOTBLK         15              /* 15 Block device required */
    EBUSY           16              /* 16 Device or resource busy */
    EEXIST          17              /* 17 File exists */
    EXDEV           18              /* 18 Cross-device link */
    ENODEV          19              /* 19 No such device */
20  ENOTDIR         20              /* 20 Not a directory */
    EISDIR          21              /* 21 Is a directory */
    EINVAL          22              /* 22 Invalid argument */
    ENFILE          23              /* 23 File table overflow */
    EMFILE          24      /*-*/   /* 24 Too many open files */
25  ENOTTY          25      /*-*/   /* 25 Not a typewriter */
    ETXTBSY         26      /*-*/   /* 26 Text file busy */
    EFBIG           27      /*-*/   /* 27 File too large */
    ENOSPC          28              /* 28 No space left on device */
    ESPIPE          29              /* 29 Illegal seek */
30  EROFS           30      /*-*/   /* 30 Read-only file system */
    EMLINK          31      /*-*/   /* 31 Too many links */
    EPIPE           32              /* 32 Broken pipe */
    EDOM            33      /*-*/   /* 33 Math argument out of domain of func */
    ERANGE          34      /*-*/   /* 34 Math result not representable */
35  EDEADLK         35      /*-*/   /* 35 Resource deadlock would occur */
    ENAMETOOLONG    36      /*-*/   /* 36 File name too long */
    ENOLCK          37      /*-*/   /* 37 No record locks available */
    EINVFNC         38              /* 38 Function not implemented */
    ENOTEMPTY       39      /*-*/   /* 39 Directory not empty */
40  ELOOP           40      /*-*/   /* 40 Too many symbolic links encountered */
    ESHELL          41      /*-*/   /* 41 It's a shell script */
    ENOSYS          EINVFNC
```

Capítulo 5

MSX-BIOS

5.1 Rotinas MSX-BIOS

A ROM do MSX é mapeada nos primeiros 32K de memória ($0-$7FFF) e inclui a BIOS MSX e o MSX BASIC. Neste capítulo documentamos as principais rotinas da BIOS. Mesmo quando a BIOS não está mapeada (dentro do MSX-DOS, por exemplo), é possível chamar suas rotinas através de uma chamada CALSLT.
Nos restringimos às rotinas úteis em programas em Assembly, omitindo aquelas interdependentes do interpretador BASIC.
A tabela resumida é ordenada por endereço, e a lista descritiva é ordenada pelo nome padronizado da rotina. Lembre-se: nunca traduza slot como conector, isso dói!
Para chamar uma rotina basta carregar os parâmetros nos registradores e fazer um CALL para o endereço da rotina. A única exceção é CALLF ($0030) que usa um método diferente para passagem de parâmetros.
Os dados deste capítulo foram obtidos de [Ava88].

Tabela 5.1: Rotinas MSX-BIOS, por endereço de chamada.

Endereço	**Rotina**	**Resumo**
$0000	CHKRAM	Partida (boot)
$000C	RDSLT	Lê dado de qualquer slot.
$0014	WRSLT	Escreve dado em qualquer slot.
$001C	CALSLT	Chama rotina em qualquer slot.
$0020	DCOMPR	Compara DE e HL.
$0024	ENASLT	Mapeia slot.
$0030	CALLF	Chama rotina em qualquer slot (inline).
$0038	KEYINT	Manipulador de interrupção padrão.
$003B	INITIO	Inicializa PSG e porta de impressora.
$003E	INIFNK	Inicializa strings das teclas de função.
$0041	DISSCR	Desativa tela.
$0044	ENASCR	Ativa tela.
$0047	WRTVDP	Escreve em registrador do VDP.
$004A	RDVRM	Lê byte da VRAM.
$004D	WRTVRM	Escreve byte na VRAM.
$0050	SETRD	Prepara VDP para leitura.
$0053	SETWRT	Prepara VDP para escrita.

⟶

Tabela 5.1: (cont.)

Endereço	**Rotina**	**Resumo**
$0056	FILVRM	Preenche bloco da VRAM.
$0059	LDIRMV	Copia bloco da VRAM para a RAM.
$005C	LDIRVM	Copia bloco da RAM para a VRAM.
$005F	CHGMOD	Altera modo do VDP.
$0062	CHGCLR	Altera cores do VDP.
$0066	NMI	Manipulador da NMI.
$0069	CLRSPR	Limpa todos os sprites.
$006C	INITXT	Inicializa VDP em modo texto 40x24.
$006F	INIT32	Inicializa VDP em modo texto 32x24.
$0072	INIGRP	Inicializa VDP em modo gráfico 256x192.
$0075	INIMLT	Inicializa VDP em modo multicolorido 64x48.
$0084	CALPAT	Calcula endereço da imagem do sprite.
$0087	CALATR	Calcula endereço do atributo do sprite.
$008A	GSPSIZ	Obtém tamanho do sprite.
$008D	GRPPRT	Escreve caractere na tela gráfica.
$0090	GICINI	Inicializa PSG.
$0093	WRTPSG	Escreve em registrador do PSG.
$0096	RDPSG	Lê registrador do PSG.
$0099	STRTMS	Desempilha fila musical.
$009C	CHSNS	Verifica buffer do teclado.
$009F	CHGET	Obtém caractere do buffer do teclado.
$00A2	CHPUT	Escreve caractere na tela.
$00A5	LPTOUT	Imprime caractere na porta de impressora.
$00A8	LPTSTT	Teste de status da impressora.
$00AE	PINLIN	Lê uma linha do console.
$00B1	INLIN	Lê uma linha do console.
$00B4	QINLIN	Lê uma linha do console.
$00B7	BREAKX	Verifica Ctrl+Stop.
$00C0	BEEP	Emite beep.
$00C3	CLS	Limpa tela.
$00C6	POSIT	Posiciona cursor.
$00CC	ERAFNK	Apaga a linha das teclas de função.
$00CF	DSPFNK	Mostra a linha das teclas de função.
$00D2	TOTEXT	Retorna VDP ao modo texto.
$00D5	GTSTCK	Lê status do joystick.
$00D8	GTTRIG	Lê status do botão do joystick.
$00DB	GTPAD	Lê status do tablet.
$00DE	GTPDL	Lê status do paddle.
$00E1	TAPION	Aciona entrada de fita.
$00E4	TAPIN	Lê entrada de fita.
$00E7	TAPIOF	Desliga entrada de fita.
$00EA	TAPOON	Aciona saída de fita.
$00ED	TAPOUT	Escreve na saída de fita.
$00F0	TAPOOF	Desliga saída de fita.
$00F3	STMOTR	Controla motor da unidade de fita.
$00F6	LFTQ	Verifica espaço em fila musical.

⟶

Tabela 5.1: (cont.)

Endereço	Rotina	Resumo
$00F9	PUTQ	Coloca byte em fila musical.
$00FC	RIGHTC	Move endereço de pixel à direita.
$00FF	LEFTC	Move endereço de pixel à esquerda.
$0102	UPC	Move endereço de pixel acima.
$0105	TUPC	Testa e move endereço de pixel acima.
$0108	DOWNC	Move endereço de pixel abaixo.
$010B	TDOWNC	Testa e Move endereço de pixel abaixo.
$010E	SCALXY	"Clipa" coordenadas gráficas.
$0111	MAPXYC	Converte coordenadas do modo gráfico.
$0114	FETCHC	Obtém endereço físico do pixel atual.
$0117	STOREC	Armazena endereço físico do pixel atual.
$011A	SETATR	Muda cor de desenho.
$011D	READC	Lê atributo do pixel atual.
$0120	SETC	Muda atributo do pixel atual.
$0123	NSETCX	Muda atributo de uma sequência de pixels.
$0132	CHGCAP	Altera LED do CAPS LOCK.
$0135	CHGSND	Altera o estado do click do teclado.
$0138	RSLREG	Lê registrador do slot primário.
$013B	WSLREG	Escreve registrador do slot primário.
$013E	RDVDP	Lê registrador de status do VDP.
$0141	SNSMAT	Lê linha da matriz de teclado.
$0156	KILBUF	Limpa buffer do teclado.
$0159	CALBAS	Chama rotina BASIC a partir de qualquer slot.

Tabela 5.1: Rotinas MSX-BIOS, por endereço de chamada.

5.2 Listagem Alfabética Descritiva

Na listagem a seguir são usados os seguintes símbolos:

◇← Parâmetros de entrada.

◇→ Parâmetros de saída.

⊗⇝⊕ Registradores e estados modificados. EI indica que as interrupções são habilitadas.

BEEP *$00C0*

■ Emite um beep de 1316 Hz.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, BC, E, EI.

BREAKX *$00B7*

■ Verifica estado da combinação de teclas Ctrl+Stop.

◇← Nenhum.
◇→ Flag C=1 se Ctrl+Stop estiver pressionada.
⊗⇝⊕ AF.

CALATR *$0087*

■ Calcula o endereço de um bloco de atributos de sprite, que é $ATRBAS + 4 \times numsprite$, com o número do sprite entre 0 e 31.

◇← A: número do sprite.
◇→ HL: endereço do atributo do sprite.
⊗⇝⊕ AF, DE, HL.

CALBAS *$0159*

■ Chama um endereço no interpretador BASIC mesmo que este não esteja mapeado, apenas carrega IY com o valor correto de slot e chama CALSLT para realizar a chamada.

◇← IX=endereço a chamar.
◇→ Nenhum.
⊗⇝⊕ AF', BC', DE', HL', IY, DI.

CALLF *$0030*

■ Chama um endereço em qualquer slot. Tem a peculiaridade de usar parâmetros inline, para permitir o uso dentro de hooks (ganchos). Exemplo:

```
rst $30 ; chama callf
.db 2 ; slot primario 2 (slot id)
.dw $4555 ; endereco a chamar
ret ; fim do hook
```

◇← Slot e endereço passados inline.
◇→ Nenhum.
⊗⇝⊕ AF', BC', DE', HL', IX, IY, DI.

CALPAT *$0084*

■ Calcula o endereço de uma imagem de sprite, que é $PATBAS + 8 \times numsprite$ para sprites 8×8 e $PATBAS + 32 \times numsprite$ para sprites 16×16.

◇← A: número do sprite.

◇→ HL: endereço da imagem do sprite.

⊗⇝⊕ AF, DE, HL.

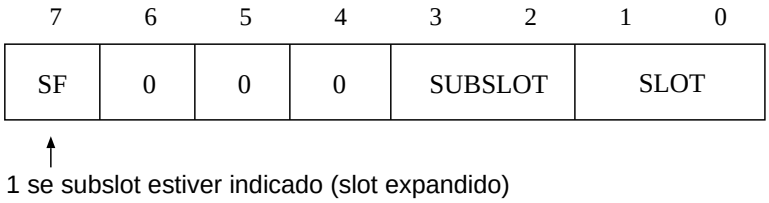


Figura 5.1: Slot ID.

CALSLT *$001C*

■ Chama rotina em qualquer slot.

◇← IY: slot ID (ver Figura 5.1) no byte mais significativo, IX: endereço a chamar.

◇→ Nenhum.

⊗⇝⊕ AF', BC', DE', HL', DI.

CHGCAP *$0132*

■ Controla o LED do CAPS LOCK (apenas o LED, não o estado do CAPS LOCK para efeito de entrada de caracteres).

◇← A: 0=desliga, 1-255=liga.

◇→ Nenhum.

⊗⇝⊕ AF.

CHGCLR *$0062*

■ Muda as cores da tela de acordo com o conteúdo de FORCLR, BAKCLR e BDRCLR.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, BC, HL, EI.

CHGET *$009F*

■ Retorna o próximo caractere no buffer de teclado ou aguarda até que uma tecla seja pressionada.

◇← Nenhum.
◇→ A: caractere.
⊗⇝⊕ AF, EI.

CHGMOD *$005F*

■ Altera o modo de vídeo.

◇← A: modo de tela (0-3).
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, HL, EI.

CHGSND *$0135*

■ Altera o estado do click sonoro do teclado.

◇← A: 0=desliga, 1-255=liga.
◇→ Nenhum.
⊗⇝⊕ AF.

CHKRAM *$0000*

■ Verifica (de forma não destrutiva) a presença de RAM em todos os slots, prepara os slots primário e secundário para mapear a maior área encontrada, reseta a área de trabalho ($F380 a $FFC9), entra em modo de interrupção 1 (IM 1) e passa o controle para outras rotinas de inicialização da BIOS.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, HL, SP.

CHPUT *$00A2*

■ Escreve um caractere na tela em modo texto e incrementa posição do cursor.

◇← A: caractere.
◇→ Nenhum.
⊗⇝⊕ EI.

CHSNS *$009C*

■ Verifica se um caractere do teclado está pronto. Em modo gráfico apenas compara os apontadores do buffer de teclado (GETPNT e PUTPNT), em modo texto também atualiza o display das teclas de função.

◇← Nenhum.
◇→ Flag Z=0 se houver caractere pronto.
⊗⇝⊕ AF, EI.

CLRSPR *$0069*

■ Limpa todos os sprites.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, HL, EI.

CLS *$00C3*

■ Limpa a tela (qualquer modo de tela).

◇← A flag Z precisa estar setada.
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, EI.

DCOMPR *$0020*

■ Comparação aritmética entre DE e HL.

◇← DE, HL.
◇→ Flag C zerada se HL maior que DE, Flag Z setada se HL igual a DE, Flag C setada se HL menor que DE.
⊗⇝⊕ AF.

DISSCR *$0041*

■ Desativa a tela (desliga bit 6 do registrador de modo 1 do VDP).

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, BC, EI.

DOWNC *$0108*

■ Modifica as variáveis de sistema CLOC e CMASK para apontar para o pixel abaixo do pixel atual. São produzidos endereços incorretos se o limite da tela for excedido.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF
.

DSPFNK *$00CF*

■ Mostra o display das teclas de função na última linha dos modos texto (não faz nada nos modos gráficos).

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF,BC,DE,EI.

ENASCR *$0044*

■ Ativa a tela (liga bit 6 do registrador de modo 1 do VDP).

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, BC, EI.

ENASLT *$0024*

■ Mapeia uma página (16K) de qualquer slot. A tentativa de remapaear a página 0 ($0000-$3FFF, onde reside o BIOS) causa perda do sistema.

◇← A: slot ID (ver Figura 5.1), HL: endereço.
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, DI.

ERAFNK *$00CC*

■ Apaga o display das teclas de função na última linha dos modos texto (não faz nada nos modos gráficos).

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF,DE,EI.

FETCHC *$0114*

■ Obtém o endereço físico do pixel atual (valores de CLOC e CMASK).

◇← Nenhum.
◇→ A: CMASK, HL: CLOC.
⊗⇝⊕ A, HL.

FILVRM *$0056*

■ Prenche uma área da VRAM com um byte de dados.

◇← A: byte, BC: comprimento, HL: endereço VRAM.
◇→ Nenhum.
⊗⇝⊕ AF, BC, EI.

GICINI *$0090*

■ Inicializa o PSG e as variáveis relacionadas à fila do comando PLAY do BASIC. Os registradores 8, 9 e 10 do PSG são colocados em amplitude zero e o registrador 7 é inicializado com $B8, ativando o gerador de sons e desativando o gerador de ruído em cada canal.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ EI.

GRPPRT *$008D*

■ Escreve um caractere na tela gráfica, na posição determinada por GRPACX e GRPACY, com cor FORCLR, e avança a posição.

◇← A: caractere.
◇→ Nenhum.
⊗⇝⊕ EI.

GSPSIZ *$008A*

■ Devolve o número de bytes ocupado por cada imagem de sprite, determinado a partir do bit de tamanho no registrador de modo 1 do VDP.

◇← Nenhum.
◇→ A: bytes por imagem (8 ou 32).
⊗⇝⊕ AF.

GTPAD *$00DB*

■ Lê o estado de um tablet ligado a uma das portas de joystick. O código de função pode estar entre 0 e 7, 0=lê status ($FF se a caneta estiver tocando o tablet, $00 se não), 1=retorna coordenada X, 2=retorna coordenada Y, 3=lê status da tecla ($00=solta, $FF pressionada), e os valores 4 a 7 são as mesmas funções relativas à segunda porta de joystick.

◇← A: código de função.
◇→ A: status/valor lido.
⊗⇝⊕ AF, BC, DE, HL, EI.

GTPDL *$00DE*

■ Lê o valor de um paddle conectado a uma das portas de joystick. Cada uma das seis linhas (4 direções + 2 botões) pode ser um paddle (gerador de pulso monoestável controlado por potenciômetro: pense tele-jogo). Os 6 paddles ímpares estão associados à primeira porta de joystick e os 6 paddles pares estão associados à segunda porta.

◇← A: número do paddle (1-12).
◇→ A: valor do paddle (0-255).
⊗⇝⊕ AF, BC, DE, EI.

GTSTCK *$00D5*

■ Lê a posição do joystick ou das teclas de seta.

◇← A: identificação do joystick (0=teclado, 1=joy 1. 2=joy 2).
◇→ A: código de posição, 0=centro, 1=norte até 8=noroeste, em sentido horário (ver Figura 5.2).
⊗⇝⊕ AF, B, DE, HL, EI.

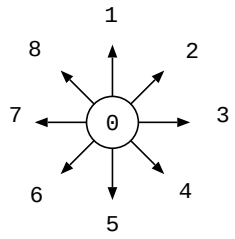


Figura 5.2: Códigos de posição do Joystick.

GTTRIG *$00D8*

■ Verifica os botões dos joysticks ou a barra de espaço.

◇← A: botão (0=espaço, 1=joy 1 botão 1, 2=joy 2 botão 1, 3=joy 1 botão 2, 4=joy 2 botão 2).
◇→ A: Estado ($00 = solto, $FF = pressionado).
⊗⇝⊕ AF, BC, EI.

INIFNK *$003E*

■ Inicializa as strings das teclas de função com seus valores default (em ROM) (copia 160 bytes de $13A9 para FNKSTR).

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ BC, DE, HL.

INIGRP *$0072*

■ Coloca o VDP em modo gráfico 256 × 192 (screen 2).

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, HL, EI.

INIMLT *$0075*

■ Coloca o VDP em modo multicolorido 64 × 48 (screen 3).

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, HL, EI.

INIT32 *$006F*

■ Coloca o VDP em modo texto 32 × 24 (screen 1).

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, HL, EI.

INITIO *$003B*

■ Inicializa o PSG e a porta de impressora. O registrador 7 do PSG é colocado em $80 (fazendo a porta A entrada e a porta B saída), o registrador 15 é colocado em $CF (para inicializar o hardware das portas de joystick), e o registrador 14 tem o bit de modo de teclado colocado em modo KANAMD. Um valor $FF é enviado à porta de status da impressora ($90) para levantar o STROBE. O restante da inicialização é realizado pela rotina GICINI.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, E, EI.

INITXT *$006C*

■ Coloca o VDP em modo texto 40 × 24 (screen 0).

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, HL, EI.

INLIN *$00B1*

■ Recebe caracteres do teclado até que Return ou Ctrl+Stop sejam pressionadas.

◇← Nenhum.
◇→ HL: endereço do buffer, Flag C=1 se terminado com Ctrl+Stop.
⊗⇝⊕ AF, BC, DE, HL, EI.

KEYINT *$0038*

■ Manipulador padrão de interrupções do VDP. O bit 7 do registrador de status é lido para garantir que é uma interrupção do VDP. Se não for, retorna imediatamente. Atualiza entradas de TRPTBL em caso de colisão de sprites, incrementa contadores INTCNT (ON INTERVAL) e JIFFY (TIME), processa fila do comando PLAY se necessário, varre teclado e atualiza estado dos botões dos joysticks.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ EI.

KILBUF *$0156*

■ Limpa KEYBUF, o buffer de 40 caracteres do teclado, igualando GETPNT e PUTPNT.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ HL.

LDIRMV *$0059*

■ Copia um bloco de dados da VRAM para a RAM.

◇← BC: comprimento, DE: endereço RAM (destino), HL: endereço VRAM (fonte).
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, EI.

LDIRVM *$005C*

■ Copia um bloco de dados da memória principal para a VRAM.

◇← BC: comprimento, DE: endereço VRAM (destino), HL: endereço memória principal (fonte).
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, HL, EI.

LEFTC *$00FF*

■ Modifica as variáveis de sistema CLOC e CMASK para apontar para o pixel à esquerda do pixel atual. São produzidos endereços incorretos se o limite da tela for excedido.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF
.

LFTQ *$00F6*

■ Retorna o número de bytes livres em uma das três filas musicais.

◇← A: número da fila
◇→ HL: espaço livre
⊗⇝⊕ AF, BC, HL.

LPTOUT *$00A5*

■ Aguarda que a impressora esteja disponível (não BUSY), escreve um caractere na porta de impressora e gera um pulso 0 no sinal de STROBE. Pode ser interrompido por Ctrl-Stop.

◇← A: caractere a imprimir.
◇→ Flag C se houver interrupção por Ctrl-Stop.
⊗⇝⊕ AF.

LPTSTT *$00A8*

■ Lê o sinal de BUSY na porta de impressora (porta $90).

◇← Nenhum.
◇→ A=0 e Flag Z setado se a impressora estiver ocupada.
⊗⇝⊕ AF.

MAPXYC *$0111*

■ Converte uma coordenada de tela dos modos gráficos no endereço VRAM correspondente.

◇← BC: coordenada x, DE: coordenada y
◇→ Endereço VRAM em CLOC ($F92A), Máscara de bit em CMASK ($F92C)
⊗⇝⊕ AF, D, HL.

NMI *$0066*

■ Tratador padrão para a interrupção não-mascarável. Não faz nada.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ Nenhum.

NSETCX *$0123*

■ Modifica para ATRBYT o atributo de uma sequência horizontal de pixels com o pixel atual (CLOC/CMASK) como pixel mais à esquerda.

◇← HL: número de pixels.
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, HL, EI.

PINLIN *$00AE*

■ Recebe caracteres do teclado até que Return ou Ctrl+Stop sejam pressionadas.

◇← Nenhum.
◇→ HL: endereço do buffer, Flag C=1 se terminado com Ctrl+Stop.
⊗⇝⊕ AF, BC, DE, HL, EI.

POSIT *$00C6*

■ Posiciona cursor (em modo texto).

◇← H: coluna, L: linha (a origem da tela é (1,1)).
◇→ Nenhum.
⊗⇝⊕ AF, EI.

PUTQ *$00F9*

■ Coloca um byte de dados em uma das três filas musicais.

◇← A: número da fila, E: dado.
◇→ Flag Z=1 se a fila estiver cheia.
⊗⇝⊕ AF, BC, HL.

QINLIN *$00B4*

■ Apresenta um caractere '?' e um espaço e transfere o controle para a rotina INLIN (usado pelo manipulador do comando INPUT).

◇← Nenhum.
◇→ HL: endereço do buffer, Flag C=1 se terminado com Ctrl+Stop.
⊗⇝⊕ AF, BC, DE, HL, EI.

RDPSG *$0096*

■ Lê um byte de dados de um dos 16 registradores do PSG.

◇← A: registrador.
◇→ A: valor lido.
⊗⇝⊕ A.

RDSLT *$000C*

■ Lê um byte de qualquer slot.

◇← A: ID do slot (ver Figura 5.1), HL: endereço.
◇→ A: valor lido.
⊗⇝⊕ AF, BC, DE, DI.

RDVDP *$013E*

■ Lê o registrador de status do VDP pela porta de comando. O uso desta rotina pode confundir o manipulador de interrupção.

◇← Nenhum.
◇→ A: registrador de status do VDP.
⊗⇝⊕ A.

RDVRM *$004A*

■ Lê um byte da VRAM.

◇← HL: endereço VRAM.
◇→ A: byte lido.
⊗⇝⊕ AF, EI.

READC *$011D*

■ Lê atributo do pixel atual (CLOC/CMASK).

◇← Nenhum.
◇→ A: cor do pixel atual.
⊗⇝⊕ AF, EI.

RIGHTC *$00FC*

■ Modifica as variáveis de sistema CLOC e CMASK para apontar para o pixel à direita do pixel atual. São produzidos endereços incorretos se o limite da tela for excedido.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF

.

RSLREG *$0138*

■ Lê o conteúdo do registrador de slot primário pela porta A da PPI.

◇← Nenhum.
◇→ A: valor lido.
⊗⇝⊕ A.

SCALXY *$10E*

■ Corta coordenadas fora da tela para coordenadas válidas (coordenadas X negativas são transformadas em 0 e maiores que 255 são transformadas em 255, coordenadas Y negativas são transformadas em 0 e maiores que 191 são transformadas em 191).

◇← BC: coordenada X, DE: coordenada Y.
◇→ BC: X, DE: Y, Flag C=1 se houve corte.
⊗⇝⊕ AF.

SETATR *$011A*

■ Muda o atributo corrente das rotinas SETC e NSETCX, copiando-o para a variável ATRBYT.

◇← A: cor.
◇→ Flag C=1 se a cor for inválida.
⊗⇝⊕ F

.

SETC *$0120*

■ Modifica o atributo do pixel atual (CLOC/CMASK) para o valor de ATRBYT.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, EI.

SETRD *$0050*

■ Prepara leitura da VRAM, escrevendo o endereço na porta de comando do VDP.

◇← HL: endereço.
◇→ Nenhum.
⊗⇝⊕ AF, EI.

SETWRT *$0053*

■ Prepara uma escrita na VRAM, escrevendo o endereço na porta de comando do VDP.

◇← HL: endereço.
◇→ Nenhum.
⊗⇝⊕ AF, EI.

SNSMAT *$0141*

■ Lê uma linha completa da matriz do teclado. A porta C da PPI é lida e reescrita com número de linha nos bits apropriados, e a saída é obtida da porta B da PPI.

◇← A: Número de linha do teclado.
◇→ A: Colunas da linha do teclado.
⊗⇝⊕ AF, C, EI.

STMOTR *$00F3*

■ Liga ou desliga o relê do motor do cassete pela porta C da PPI.

◇← A: 0=desligado, 1=ligado, 255=estado de corrente reversa.
◇→ Nenhum.
⊗⇝⊕ AF.

STOREC *$0117*

■ Muda o endereço do pixel atual (CLOC e CMASK).

◇← A: CMASK, HL: CLOC.
◇→ Nenhum.
⊗⇝⊕ Nada.

.

STRTMS *$0099*

■ Verifica as filas musicais (PLAY) e inicia a execução, se necessário.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ Todos.

TAPIOF *$00E7*

■ Pára o motor do cassete.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ EI.

TAPIN *$00E4*

■ Lê um byte do cassete.

◇← Nenhum.
◇→ A: byte lido, flag C setado em caso de Ctrl-Stop ou erro de I/O.
⊗⇝⊕ AF, BC, DE, L.

TAPION *$00E1*

■ Liga o motor da unidade de fita cassete e analisa o cabeçalho para preparar uma leitura.

◇← Nenhum.
◇→ Flag C se interrompido por Ctrl-Stop.
⊗⇝⊕ AF, BC, DE, HL, DI.

TAPOOF *$00F0*

■ Chamada após uma gravação. Aguarda 550 ms e interrompe o motor do cassete através da chamada TAPIOF.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ EI.

TAPOON *$00EA*

■ Liga o motor do cassete, aguarda 550 ms (para que a unidade atinja a velocidade correta) e escreve um cabeçalho que servirá para determinar a velocidade de gravação do bloco a ser gravado.

◇← A: 0=cabeçalho curto, 1-255=cabeçalho longo.
◇→ Flag C em caso de interrupção por Ctrl-Stop.
⊗⇝⊕ AF, BC, HL, DI.

TAPOUT *$00ED*

■ Escreve um byte de dados no cassete.

◇← A: dado.
◇→ Flag C se houver interrupção com Ctrl-Stop.
⊗⇝⊕ AF, B, HL.

TDOWNC *$010B*

■ Testa se o pixel não está na última linha do vídeo, e então passa o controle para DOWNC.

◇← Nenhum.
◇→ Flag C=1 se o pixel atual estiver na última linha.
⊗⇝⊕ AF
.

TOTEXT *$00D2*

■ Se estiver em modo gráfico (screens 2 ou 3) volta ao modo texto anterior, senão não faz nada.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF, BC, DE, HL, EI.

TUPC *$0105*

■ Testa se o pixel não está na primeira linha do vídeo, e então passa o controle para UPC.

◇← Nenhum.
◇→ Flag C=1 se o pixel atual estiver na primeira linha.
⊗⇝⊕ AF
.

UPC *$0102*

■ Modifica as variáveis de sistema CLOC e CMASK para apontar para o pixel acima do pixel atual. São produzidos endereços incorretos se o limite da tela for excedido.

◇← Nenhum.
◇→ Nenhum.
⊗⇝⊕ AF
.

WRSLT *$0014*

■ Escreve um byte em qualquer slot.

◇← A: slot ID (ver Figura 5.1), HL: endereço, E: byte de dado.
◇→ Nenhum.
⊗⇝⊕ AF, BC, D, DI.

WRTPSG *$0093*

■ Escreve um byte de dados em um dos 16 registradores do PSG.

◇← A: registrador, E: valor.
◇→ Nenhum.
⊗⇝⊕ EI.

WRTVDP *$0047*

■ Escreve um byte de dados em um dos registradores de modo do VDP, fazendo cópia na área de trabalho (RG0SAV a RG7SAV).

◇← B: byte de dado, C: registrador de modo.
◇→ Nenhum.
⊗⇝⊕ AF, B, EI.

WRTVRM *$004D*

■ Escreve um byte na VRAM.

◇← A: byte de dado, HL: endereço VRAM.
◇→ Nenhum.
⊗⇝⊕ EI.

WSLREG *$013B*

■ Escreve no registrador de slot primário pela porta A da PPI.

◇← A: Valor a escrever.
◇→ Nenhum.
⊗⇝⊕ Nada.

5.3 Variáveis da área de trabalho

O bloco de RAM entre $F380 e $FD99 contém rotinas auxiliares às rotinas de comutação de slot da BIOS (que não são chamadas diretamente) e diversas variáveis que são consultadas e atualizadas tanto por rotinas da BIOS quanto pelo interpretador BASIC.

Tabela 5.2: Variáveis da área de trabalho, por endereço.

Endereço	Nome	Bytes	Resumo
$F39A	USRTAB	20 (10×2)	Endereços das funções USR0...USR9.
$F3AE	LINL40	1	Largura da screen 0. (comando WIDTH).
$F3AF	LINL32	1	Largura da screen 1. (comando WIDTH).
$F3B0	LINLEN	1	Largura do modo de texto atual.
$F3B1	CRTCNT	1	Número de linhas nos modos texto.
$F3B3	TXTNAM	2	Modo texto 40×24: Base da tabela de nomes.
$F3B5	TXTCOL	2	Modo texto 40×24: Base da tabela de cores.
$F3B7	TXTCGP	2	Modo texto 40×24: Base da tabela de caracteres.
$F3B9	TXTATR	2	Modo texto 40×24: Base dos atributos de sprite.
$F3BB	TXTPAT	2	Modo texto 40×24: Base das imagens de sprite.
$F3BD	T32NAM	2	Modo texto 32×24: Base da tabela de nomes.
$F3BF	T32COL	2	Modo texto 32×24: Base da tabela de cores.
$F3C1	T32CGP	2	Modo texto 32×24: Base da tabela de caracteres.
$F3C3	T32ATR	2	Modo texto 32×24: Base dos atributos de sprite.
$F3C5	T32PAT	2	Modo texto 32×24: Base das imagens de sprite.
$F3C7	GRPNAM	2	Modo gráfico 256×192: Base da tabela de nomes.
$F3C9	GRPCOL	2	Modo gráfico 256×192: Base da tabela de cores.
$F3CB	GRPCGP	2	Modo gráfico 256×192: Base da tabela de caracteres.
$F3CD	GRPATR	2	Modo gráfico 256×192: Base dos atributos de sprite.
$F3CF	GRPPAT	2	Modo gráfico 256×192: Base das imagens de sprite.
$F3D1	MLTNAM	2	Modo gráfico 64×48: Base da tabela de nomes.
$F3D3	MLTCOL	2	Modo gráfico 64×48: Base da tabela de cores.
$F3D5	MLTCGP	2	Modo gráfico 64×48: Base da tabela de caracteres.
$F3D7	MLTATR	2	Modo gráfico 64×48: Base dos atributos de sprite.
$F3D9	MLTPAT	2	Modo gráfico 64×48: Base das imagens de sprite.
$F3DB	CLIKSW	1	Click do teclado: 0=desligado, 1-255: ligado.
$F3DC	CSRY	1	Coordenada Y do cursor em modo texto (1...CRTCNT).
$F3DD	CSRX	1	Coordenada X do cursor em modo texto (1...LINLEN).
$F3DE	CONSDFG	1	Apresentação das teclas de função: 0=desligado.
$F3DF	RGnSAV	7 (7×1)	Cópia dos 8 registradores (n de 0 a 7) de modo (somente escrita) do VDP.
$F3E7	STATFL	1	Cópia do registrador de status do VDP.
$F3E8	TRGFLG	1	Estado dos botões dos joysticks e da barra de espaço.
$F3E9	FORCLR	1	Cor do primeiro plano.
$F3EA	BAKCLR	1	Cor do segundo plano (fundo).
$F3EB	BDRCLR	1	Cor da borda.
$F3EC	MAXUPD	3	(Código automodificável usado para traçar retas).
$F3EF	MINUPD	3	(Código automodificável usado para traçar retas).

⟶

Tabela 5.2: (cont.)

Endereço	Nome	Bytes	Resumo
$F3F2	ATRBYT	1	Cor da tinta gráfica em funções gráficas da BIOS.
$F3F3	QUEUES	2	Contém endereço do bloco de controle das filas musicais.
$F3F5	FRCNEW	1	Indicador que distingue CLOAD (0) de CLOAD? (255).
$F3F6	SCNCNT	1	Contador usado para controlar a frequência das varreduras do teclado.
$F3F7	REPCNT	1	Contador usado para controlar o número de repetições de uma tecla.
$F3F8	PUTPNT	2	Endereço de inserção em KEYBUF (buffer circular).
$F3FA	GETPNT	2	Endereço de remoção em KEYBUF (buffer circular).
$F3FC	CS1200	5	Parâmetros do cassete para 1200 baud.
$F401	CS2400	5	Parâmetros do cassete para 2400 baud.
$F406	LOW	2	Parâmetros do cassete atuais: ciclo LO.
$F408	HIGH	2	Parâmetros do cassete atuais: ciclo HI.
$F40A	HEADER	1	Parâmetros do cassete atuais: ciclos no cabeçalho.
$F40B	ASPCT1	2	Inverso da razão de aspecto do CIRCLE × 256.
$F40D	ASPCT2	2	Razão de aspecto do CIRCLE × 256.
$F40F	ENDPRG	5	Usado pelo interpretador BASIC para ON ERROR GOTO.
$F414	ERRFLG	1	Armazena o código de erro do BASIC.
$F415	LPTPOS	1	Usada por LPRINT, armazena posição da cabeça de impressão.
$F416	PRTFLG	1	Determina se OUTDO deve direcionar sua saída para a tela (0) ou impressora (1).
$F417	NTMSXP	1	Determina conversão de caracteres gráficos de OUTDO (0=sem conversão, 1-255=espaços).
$F418	RAWPRT	1	Usada por OUTDO: 0=trata prefixos gráficos, 1=envia dados para a impressora sem tratamento.
$F419	VLZADR	2	Temporário usado pela função VAL do BASIC.
$F41B	VLZDAT	1	Temporário usado pela função VAL do BASIC.
$F41E	KBFMIN	1	Utilizado no tratamento de erros do BASIC.
$F41F	KBUF	318	Utilizado no tratamento de erros do BASIC.
$F55D	BUFMIN	1	Utilizado na entrada de linhas BASIC.
$F55E	BUF	259	Utilizado na entrada de linhas BASIC.
$F661	TTYPOS	1	Posição de tela atual, usada pelo PRINT.
$F662	DIMFLG	1	Usado pelo interpretador BASIC na instrução DIM.
$F663	VALTYP	1	Usado pelo interpretador BASIC.
$F664	DORES	1	Usada pelo interpretador BASIC (linhas DATA).
$F665	DONUM	1	Usada pelo interpretador BASIC.
$F666	CONTXT	2	Usada pelo interpretador BASIC.
$F668	CONSAV	1	Usada pelo interpretador BASIC.
$F669	CONTYP	1	Usada pelo interpretador BASIC.
$F66A	CONLO	8	Usada pelo interpretador BASIC.
$F674	STKTOP	2	Contém o endereço do topo da pilha.
$F676	TXTTAB	2	Contém o endereço do texto de programa BASIC.

⟶

Tabela 5.2: (cont.)

Endereço	Nome	Bytes	Resumo
$F678	TEMPPT	2	Aponta para a próxima entrada livre de TEMPST.
$F67A	TEMPST	30	Buffer de descritores de string (BASIC).
$F698	DSCTMP	3	Buffer temporário para funções de string (BASIC).
$F69B	FRETOP	2	Aponta para a próxima posição livre no buffer apontado por MEMSIZ.
$F69D	TEMP3	2	Variável temporária (interpretador BASIC).
$F69F	TEMP8	2	Variável temporária (interpretador BASIC).
$F6A1	ENDFOR	2	Usada pelo interpretador BASIC (loops FOR).
$F6A3	DATLIN	2	Linha do programa BASIC do item DATA atual.
$F6A5	SUBFLG	1	Usada pelo interpretador BASIC.
$F6A6	FLGINP	1	BASIC: distingue INPUT (0) de READ (1-255).
$F6A7	TEMP	2	Variável temporária (interpretador BASIC).
$F6A9	PTRFLG	1	Usada pelo interpretador BASIC.
$F6AA	AUTFLG	1	Flag do modo AUTO do BASIC.
$F6AB	AUTLIN	2	Número da linha AUTO atual (BASIC).
$F6AD	AUTINC	2	Incremento atual do AUTO (BASIC).
$F6AF	SAVTXT	2	Apontador de programa usado pelo manipulador de erro (BASIC).
$F6B3	ERRLIN	2	Número da linha que gerou o erro (BASIC).
$F6B5	DOT	2	Usada pelo interpretador BASIC.
$F6B7	ERRTXT	2	Usada pelo interpretador BASIC (RESUME).
$F6B9	ONELIN	2	Usada pelo interpretador BASIC (ON ERROR GOTO).
$F6BB	ONEFLG	1	Usada pelo interpretador BASIC (ON ERROR GOTO).
$F6BC	TEMP2	2	Variável temporária (interpretador BASIC).
$F6BE	OLDLIN	2	Linha que terminou o programa BASIC, usada por CONT.
$F6C0	OLDTXT	2	Aponta a instrução que terminou o programa.
$F6C2	VARTAB	2	Aponta para a área de armazenamento de variáveis (BASIC).
$F6C4	ARYTAB	2	Aponta para a área de armazenamente de arrays (BASIC).
$F6C6	STREND	2	Aponta para o byte seguinte à área de ARYTAB (BASIC).
$F6C8	DATPTR	2	Aponta para item DATA atual (BASIC).
$F6CA	DEFTBL	26	Tipos de variável, por letra (BASIC), definidos por DEFINT, DEFSTR, etc.
$F6E4	PRMSTK	2	Usada pelo interpretador BASIC.
$F6E6	PRMLEN	2	Usada pelo interpretador BASIC.
$F6E8	PARM1	100	Usada pelo interpretador BASIC (buffer do FN).
$F74C	PRMPRV	2	Usada pelo interpretador BASIC (FN).
$F74E	PRMLN2	2	Usada pelo interpretador BASIC (FN).
$F750	PARM2	100	Usada pelo interpretador BASIC (buffer do FN).
$F7B4	PRMFLG	1	Usada pelo interpretador BASIC.
$F7B5	ARYTA2	2	Usada pelo interpretador BASIC.

⟶

Tabela 5.2: (cont.)

Endereço	Nome	Bytes	Resumo
$F7B7	NOFUNS	1	Usada pelo interpretador BASIC.
$F7B8	TEMP9	2	Variável temporária (interpretador BASIC).
$F7BA	FUNACT	2	Usada pelo interpretador BASIC.
$F7BC	SWPTMP	8	Usada pelo interpretador BASIC (SWAP).
$F7C4	TRCFLG	1	Ativado quando TRON está ligado (BASIC).
$F7C5	FBUFFR	43	Buffer de conversão numérica (BASIC).
$F7F2	DECTM2	2	Variável temporária (interpretador BASIC).
$F7F4	DECCNT	2	Variável temporária (interpretador BASIC).
$F7F6	DAC	16	Buffer de avaliação de expressão do BASIC.
$F806	HOLD8	65	Buffer temporário de multiplicação (BASIC).
$F847	ARG	16	Buffer de avaliação de expressão do BASIC.
$F857	RNDX	8	Contém o último número aleatório (precisão dupla).
$F85F	MAXFIL	1	Número de buffers de E/S alocados (BASIC).
$F860	FILTAB	2	Aponta para a tabela de FCBs dos buffers de E/S (BASIC).
$F862	NULBUF	2	Aponta para o buffer de E/S.
$F864	PTRFIL	2	Aponta para o FCB do buffer de E/S ativo.
$F866	FILNAM	11	Buffer de nome de arquivo. (BASIC).
$F871	FILNM2	11	Buffer de nome de arquivo. (BASIC).
$F87C	NLONLY	1	Usada pelo interpretador BASIC.
$F87D	SAVEND	2	Usada pelo interpretador BASIC.
$F87F	FNKSTR	160	Buffer com strings das teclas de função.
$F91F	CGPNT	3	Aponta para a tabela de caracteres em ROM (Slot ID 0 seguido do endereço $1BBF).
$F922	NAMBAS	2	Base da tabela de nomes no modo de vídeo atual.
$F924	CGPBAS	2	Base da tabela de caracteres no modo de vídeo atual.
$F926	PATBAS	2	Base da tabela de imagens de sprites no modo de vídeo atual.
$F928	ATRBAS	2	Base da tabela de atributos de sprites no modo de vídeo atual.
$F92A	CLOC	2	Endereço do pixel atual (funções gráficas da BIOS).
$F92C	CMASK	1	Máscara do pixel atual.
$F92D	MINDEL	2	Usado pela instrução LINE.
$F92F	MAXDEL	2	Usado pela instrução LINE.
$F931	ASPECT	2	Usado pela instrução CIRCLE.
$F933	CENCNT	2	Usado pela instrução CIRCLE.
$F935	CLINEF	1	Usado pela instrução CIRCLE.
$F936	CNPNTS	2	Usado pela instrução CIRCLE.
$F938	CPLOTF	1	Usado pela instrução CIRCLE.
$F939	CPCNT	2	Usado pela instrução CIRCLE.
$F93B	CPCNT8	2	Usado pela instrução CIRCLE.
$F93D	CRCSUM	2	Usado pela instrução CIRCLE.
$F93F	CSTCNT	2	Usado pela instrução CIRCLE.
$F941	CSCLXY	1	Usado pela instrução CIRCLE.
$F942	CSAVEA	2	Temporário usado por funções gráficas da BIOS.
$F944	CSAVEM	1	Temporário usado por funções gráficas da BIOS.

⟶

Tabela 5.2: (cont.)

Endereço	Nome	Bytes	Resumo
$F945	CXOFF	2	Usado pela instrução CIRCLE.
$F947	CYOFF	2	Usado pela instrução CIRCLE.
$F949	LOHMSK	1	Usado pela instrução PAINT.
$F94A	LOHDIR	1	Usado pela instrução PAINT.
$F94B	LOHADR	2	Usado pela instrução PAINT.
$F94D	LOHCNT	2	Usado pela instrução PAINT.
$F94F	SKPCNT	2	Usado pela instrução PAINT.
$F951	MOVCNT	2	Usado pela instrução PAINT.
$F953	PDIREC	1	Usado pela instrução PAINT.
$F954	LEPROG	1	Usado pela instrução PAINT.
$F955	RTPROG	1	Usado pela instrução PAINT.
$F958	MCLFLG	1	Linguagem de macro atual, 0=DRAW, 1-255=PLAY.
$F959	QUETAB	24	Blocos de controle das filas musicais.
$F971	QUEBAK	4	Usado pelo manipulador de fila musical.
$F975	VOICAQ	128	Buffer da fila musical A.
$F9F5	VOICBQ	128	Buffer da fila musical B.
$FA75	VOICCQ	128	Buffer da fila musical C.
$FAF5	RS2IQ	64	Buffer da fila RS232.
$FB35	PRSCNT	1	Usado pelo interpretador BASIC (PLAY).
$FB36	SAVSP	2	Usado pelo interpretador BASIC (PLAY).
$FB38	VOICEN	1	Voz atual do interpretador PLAY.
$FB39	SAVVOL	2	Usado pelo interpretador BASIC (PLAY).
$FB3B	MCLLEN	1	Comprimento do operando de macro-linguagem analisado.
$FB3C	MCLPTR	2	Aponta para caractere de macro-linguagem sendo analisado.
$FB3E	QUEUEN	1	Fila atual do interpretador PLAY.
$FB3F	MUSICF	1	Usado pelo interpretador BASIC (PLAY).
$FB40	PLACNT	1	Usado pelo interpretador BASIC (PLAY).
$FB41	VCBA	37	Buffer com parâmetros da voz A do PLAY.
$FB66	VCBB	37	Buffer com parâmetros da voz B do PLAY.
$FB8B	VCBC	37	Buffer com parâmetros da voz C do PLAY.
$FBB0	ENSTOP	1	Se diferente de 0, faz warm boot quando CODE+GRAPH+CTRL+SHIFT forem pressionadas.
$FBB1	BASROM	1	Ativa (0) ou desativa (1-255) manipulador de CTRL+STOP.
$FBB2	LINTTB	24	Variável interna de funções da BIOS.
$FBCA	FSTPOS	2	Usada internamente pelo editor de tela do BASIC.
$FBCC	CURSAV	1	Armazena o caractere sob o cursor.
$FBCD	FNKSWI	1	Usada pela rotina CHSNS para determinar se SHIFT está pressionado (0) ou não (1) para apresentar as strings das teclas de função.
$FBCE	FNKFLG	10	Usada pelo BASIC. (indicadores de KEY(n) ON).
$FBD8	ONGSBF	1	Usado pelo interpretador BASIC.
$FBDA	OLDKEY	11	Armazena o estado anterior da matriz de teclado.

⟶

Tabela 5.2: (cont.)

Endereço	Nome	Bytes	Resumo
$FBE5	NEWKEY	11	Armazena o estado atual da matriz de teclado.
$FBF0	KEYBUF	40	Buffer circular do teclado (caracteres decodificados).
$FC18	LINWRK	40	Buffer de linha de tela, usado pelo BIOS.
$FC40	PATWRK	8	Buffer usado pelo BIOS.
$FC48	BOTTOM	2	Armazena o início da RAM usada pelo interpretador BASIC.
$FC4C	TRPTBL	78 (26×3)	Usado pelas instruções de interrupção (ON...) do BASIC.
$FC9A	RTYCNT	1	Não-utilizada.
$FC9B	INTFLG	1	Flag de detecção de CTRL-STOP (3) e STOP (4).
$FC9C	PADY	1	Última coordenada Y do tablet.
$FC9D	PADX	1	Última coordenada X do tablet.
$FC9E	JIFFY	2	Contador incrementado a cada interrupção do VDP.
$FCA0	INTVAL	2	Duração do intervalo do ON INTERVAL (BASIC).
$FCA2	INTCNT	2	Contador do ON INTERVAL (BASIC).
$FCA4	LOWLIM	1	Duração mínima do bit de partida no cassete (TAPION).
$FCA5	WINWID	1	Duração de discriminação LO/HI (TAPION).
$FCA6	GRPHED	1	Variável auxiliar da rotina CNVCHR do BIOS.
$FCA7	ESCCNT	1	Variável auxiliar da rotina CHPUT do BIOS.
$FCA8	INSFLG	1	Indica modo de inserção do editor de tela.
$FCAA	CSTYLE	1	Estilo do cursor, bloco (0) ou sublinhado (1-255).
$FCAB	CAPST	1	Status do CAPS LOCK (0=desligado, 1-255=ligado).
$FCAC	KANAST	1	Status do KANA LOCK (0=desligado, 1-255=ligado).
$FCAD	KANAMD	1	Modo kana (MSX japoneses).
$FCAE	FLBMEM	1	Usada pelo manipulador de erro de E/S.
$FCAF	SCRMOD	1	Modo de tela (SCREEN) atual.
$FCB0	OLDSCR	1	Último modo de texto.
$FCB1	CASPRV	1	Temporário de E/S do cassete.
$FCB2	BDRATR	1	Usado pelas rotinas gráficas do BIOS.
$FCB3	GXPOS	2	Temporário das rotinas gráficas do BIOS.
$FCB5	GYPOS	2	Temporário das rotinas gráficas do BIOS.
$FCB7	GRPACX	2	Temporário das rotinas gráficas do BIOS.
$FCB9	GRPACY	2	Temporário das rotinas gráficas do BIOS.
$FCBB	DRWFLG	1	Usado pelo manipulador do DRAW.
$FCBC	DRWSCL	1	Usado pelo manipulador do DRAW.
$FCBD	DRWANG	1	Usado pelo manipulador do DRAW.
$FCBE	RUNBNF	1	Usado pelo manipulador do BLOAD.
$FCBF	SAVENT	2	Usado pelo manipulador do BLOAD.
$FCC1	EXPTBL	4	Indicadores de expansão dos 4 slots, ($00=não expandido, $80=expandido).
$FCC5	SLTTBL	4	Cópia dos registradores de slot primário (válidos apenas nos slots expandidos).
$FCC9	SLTATR	64	Atributos de ROM, 16 bytes por slot (desses, 4 por subslot).

⟶

Tabela 5.2: (cont.)

Endereço	Nome	Bytes	Resumo
$FD09	SLTWRK	128	Dois bytes de trabalho local para cada uma das 64 extensões de ROM possíveis.
$FD89	PROCNM	16	Buffer para nome de dispositivo ou instrução a ser analisado por uma ROM de extensão.
$FD99	DEVICE	1	Usada para passar um número de dispositivo para uma ROM de extensão.

Tabela 5.2: Variáveis da área de trabalho, por endereço.

A área entre $FD9A e $FFC9 és utilizada para ganchos de chamadas BIOS/BASIC para permitir a extensão da ROM (Disk BASIC, por exemplo), listados na próxima seção.. A área entre $FFCA e $FFFE (53 bytes) não é utilizada. O endereço $FFFF é utilizado por expansores de slot, como descrito no capítulo 7.

Tabela 5.3: Variáveis da área de trabalho, ordem alfabética.

Nome	End.	Nome	End.	Nome	End.	Nome	End.
ARG	$F847	ARYTA2	$F7B5	ARYTAB	$F6C4	ASPCT1	$F40B
ASPCT2	$F40D	ASPECT	$F931	ATRBAS	$F928	ATRBYT	$F3F2
AUTFLG	$F6AA	AUTINC	$F6AD	AUTLIN	$F6AB	BAKCLR	$F3EA
BASROM	$FBB1	BDRATR	$FCB2	BDRCLR	$F3EB	BOTTOM	$FC48
BUF	$F55E	BUFMIN	$F55D	CAPST	$FCAB	CASPRV	$FCB1
CENCNT	$F933	CGPBAS	$F924	CGPNT	$F91F	CLIKSW	$F3DB
CLINEF	$F935	CLOC	$F92A	CMASK	$F92C	CNPNTS	$F936
CONLO	$F66A	CONSAV	$F668	CONSDFG	$F3DE	CONTXT	$F666
CONTYP	$F669	CPCNT8	$F93B	CPCNT	$F939	CPLOTF	$F938
CRCSUM	$F93D	CRTCNT	$F3B1	CS1200	$F3FC	CS2400	$F401
CSAVEA	$F942	CSAVEM	$F944	CSCLXY	$F941	CSRX	$F3DD
CSRY	$F3DC	CSTCNT	$F93F	CSTYLE	$FCAA	CURSAV	$FBCC
CXOFF	$F945	CYOFF	$F947	DAC	$F7F6	DATLIN	$F6A3
DATPTR	$F6C8	DECCNT	$F7F4	DECTM2	$F7F2	DEFTBL	$F6CA
DEVICE	$FD99	DIMFLG	$F662	DONUM	$F665	DORES	$F664
DOT	$F6B5	DRWANG	$FCBD	DRWFLG	$FCBB	DRWSCL	$FCBC
DSCTMP	$F698	ENDFOR	$F6A1	ENDPRG	$F40F	ENSTOP	$FBB0
ERRFLG	$F414	ERRLIN	$F6B3	ERRTXT	$F6B7	ESCCNT	$FCA7
EXPTBL	$FCC1	FBUFFR	$F7C5	FILNAM	$F866	FILNM2	$F871
FILTAB	$F860	FLBMEM	$FCAE	FLGINP	$F6A6	FNKFLG	$FBCE
FNKSTR	$F87F	FNKSWI	$FBCD	FORCLR	$F3E9	FRCNEW	$F3F5
FRETOP	$F69B	FSTPOS	$FBCA	FUNACT	$F7BA	GETPNT	$F3FA
GRPACX	$FCB7	GRPACY	$FCB9	GRPATR	$F3CD	GRPCGP	$F3CB
GRPCOL	$F3C9	GRPHED	$FCA6	GRPNAM	$F3C7	GRPPAT	$F3CF
GXPOS	$FCB3	GYPOS	$FCB5	HEADER	$F40A	HIGH	$F408
HOLD8	$F806	INSFLG	$FCA8	INTCNT	$FCA2	INTFLG	$FC9B
INTVAL	$FCA0	JIFFY	$FC9E	KANAMD	$FCAD	KANAST	$FCAC
KBFMIN	$F41E	KBUF	$F41F	KEYBUF	$FBF0	LEPROG	$F954

⟶

Tabela 5.3: (cont.)

Nome	End.	Nome	End.	Nome	End.	Nome	End.
LINL32	$F3AF	LINL40	$F3AE	LINLEN	$F3B0	LINTTB	$FBB2
LINWRK	$FC18	LOHADR	$F94B	LOHCNT	$F94D	LOHDIR	$F94A
LOHMSK	$F949	LOW	$F406	LOWLIM	$FCA4	LPTPOS	$F415
MAXDEL	$F92F	MAXFIL	$F85F	MAXUPD	$F3EC	MCLFLG	$F958
MCLLEN	$FB3B	MCLPTR	$FB3C	MINDEL	$F92D	MINUPD	$F3EF
MLTATR	$F3D7	MLTCGP	$F3D5	MLTCOL	$F3D3	MLTNAM	$F3D1
MLTPAT	$F3D9	MOVCNT	$F951	MUSICF	$FB3F	NAMBAS	$F922
NEWKEY	$FBE5	NLONLY	$F87C	NOFUNS	$F7B7	NTMSXP	$F417
NULBUF	$F862	OLDKEY	$FBDA	OLDLIN	$F6BE	OLDSCR	$FCB0
OLDTXT	$F6C0	ONEFLG	$F6BB	ONELIN	$F6B9	ONGSBF	$FBD8
PADX	$FC9D	PADY	$FC9C	PARM1	$F6E8	PARM2	$F750
PATBAS	$F926	PATWRK	$FC40	PDIREC	$F953	PLACNT	$FB40
PRMFLG	$F7B4	PRMLEN	$F6E6	PRMLN2	$F74E	PRMPRV	$F74C
PRMSTK	$F6E4	PROCNM	$FD89	PRSCNT	$FB35	PRTFLG	$F416
PTRFIL	$F864	PTRFLG	$F6A9	PUTPNT	$F3F8	QUEBAK	$F971
QUETAB	$F959	QUEUEN	$FB3E	QUEUES	$F3F3	RAWPRT	$F418
REPCNT	$F3F7	RGnSAV	$F3DF	RNDX	$F857	RS2IQ	$FAF5
RTPROG	$F955	RTYCNT	$FC9A	RUNBNF	$FCBE	SAVEND	$F87D
SAVENT	$FCBF	SAVSP	$FB36	SAVTXT	$F6AF	SAVVOL	$FB39
SCNCNT	$F3F6	SCRMOD	$FCAF	SKPCNT	$F94F	SLTATR	$FCC9
SLTTBL	$FCC5	SLTWRK	$FD09	STATFL	$F3E7	STKTOP	$F674
STREND	$F6C6	SUBFLG	$F6A5	SWPTMP	$F7BC	T32ATR	$F3C3
T32CGP	$F3C1	T32COL	$F3BF	T32NAM	$F3BD	T32PAT	$F3C5
TEMP2	$F6BC	TEMP3	$F69D	TEMP8	$F69F	TEMP9	$F7B8
TEMP	$F6A7	TEMPPT	$F678	TEMPST	$F67A	TRCFLG	$F7C4
TRGFLG	$F3E8	TRPTBL	$FC4C	TTYPOS	$F661	TXTATR	$F3B9
TXTCGP	$F3B7	TXTCOL	$F3B5	TXTNAM	$F3B3	TXTPAT	$F3BB
TXTTAB	$F676	USRTAB	$F39A	VALTYP	$F663	VARTAB	$F6C2
VCBA	$FB41	VCBB	$FB66	VCBC	$FB8B	VLZADR	$F419
VLZDAT	$F41B	VOICAQ	$F975	VOICBQ	$F9F5	VOICCQ	$FA75

Tabela 5.3: Variáveis da área de trabalho, ordem alfabética.

5.4 Ganchos

A região entre `$FD9A` e `$FFC9` contem 112 ganchos, de 5 bytes cada, o que é suficiente para uma chamada para qualquer conector:

```
rst $30      ; $f7     CALLF
.db slotid   ; $??     slotid
.dw endereco ; $?? $?? endereco da rotina
ret          ; $c9     retorna
```

Na inicialização toda a região é inicializada com RETs ($c9). Os ganchos são chamados de partes estratégicas da BIOS para permitir a extensão do software em ROM, como por exemplo a adição de rotinas de disco.

Tabela 5.4: Ganchos

Endereço	Nome	Resumo
$FD9A	HKEYI	Manipulador de Interrupção ($0C4A).
$FD9F	HTIMI	Manipulador de Interrupção ($0C53).
$FDA4	HCHPU	Rotina CHPUT ($08C0).
$FDA9	HDSPC	Mostra cursor ($09E6).
$FDAE	HERAC	Apaga cursor ($0A33).
$FDB3	HDSPF	Rotina DSPFNK ($0B2B).
$FDB8	HERAF	Rotina ERAFNK ($0B15).
$FDBD	HTOTE	Rotina TOTEXT ($0842).
$FDC2	HCHGE	Rotina CHGET ($10CE).
$FDC7	HINIP	Cópia da tabela de caracteres (ROM→VDP) ($071E).
$FDCC	HKEYC	Decodificador do teclado ($1025).
$FDD1	HKEYA	Decodificador do teclado ($0F10).
$FDD6	HNMI	Rotina NMI ($1398).
$FDDB	HPINL	Rotina PINLIN ($23BF).
$FDE0	HQINL	Rotina QINLIN ($23CC).
$FDE5	HINLI	Rotina INLIN ($23D5).
$FDEA	HONGO	ON *dispositivo* GOSUB ($7810).
$FDEF	HDSKO	DSKO$ ($7C16).
$FDF4	HSETS	SET ($7C1B).
$FDF9	HNAME	NAME ($7C20).
$FDFE	HKILL	KILL ($7C25).
$FE03	HIPL	IPL ($7C2A).
$FE08	HCOPY	COPY ($7C2F).
$FE0D	HCMD	DSKF ($7C39).
$FE17	HDSKI	DSKI$ ($7C3E).
$FE1C	HATTR	ATTR$ ($7C43).
$FE21	HLSET	LSET ($7C48).
$FE26	HRSET	RSET ($7C4D).
$FE2B	HFIEL	FIELD ($7C52).
$FE30	HMKI	MKI$ ($7C57).
$FE35	HMKS	MKS$ ($7C5C).
$FE3A	HMKD	MKD$ ($7C61).
$FE3F	HCVI	CVI ($7C66).
$FE44	HCVS	CVS ($7C6B).
$FE49	HCVD	CVS ($7C70).
$FE4E	HGETP	Localizar FCB ($6A93).
$FE53	HSETF	Localizar FCB ($6AB3).
$FE58	HNOFO	OPEN: not found ($6AF6).
$FE5D	HNULO	OPEN ($6B0F).
$FE62	HNTFL	Fecha buffer 0 de I/O ($6B3B).
$FE67	HMERG	MERGE/LOAD ($6B63).
$FE6C	HSAVE	SAVE ($6BA6).
$FE71	HBINS	SAVE ($6BCE).
$FE76	HBINL	MERGE/LOAD ($6BD4).
$FE7B	HFILE	FILES ($6C2F).
$FE80	HDGET	GET/PUT ($6C3B).

⟶

Tabela 5.4: (cont.)

Endereço	Nome	Resumo
$FE85	HFILO	Saída Sequencial ($6C51).
$FE8A	HINDS	Entrada Sequencial ($6C79).
$FE8F	HRSLF	INPUT$ ($6CD8).
$FE94	HSAVD	LOC ($6D03), LOF ($6D14), EOF ($6D25), FPOS ($6D39).
$FE99	HLOC	LOC ($6D0F).
$FE9E	HLOF	LOF ($6D20).
$FEA3	HEOF	EOF ($6D33).
$FEA8	HFPOS	FPOS ($6D43).
$FEAD	HBAKU	LINE INPUT # ($6E36).
$FEB2	HPARD	Análise de nome de dispositivo ($6F15).
$FEB7	HNODE	Análise de nome de dispositivo ($6F33).
$FEBC	HPOSD	Análise de nome de dispositivo ($6F37).
$FEC1	HDEVN	*Não utilizado*
$FEC6	HGEND	Despachador de função de I/O ($6F8F).
$FECB	HRUNC	Processar-Liberar ($629A).
$FED0	HCLEA	Processar-Liberar ($62A1).
$FED5	HLOPD	Processar-Liberar ($62AF).
$FEDA	HSTKE	Limpar Pilha ($62F0).
$FEDF	HISFL	Rotina ISFLIO ($145F).
$FEE4	HOUTD	Rotina OUTDO ($1B46).
$FEE9	HCRDO	CR, LF para OUTDO ($7328).
$FEEE	HDSKC	Entrada de linha no laço principal ($7274).
$FEF3	HDOGR	Traçar linha ($593C).
$FEF8	HPRGE	Fim de programa ($4039).
$FEFD	HERRP	Manipulador de erro ($40DC).
$FF02	HERRF	Manipulador de erro ($40FD).
$FF07	HREAD	"OK" do laço principal ($4128).
$FF0C	HMAIN	Laço Principal ($4134).
$FF11	HDIRD	Comando direto do laço principal ($41A8).
$FF16	HFINI	Fim do laço principal ($4237).
$FF1B	HFINE	Fim do laço principal ($4247).
$FF20	HCRUN	Tokenização ($42B9).
$FF25	HCRUS	Tokenização ($4353).
$FF2A	HISRE	Tokenização ($437C).
$FF2F	HNTFN	Tokenização ($43A4).
$FF34	HNOTR	Tokenização ($44EB).
$FF39	HSNGF	FOR ($45D1).
$FF3E	HNEWS	laço de execução: novo comando ($4601).
$FF43	HGONE	laço de execução ($4646).
$FF48	HCHRG	Rotina CHRGTR ($4666).
$FF4D	HRETU	RETURN ($4821).
$FF52	HPRTF	PRINT ($4A5E).
$FF57	HCOMP	PRINT ($4A94).
$FF5C	HFINP	PRINT ($4AFF).
$FF61	HTRMN	Erro READ/INPUT ($484D).
$FF66	HFRME	Avaliador de expressões ($4C6D).

⟶

Tabela 5.4: (cont.)

Endereço	Nome	Resumo
$FF6B	HNTPL	Avaliador de expressões ($4CA6).
$FF70	HEVAL	Avaliador de fatores ($4DD9).
$FF75	HOKNO	Avaliador de fatores ($4F2C).
$FF7A	HFING	Avaliador de fatores ($4F3E).
$FF7F	HISMI	laço de execução ($51C3).
$FF84	HWIDT	WIDTH ($51CC).
$FF89	HLIST	LIST ($522E).
$FF8E	HBUFL	Destokenização ($532D).
$FF93	HFRQI	Conversão para inteiro ($543F).
$FF98	HSCNE	Conversão de número de linha para ponteiro ($5514).
$FF9D	HFRET	Liberar descritor ($67EE).
$FFA2	HPTRG	Busca de variável ($5EA9).
$FFA7	HPHYD	Rotina PHYDIO ($148A).
$FFAC	HFORM	Rotina FORMAT ($148E).
$FFB1	HERRO	Manipulador de erro ($406F).
$FFB6	HLPTO	Rotina LPTOUT ($085D).
$FFBB	HLPTS	Rotina LPTSTT ($0884).
$FFC0	HSCRE	SCREEN ($79CC).
$FFC5	HPLAY	PLAY ($73E5).

Tabela 5.4: Ganchos

Capítulo 6

MSX-DOS e BDOS

O MSX-DOS é o sistema de disco do MSX, que é uma adaptação do CP/M da Digital Research para o padrão MSX. O CP/M define um conjunto padrão de operações que permitia que programas CP/M executassem sem modificação em qualquer encarnação de CP/M, permitindo o surgimento de aplicativos como SuperCalc, dBASE e Turbo Pascal para a babel de microcomputadores de 8 bits lançados na década de 1980.

6.1 BDOS

O BDOS é um conjunto de rotinas padronizadas disponíveis em MSX com interface de disco. As rotinas podem ser chamadas do MSX-DOS ou do Disk BASIC, com a diferença que no MSX-DOS é feito um `CALL $0005`, e no Disk BASIC é feito um `CALL $F37D`. Embora o BIOS não esteja presente (o MSX-DOS mapeia 64 KB de RAM), ainda é possível chamar a rotina CALSLT (Capítulo 5) no endereço $0030 (fora do DOS seu endereço seria $001C).

O número da rotina é carregado no registrador **C**, e os parâmetros são carregados nos registradores **DE** e **HL**, de acordo com a especificação da rotina.

As rotinas do BDOS não têm nome oficial, os nomes dados aqui são apenas descrições resumidas de seu propósito.

Boot *$00*

■ Provoca o reinício do sistema. Quando chamada de dentro do MSX-DOS esta rotina "limpa a casa" e retorna ao prompt do DOS. Quando executada a partir do Disk BASIC, esta rotina provoca um reinício total do sistema.

◇← Nenhum.

◇→ Nenhum.

GetCharAndEcho *$01*

■ Lê um caractere do teclado e o imprime na tela.

◇← Nenhum.
◇→ A: caractere lido.

PutChar *$02*

■ Imprime um caractere na tela. Aceita códigos de controle VT-52.

◇← E: caractere a imprimir.
◇→ Nenhum.

ReadSerial *$03*

■ Lê um caractere da porta serial. Pode ser interrompida por Control+S. Praticamente nenhum modem ou interface serial de MSX permite a leitura de caracteres por esta rotina.

◇← Nenhum.
◇→ A: caractere lido.

WriteSerial *$04*

■ Escreve um caractere na porta serial. Os modems e seriais MSX não permitem o uso desta rotina.

◇← E: caractere a escrever.
◇→ Nenhum.

WritePrinter *$05*

■ Envia um caractere para a impressora. Pode ser interrompida por Control+S.

◇← E: caractere.
◇→ Nenhum.

ReadKey *$06*

■ Lê uma tecla do teclado (sem espera). Se o registrador E for diferente de 255, imprime o conteúdo de A na tela antes de ler o teclado.

◇← E: 255=sem eco, 0-254: com eco, A: caractere a ecoar.

◇→ A: código ASCII da tecla lida, ou 0 se não houve tecla pressionada.

WaitKey *$07*

■ Espera um tecla ser pressionada.

◇← Nenhum.

◇→ A: código ASCII da tecla pressionada.

GetChar *$08*

■ Semelhante à rotina $01, porém sem eco.

◇← Nenhum.

◇→ A: caractere lido.

PutString *$09*

■ Imprime uma string terminada com $ (ASCII 36) na tela.

◇← DE: endereço da string.

◇→ Nenhum.

GetString *$0A*

■ Lê uma linha do teclado até que Enter/Return seja pressionado. O buffer apontado por DE deve ter o limite de bytes a ler no primeiro byte, o número de caracteres efetivamente lidos será colocado no segundo byte e a string lida será armazenada a partir de DE+2.

◇← DE: endereço do buffer.

◇→ Nenhum.

KeyboardStatus *$0B*

■ Verifica o teclado e retorna indicando se alguma tecla (mesmo teclas de controle) foram pressionadas.

◇← Nenhum.
◇→ A: 255 se houve tecla pressionada, 0 se não houve.

GetSysVersion *$0C*

■ Obtém a versão do DOS. No caso do MSX-DOS, retorna sempre $0022, indicando compatibilidade com o CP/M 2.2.

◇← Nenhum.
◇→ HL: versão do DOS.

RefreshDiskInfo *$0D*

■ Atualiza os dados do disquete (tipo, número de arquivos, etc.) e fecha todos os buffers (FCBs).

◇← Nenhum.
◇→ Nenhum.

SetCurrentDrive *$0E*

■ Muda o drive corrente.

◇← E: drive (1=A, 2=B, etc.)
◇→ Nenhum.

OpenFile *$0F*

■ Abre um arquivo.

◇← DE: endereço do FCB.
◇→ A: 0=arquivo existe, 255=arquivo não existe.

UpdateFile *$10*

■ Atualiza a entrada de diretório de um arquivo após a gravação de registros.

◇← DE: endereço do FCB.
◇→ Nenhum.

FindFirst *$11*

■ Procura o primeiro arquivo no diretório que combine com a descrição contida no FCB dado, que pode usar o caractere '?' como coringa.

◇← DE: endereço do FCB.
◇→ A: 0=foi encontrado (FCB colocado na área de transferência), 1=não encontrado.

FindNext *$12*

■ Procura o próximo arquivo (após o último encontrado por FindFirst ou FindNext) no diretório que combine com a descrição contida no FCB dado, que pode usar o caractere '?' como coringa.

◇← DE: endereço do FCB.
◇→ A: 0=foi encontrado (FCB na área de transferência), 1=não encontrado.

DeleteFile *$13*

■ Apaga o arquivo descrito pelo FCB dado, que pode ter coringas.

◇← DE: endereço do FCB.
◇→ A: 0=sucesso, 255=erro.

ReadBlock *$14*

■ Lê 128 bytes de um arquivo sequencial e os coloca na área de transferência.

◇← DE: endereço do FCB.
◇→ A: 0=sucesso, 1=sucesso+EOF, 2=erro.

WriteBlock *$15*

■ Grava bloco de 128 bytes (da área de transferência) em arquivo sequencial.

◇← DE: endereço do FCB.
◇→ A: 0=sucesso, 1=falta espaço em disco, 2-255: outro erro.

CreateFile *$16*

■ Cria arquivo.

◇← DE: endereço do FCB.
◇→ A: 0=sucesso, 255=erro.

RenameFiles *$17*

■ Renomeia um ou mais arquivos. O FCB de entrada deve conter o drive em FCB+$00 e nome original em FCB+$01 (podendo ter coringas) e o novo nome a partir de FCB+$11. posição.

◇← DE: endereço do FCB
◇→ A: 0=sucesso, 255=erro.

GetSysDrives *$18*

■ Devolve os drives presentes no sistema.

◇← Nenhum.
◇→ HL: bitmap de drives (bit 0 de L=drive A, bit 1 de L=drive B, ..., bit 7 de H=drive P).

GetCurrentDrive *$19*

■ Devolve o drive atual.

◇← Nenhum.
◇→ A: drive atual (1=A, 2=B, etc.)

SetBuffer *$1A*

■ Ajusta o endereço inicial da área de transferência de dados. Na inicialização o valor default é $0080.

◇← DE: endereço.

◇→ Nenhum.

GetDiskInfo *$1B*

■ Retorna informações do disquete em um drive.

◇← E: drive.

◇→ A: 255 em erro, número de setores por cluster caso contrário, BC: tamanho do setor em bytes, DE: número de clusters no disco, HL: número de clusters livres, IY: endereço da cópia da FAT em RAM, IX: endereço do BPB do disco.

As rotinas $1C a $20 não estão implementadas no MSX-DOS.

ReadRecord *$21*

■ Lê um registro de arquivo de acesso aleatório para a área de transferência. FCB+$0E (2 bytes) deve ter o tamanho do registro a ler e FCB+$21 (4 bytes) o número do registro a ser lido.

◇← DE: endereço do FCB.

◇→ A: 0=sucesso, 1=sucesso+EOF, 2=erro.

WriteRecord *$22*

■ Grava um registro (na área de transferência) em um arquivo de acesso aleatório. FCB+$0E (2 bytes) deve ter o tamanho do registro a escrever e FCB+$21 (4 bytes) o número do registro a ser escrito.

◇← DE: endereço do FCB.

◇→ A: 0=sucesso, 1=faltou espaço, 2-255: outro erro.

GetFileSize *$23*

■ Devolve o tamanho do arquivo como valor de 4 bytes a partir de FCB+$10.

◇← DE: endereço do FCB.
◇→ A: 0=sucesso, 1=erro.

ReadNextRecord *$24*

■ Incrementa o número do registro do FCB e lê um registro em arquivo de acesso aleatório.

◇← DE: endereço do FCB.
◇→ A: 0=sucesso, 255=erro.

As rotinas $25 e $26 não estão implementadas no MSX-DOS.

ReadRecords *$27*

■ Lê uma sequência de registros em arquivo de acesso aleatório.

◇← DE: endereço do FCB, HL: número de registros a ler.
◇→ A: 0=sucesso, 1=sucesso+EOF, 2-255=erro; HL: número de registros lidos.

WriteRecords *$28*

■ Grava uma sequência de registros em arquivo de acesso aleatório.

◇← DE: endereço do FCB, HL: número de registros a gravar.
◇→ A: 0=sucesso, 1=espaço insuficiente; HL: número de registros efetivamente gravados.

A rotina $29 não está implementada no MSX-DOS.

GetDate *$2A*

■ Devolve a data do sistema.

◇← Nenhum.
◇→ HL: ano, D: mês (1=Janeiro), E: dia, A: dia da semana (0=domingo).

SetDate *$2B*

■ Modifica a data do sistema.

◇← HL: ano, D: mês (1=Janeiro), E: dia.
◇→ Nenhum.

GetTime *$2C*

■ Devolve a hora do sistema.

◇← Nenhum.
◇→ H: horas, L: minutos, D: segundos, E: centésimos de segundo.

SetTime *$2D*

■ Modifica a hora do sistema.

◇← H: horas, L: minutos, D: segundos, E: centésimos de segundo.
◇→ Nenhum.

A rotina $2E (WriteVerify) não está implementada (coerentemente) no MSX-DOS.

ReadSectors *$2F*

■ Leitura direta de setores (para a área de transferência).

◇← DE: primeiro setor, H: número de setores a ler, L: drive (1=A, 2=B, etc.)
◇→ Flag C=1 se houver erro.

WriteSectors *$30*

■ Gravação direta de setores (da área de transferência).

◇← DE: primeiro setor, H: número de setores a ler, L: drive (1=A, 2=B, etc.)
◇→ Flag C=1 se houver erro.

6.2 FCB

O FCB (File Control Block) é a estrutura de dados básica usada nas rotinas do BDOS. Sua estrutura é mostrada abaixo. A única rotina que usa o FCB desrespeitando seus campos é a $17, que usa o

deslocamento $11 para armazenar o novo nome de arquivo.

Tabela 6.1: FCB

Desloc.	**Tam.**	**Campo**
$00	1	Drive (1=A, 2=B, etc.)
$01	8	Nome do arquivo.
$09	3	Extensão do arquivo.
$0C	2	Número do bloco corrente.
$0E	2	Tamanho do registro (em bytes).
$10	4	Tamanho do arquivo (em bytes).
$14	2	Data.
$16	2	Hora.
$18	8	*Reservado.*
$20	1	Registro corrente.
$21	4	Número do registro de acesso aleatório.

Tabela 6.1: FCB

6.3 BPB

O BPB (BIOS Parameter Block) retornado pela rotina $1B tem a estrutura abaixo.

Tabela 6.2: BPB

Desloc.	**Tam.**	**Campo**
$00	1	Número do drive - 1 (0=A, 1=B, etc.)
$01	1	Tipo de formatação. $F8,$F9,$FC,$FD = 3.5 SS, 3.5 DS, 5.25 SS e 5.25 DS.
$02	2	Bytes por setor.
$04	2	?? (Barbosa não sabe)
$06	1	Faces (0=simples, 1=dupla).
$07	1	Setores por cluster.
$08	2	Número de setores reservados.
$0A	1	Número de FATs.
$0B	1	Número máximo de entradas de diretório.
$0C	2	Primeiro setor da área de armazenamento.
$0E	2	Número de clusters no disco.
$10	1	Setores por FAT.
$11	2	Primeiro setor do diretório.
$13	2	Endereço da cópia da FAT em RAM.

Tabela 6.2: BPB

6.4 Sequências VT-52

A rotina $02 permite o uso de sequências VT-52 para controle da tela:

Tabela 6.3: Códigos VT-52

ASCII	**Hexa**	**Ação**
Ctrl-G	`$07`	Beep.
Ctrl-H	`$08`	Backspace.
Ctrl-I	`$09`	Tab.
Ctrl-J	`$0A`	Avança uma linha.
Ctrl-K	`$0B`	Move cursor para o origem.
Ctrl-L	`$0C`	Limpa a tela e move cursor para a origem.
Ctrl-M	`$0D`	Retorno de carro.
Ctrl-\	`$1C`	Avança cursor uma posição.
Ctrl-]	`$1D`	Retrocede cursor uma posição.
Ctrl-^	`$1E`	Move cursor para cima.
Ctrl-_	`$1F`	Move cursor para baixo.
	`$7F`	Deleta caractere e move cursor à esquerda.
Esc A	`$1B $41`	Move cursor para cima.
Esc B	`$1B $42`	Move cursor para baixo.
Esc C	`$1B $43`	Move cursor para a direita.
Esc D	`$1B $44`	Move cursor para a esquerda.
Esc E	`$1B $45`	Limpa a tela e coloca cursor na origem.
Esc H	`$1B $48`	Coloca cursor na origem.
Esc J	`$1B $4A`	Apaga até o fim da tela, não move cursor.
Esc j	`$1B $6A`	Limpa a tela e coloca cursor na origem.
Esc K	`$1B $4B`	Apaga até o fim da linha, não move cursor.
Esc L	`$1B $4C`	Insere linha acima do cursor, move o resto da tela para baixo, deixa cursor no início da nova linha.
Esc l	`$1B $6C`	Apaga até o fim da linha, não move cursor.
Esc M	`$1B $4D`	Apaga linha do cursor, move o resto da tela para linha, coloca cursor no início da próxima linha.
Esc x 4	`$1B $78 $34`	Seleciona cursor em bloco.
Esc x 5	`$1B $78 $35`	Desliga cursor.
Esc Y n m	`$1B $59 m n`	Move cursor para coluna m-32 e linha y-32.
Esc y 4	`$1B $79 $34`	Seleciona cursor sublinhado
Esc y 5	`$1B $79 $35`	Liga cursor.

Tabela 6.3: Códigos VT-52

Capítulo 7

PPI - Interface Programável de Periféricos

A PPI 8255 está presente em todos os computadores padrão MSX (desde o MSX 1). Seu papel é controlar o mapeamento de slots no espaço de endereçamento (64K) do Z80, ler o teclado, controlar o motor e saída para a unidade de fita cassete (a entrada é controlada pelo PSG), controlar o click de tecla e o led de caps lock.
A PPI é acessada através de 4 portas de I/O do Z80 ($A8-$AB), representando as portas A, B, C e a porta de modo ($AB).

7.1 Porta A ($A8)

A porta A da PPI seleciona o slot primário mapeado em cada uma das 4 páginas de 16 KB, conforme o mapa de bits abaixo.

PPI, Porta A ($A8)

Slot Prim. $C000-$FFFF		Slot Prim. $8000-$BFFF		Slot Prim. $4000-$7FFF		Slot Prim. $0000-$3FFF	
7	6	5	4	3	2	1	0

A arquitetura MSX permite que dispositivos tenham 4 subslots (ou slots secundários) por slot primário. Neste caso, cada expansor (através da sua implementação de hardware) mapeia um registrador de subslot no endereço $FFFF (página 3). Selecionar um subslot requer portanto mapear a página 3 do slot expandido e alterar o endereço $FFFF de acordo com a seleção de subslots desejadas. A BIOS do MSX já fornece rotinas para comutar slots e subslots. O mapa de bits abaixo mostra o formato do registrador de subslot.

Registrador de Subslot ($FFFF)

Subslot $C000-$FFFF		Subslot $8000-$BFFF		Subslot $4000-$7FFF		Subslot $0000-$3FFF	
7	6	5	4	3	2	1	0

7.2 Porta B ($A9)

A porta B da PPI (apenas para leitura) e contém as colunas da linha de teclado selecionada pela porta C. O teclado MSX é uma matriz de 11 linhas por 8 colunas, embora os MSX atuais usem apenas as 9 linhas (0 a 8).

PPI, Porta B ($A9)

Colunas do Teclado							
7	6	5	4	3	2	1	0

7.3 Porta C ($AA)

A porta C controla o click de tecla, led do caps lock (o sinal está sempre presente, embora alguns MSX não possuam um LED que o aproveite), saída e motor do cassete (a geração de tom do cassete é realizada por rotinas do BIOS) e seleção de linha do teclado apresentada na porta B, conforme o formato abaixo.

PPI, Porta C ($AA)

Click Tecla	CAPS LED	Saída Cas	Motor Cas	Seleção Linha Teclado			
7	6	5	4	3	2	1	0

7.4 Porta de Modo ($AB)

A porta de modo pode operar em dois modos: normal (bit 7 setado) ou set/reset da porta C (bit 7 zerado). No modo normal a porta de modo controla a direção (leitura ou escrita) e modo de operação das portas A, B e C. O hardware do MSX está preparado para trabalhar apenas na sua configuração original. O formato da porta de modo é mostrado abaixo.

PPI, Porta de Modo ($AB)

1	Modo A & C[4-7]		Dir A	Dir C 4-7	Modo B & C C[0-3]	Dir B	Dir C 0-3
7	6	5	4	3	2	1	0

Os flags de direção indicam se as portas operarão como entradas (1) ou saídas (0). A configuração normal do MSX é Portas A e C como saída e B como entrada. Os bits 6 e 5 determinam o modo de operação da porta A e dos 4 bits altos da porta C: 00=Normal (MSX), 01=Strobe e 10=bidirecional. O bit 2 controla o modo de operação da porta B e dos 4 bits baixos da porta C: 0=Normal (MSX) e 1=Strobe.
Com o bit 7 zerado, a porta de modo permite ligar ou desligar bits individuais da porta C, conforme o formato abaixo.

PPI, Porta de Modo ($AB)

0	Não utilizados			Número do bit			Set Reset
7	6	5	4	3	2	1	0

O bit 0 é o inverso do efeito desejado no bit (0=set, 1=reset). Por exemplo, as instruções abaixo acendem o led do CAPS LOCK.

```
LD  %A,  #1100
OUT $AB, %A
```

Capítulo 8

VDP 9929/9918 (MSX 1)

O VDP (Video Display Processor) do MSX 1 usa duas portas de I/O do Z80 ($98 e $99) para comunicação com a CPU, endereça RAM própria (16 KB de VRAM) e oferece 4 modos de vídeo, 2 modos texto e 2 modos gráficos.

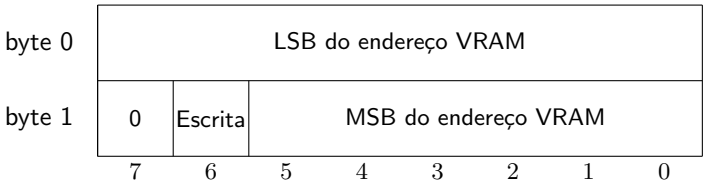
A porta $98 (porta de dados) do VDP, usada para ler e escrever a VRAM. O endereço da VRAM e o modo (escrita/leitura) é controlado através da porta de comando.

A porta $99 (porta de comando) permite preparar uma leitura ou escrita na VRAM, ler o regitrador de status ou escrever em um registrador de modo do VDP.

Ler a porta de comando (instrução IN) retorna o registrador de status do VDP (status register).

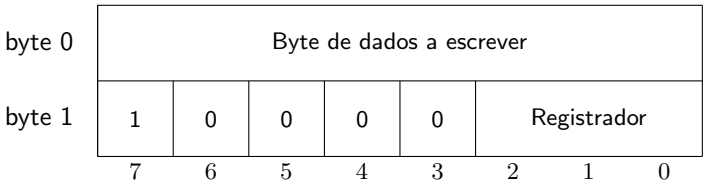
Para preparar um acesso à VRAM são enviados dois bytes à porta de comando, conforme o formato abaixo, usando o bit 6 do segundo byte para selecionar leitura (0) ou escrita (1). Após a seleção do endereço, bytes de dados podem ser lidos/escritos sequencialmente pela porta de dados (Após cada leitura/escrita o endereço é automaticamente incrementado).

Porta de comando: Preparação de Leitura/Escrita na VRAM



Para escrever em um dos 8 registradores de modo (somente escrita) é o usado o formato abaixo. É recomendável utilizar as rotinas BIOS para acesso ao VDP e à VRAM. As rotinas BIOS mantêm cópias dos registradores de modo do VDP a partir de RG0SAV (ver capítulo 5).

Porta de comando: Escrita em registrador de modo



8.1 Registrador de Status

Registrador de Status do VDP

Flag Quadro	Flag 5S	Flag Colisão	Número do Quinto Sprite				
7	6	5	4	3	2	1	0

O flag de quadro é setado ao final da varredura do vídeo (uma vez a cada 50 Hz ou 60 Hz, dependendo do MSX) e zerado automaticamente após a leitura do registrador de status, indicando portanto se esta é ou não a primeira leitura do registrador após a última varredura de tela.
O flag de colisão é setado se houver sobreposição de sprites.
O flag 5S é 1 se houver pixels de mais de 4 sprites em qualquer linha de pixels qualquer. Neste caso o número do quinto sprite é colocado nos 5 bits mais baixos do registrador. Este flag é zerado automaticamente após a leitura do registrador, de forma semelhante ao flag de quadro.

8.2 Registradores de Modo

Os registradores de modo são apenas para escrita e não podem ser lidos. Recomenda-se acessar o VDP através das rotinas BIOS, que garantem a integridade de uma cópia dos registradores de modo na área de trabalho (RG0SAV).

VDP: Registrador de Modo 0

0	0	0	0	0	0	M3	VE
7	6	5	4	3	2	1	0

O bit VE ativa (1) ou desativa (0) a entrada externa do VDP. O bit M3 é usado para determinar o modo de tela em conjunto com o registrador 1.

VDP: Registrador de Modo 1

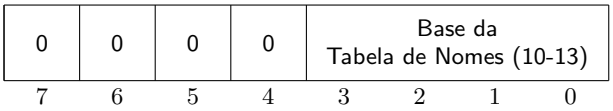
4/16K	Limpa	Int Enable	M1	M2	0	Tam. Sprites	MAG
7	6	5	4	3	2	1	0

O bit MAG define se os sprites devem ter tamanho normal (0) ou duplicado (1). O bit Tam.Sprites define se os sprites são 8×8 (0) ou 16×16 (1). Os bits M1, M2 e M3 determinam o modo de tela:

M1	M2	M3	
0	0	0	Modo texto 32×24 (screen 1)
0	0	1	Modo gráfico 256×192 (screen 2)
0	1	0	Modo gráfico 64×48 (screen 3)
1	0	0	Modo texto 40×24 (screen 0)

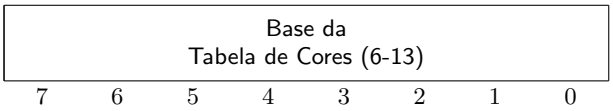
O bit Int.Enable ativa (1) ou desativa (0) a geração de interrupção do VDP. O bit Limpa ativa (1) ou desativa (0) a saída de vídeo: quando desativada a tela toda terá a mesma cor da borda. O bit 4/16K muda as características de endereçamento da VRAM para que use chips de 4K (0) ou 16K (1).

VDP: Registrador de Modo 2



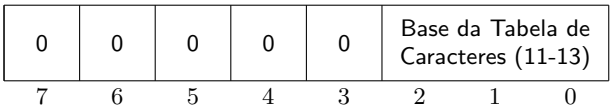
Os 4 bits (0-3) representam os bits 10-13 do endereço base. O conteúdo $0F (15, #1111) representaria o endereço $3C00, por exemplo.

VDP: Registrador de Modo 3



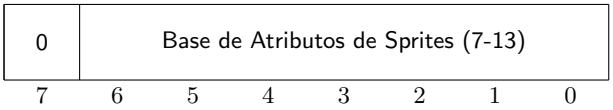
O 8 bits (0-7) representam os bits 6-13 do endereço base. O conteúdo $FF (255) representaria o endereço $3FC0, por exemplo. No modo gráfico 256×192 apenas o bit 7 tem efeito (permitindo bases $0000 e $2000) e os bits 0-6 devem estar setados.

VDP: Registrador de Modo 4



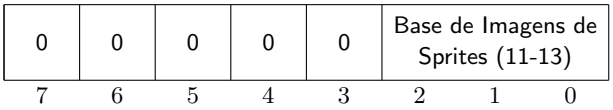
Os 3 bits (0-2) representam os bits 11-13 do endereço base. O conteúdo $07 representaria o endereço $3800, por exmeplo. No modo gráfico 256×192 apenas o bit 2 é efetivo e os bits 0 e 1 devem estar setados.

VDP: Registrador de Modo 5



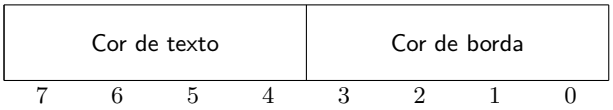
Os 7 bits (0-6) representam os bits 7-13 do endereço base. O conteúdo $7F representaria o endereço $3F80, por exemplo.

VDP: Registrador de Modo 6



Os 3 bits (0-2) representam os bits 11-13 do endereço base. O conteúdo $07 representaria o endereço $3800, por exmeplo.

VDP: Registrador de Modo 7



A cor de borda determina a cor da borda da tela e dos pixels transparentes (cor 0) em todos os modos de vídeo. Determina também a cor de fundo no modo texto 40×24.
A cor de texto determina a cor dos caracteres no modo texto 40×24. Nos outros modos não tem efeito algum.
As cores do VDP estão listadas na tabela abaixo, usando coeficientes RGB entre 0 e 1 (para obter o valor em RGB de 24 bits $(8+8+8)$ multiplique por 255).

#	Cor	R	G	B
0	Transparente	N/A	N/A	N/A
1	Preto	0.00	0.00	0.00
2	Verde Médio	0.13	0.79	0.26
3	Verde Claro	0.37	0.86	0.47
4	Azul Escuro	0.33	0.33	0.93
5	Azul Claro	0.49	0.46	0.99
6	Vermelho Escuro	0.83	0.32	0.30
7	Ciano	0.26	0.92	0.96
8	Vermelho Médio	0.99	0.33	0.33
9	Vermelho Claro	1.00	0.47	0.47
10	Amarelo Escuro	0.83	0.76	0.33
11	Amarelo Claro	0.90	0.81	0.50
12	Verde Escuro	0.13	0.69	0.23
13	Magenta	0.79	0.36	0.73
14	Cinza	0.80	0.80	0.80
15	Branco	1.00	1.00	1.00

8.3 Modo Texto 40×24

Tabela	Endereço	Tamanho
Nomes	$0000-$03BF	960
Caracteres	$0800-$0FFF	2048

A tabela de nomes contém caracteres ASCII e a tabela de caracteres contém as imagens dos 256 caracteres possíveis, usando 8 bytes por caractere, embora os 2 bits menos significativos de cada byte da tabela de caracteres sejam desprezados, mostrando caracteres 6×8 em vez de 8×8. As cores dos caracteres são definidas pelo registrador de modo 7.

8.4 Modo Texto 32×24

Tabela	Endereço	Tamanho
Caracteres	$0000-$07FF	2048
Nomes	$1800-$1AFF	768
Cores	$2000-$201F	32

O formato da tabela de caracteres é semelhante ao modo anterior, porém todos os bits são utilizados, formando caracteres 8×8. A cor da borda da tela é definida pelo registrador de modo 7, mas a cor de cada grupo de 8 caracteres é definida pela tabela de cores. Cada byte da tabela de cores define a cor de frente e fundo para um grupo de 8 códigos ASCII (0-7, 8-15, etc.), conforme o formato abaixo:

Modo 32×24: byte de dado na tabela de cores

Cor de frente				Cor de fundo			
7	6	5	4	3	2	1	0

8.5 Modo Gráfico 256×192

Tabela	Endereço	Tamanho
Caracteres	$0000-$17FF	6144
Nomes	$1800-$1AFF	768
Cores	$2000-$37FF	6144

A tabela de nomes é semelhante ao modo texto 32×24, com 768 bytes, cada um descrevendo o padrão de um bloco 8×8. No entanto, as tabelas de caracteres e cores têm 6K (o triplo do modo 32×24). Os primeiros 2K de cada tabela definem imagem (Tabela de Caracteres) e cor (Tabela de Cores) dos caracteres 0-255 no terço superior da tela (8 linhas de texto ou 64 pixels), o segundo e o terceiro trechos de 2K são usados, analogamente, para formar o segundo e terceiro terços verticais da tela. Acender ou apagar um pixel requer o cálculo do endereco e do bit a ser modificado. O BIOS provê este cálculo através da rotina MAPXYC (Capítulo 5). Embora haja maior resolução de cores que no modo 32×24, cada grupo de 8 pixels adjacentes compartilha o mesmo atributo de cor, causando borramento.

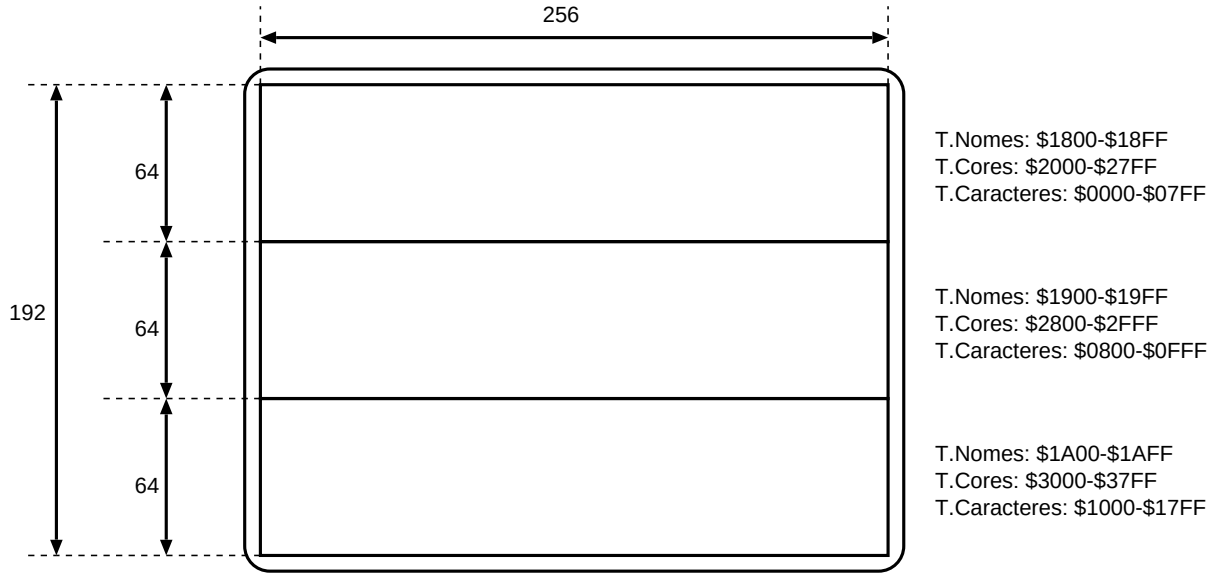


Figura 8.1: Organização do Modo Gráfico 256×192.

8.6 Modo Gráfico 64×48

Tabela	Endereço	Tamanho
Caracteres	$0000-$05FF	1536
Nomes	$0800-$0AFF	768

A tabela de nomes é inicializada com a sequência $4 \times [\$00 \ldots \$1F], 4 \times [\$20 \ldots \$3F], \ldots, 4 \times [\$A0 \ldots \$BF]$. Cada linha de 32 bytes da tabela de nomes representa uma linha 64×2 da tela. As linhas são repetidas em grupos de 4, mas cada repetição usa um dado diferente da tabela de caracteres para determinar sua cor.

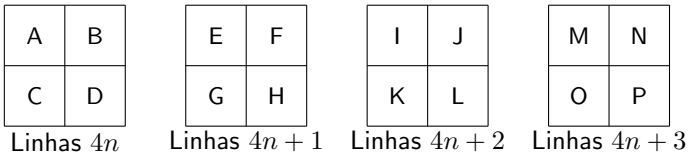
A tabela de caracteres representa as cores dos caracteres (os primeiro bloco de 8 bytes define o caractere 0, o segundo bloco define o caractere 1, etc., definindo os 192 caracteres usados, $00-$BF), mas cada caractere tem 4 padrões de cores diferentes, dependendo da linha em que aparece. Cada entrada na tabela de nomes representa 4 pixels na tela.

Por exemplo, o bloco de 8 bytes (na tabela de caracteres) abaixo:

Bloco da tabela de caracteres (Modo 64×48)

	7 6	5 4	3 2	1 0
byte 0	A		B	
byte 1	C		D	
byte 2	E		F	
byte 3	G		H	
byte 4	I		J	
byte 5	K		L	
byte 6	M		N	
byte 7	O		P	

Define os seguintes grupos 2×2 de pixels:



Onde A, B, ... , P são códigos de cor de 4 bits, e n é um inteiro de 0 a 5.

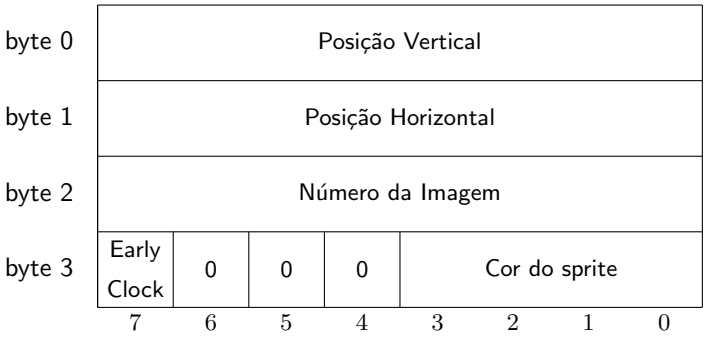
8.7 Sprites

Todos os modos de vídeo (exceto o modo texto 40×24) permitem a exibição de até 32 sprites, que são independentes dos pixels normais e não borram.

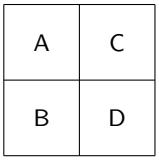
As duas tabelas que definem os sprites são a Tabela de Atributos de Sprite e a Tabela de Imagens de Sprite.

A Tabela de Atributos tem 128 bytes (32 blocos de atributos de 4 bytes cada) e é inicializada no intervalo $1B00-$1B7F da VRAM, em todos os modos de tela.

Bloco de Atributo de Sprite



A posição vertical mapeia as 192 linhas da tela de -1 a 190 (ao contrário das coordenadas normais, que variam de 0 a 191), mas pode receber valores fora do intervalo para representar sprites parcialmente fora da tela.
A posição horizontal mapeia as 256 colunas da tela de 0 a 255. Para permitir sprites entrando/saindo da tela gradualmente, o bit Early Clock pode ser setado para deslocar o sprite 32 pixels à esquerda.
O número da imagem seleciona uma das 256 imagens de sprite 8×8, ou se o VDP estiver usando sprites 16×16, os dois bits menos significativos são ignorados (pois cada 4 imagens 8×8 consecutivas formam um sprite 16×16).
A cor do sprite seleciona a cor dos bits 1 da imagem. Os bits 0 são sempre transparentes.
A Tabela de Imagens de Sprites ocupa 2048 bytes de VRAM, e é inicializada no intervalo $3800-$3FFF. Cada bloco de 8 bytes define uma imagem de sprite 8×8, de forma semelhante à formação de caracteres na tabela de caracteres do modo texto 32×24. Quando forem usados sprites 16×16 (bit Tamanho do registrador de modo 1 setado) cada 4 blocos consecutivos A, B, C e D formam um sprite como na figura abaixo.



Capítulo 9

PSG - Gerador de Som Programável

O PSG (Programmable Sound Generator) AY-3-8910 (General Instruments), presente em todo MSX, provê 3 canais de som, 3 geradores de tom, 1 gerador de ruído, gerador de envoltória e duas portas de controle usadas para controlar as portas de joystick, a entrada de dados do cassete e o modo Kana dos teclados de MSX japoneses.

O PSG tem 16 registradores internos, acessados através de 3 portas de I/O do Z80, $A0, $A1 e $A2.

9.1 Porta de Endereço ($A0)

PSG, Porta de Endereço ($A0)

0	0	0	0	Número de Registrador			
7	6	5	4	3	2	1	0

A porta de endereço seleciona um dos 16 registradores internos. Uma vez selecionado, podem ser realizadas leituras e/ou escritas através das porta de dados.

9.2 Portas de Dados ($A1 e $A2)

A porta $A1 é e porta de escrita de dados, escrever um byte nesta porta sobrescreve o registrador interno selecionado através da porta de endereço.

A porta $A2 é a porta de leitura de dados, ler esta porta tem o efeito de ler o registrador selecionado através da porta de endereço.

9.3 Registradores do PSG

Os 6 registradores de 0 a 5 controlam as frequências dos 3 geradores de tom.

PSG: Período do Gerador de Tom do Canal A

	7	6	5	4	3	2	1	0
Registrador 0	Período do Canal A (LSB)							
Registrador 1	×	×	×	×	Período do Canal A (MSB)			

PSG: Período do Gerador de Tom do Canal B

	7	6	5	4	3	2	1	0
Registrador 2	Período do Canal B (LSB)							
Registrador 3	×	×	×	×	Período do Canal B (MSB)			

PSG: Período do Gerador de Tom do Canal C

	7	6	5	4	3	2	1	0
Registrador 4	Período do Canal C (LSB)							
Registrador 5	×	×	×	×	Período do Canal C (MSB)			

Os períodos são valores entre 1 e 4095 que dividem uma frequência mestre do PSG. A frequência mestre do gerador de tom é a frequência do clock do PSG dividida por 16. A frequência do clock do PSG é metade da frequência do clock do Z80, que é de aproximadamente 3.57 MHz, variando um pouco em cada modelo de MSX. A tabela abaixo lista as frequências dos MSX brasileiros.

Modelo	**Freq. Z80**	**Freq. PSG**	**Freq. Gerador de Tom**
Gradient Expert	3575611 Hz	1787805.5 Hz	111737.8 Hz (≈ 111738 Hz)
Epcom HotBit	3579545 Hz	1789772.5 Hz	111860.8 Hz (≈ 111861 Hz)

A frequências foram tomadas de [P+87]. Para obter a frequência mestre de geração de tons basta dividir a frequência de clock do Z80 por 32.
Esta frequência mestre permite tons entre ≈27 Hz (Período 4095) e ≈111738 Hz (período 1). A faixa audível para seres humanos é (aproximadamente) de 20 a 20000 Hz, e ainda assim o som gerado estará limitado pela resposta de frequência do alto falante utilizado. Para tocar um Lá central (440 Hz) o valor do período seria 254.

PSG: Período do Gerador de Ruído

	7	6	5	4	3	2	1	0
Registrador 6	×	×	×	Período do Gerador de Ruído				

O período do gerador de ruído é um divisor entre 1 e 31 para a mesma frequência mestre dos geradores de tom.

PSG: Controle de Canais

	7	6	5	4	3	2	1	0
Registrador 7	Dir B	Dir A	Ruído C Off	Ruído B Off	Ruído A Off	Tom C Off	Tom B Off	Tom A Off

O registrador 7 controla o mixer dos três canais de som e a direção das duas portas de dados (usadas para joystick e cassete). Nos bits 6 e 7 0=Entrada e 1=Saída, e a configuração deve ser sempre Dir A=0, Dir B=1 (Reg.7 = 10××××××) para evitar danos ao hardware. Os bits 0-5 controlam o mixer dos canais, 0=ativa, 1=desativa.

PSG: Amplitude dos Canais

	7	6	5	4	3	2	1	0
Registrador 8	×	×	×	Modo Env A	Amplitude Canal A			
Registrador 9	×	×	×	Modo Env B	Amplitude Canal B			
Registrador 10	×	×	×	Modo Env C	Amplitude Canal C			

Os registradores 8-10 controlam a amplitude (volume) de cada canal. A amplitude pode ser fixa (Modo Env=0) com o valor entre 0 e 15 colocado nos bits 0-3 ou modulada pelo gerador de envoltória (Modo Env=1). Quando Modo Env=1 os valores nos bits 0-3 são ignorados.

PSG: Período da Envoltória

	7	6	5	4	3	2	1	0
Registrador 11	Período da Envoltória (LSB)							
Registrador 12	Período da Envoltória (MSB)							

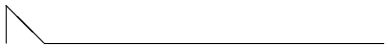
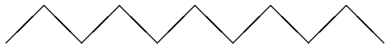
Os registradores 11 e 12 controlam o período da envoltória, um divisor entre 1 e 65535, pelo qual é dividida a frequência mestre do gerador de envoltória, que é a frequência do clock do Z80 dividida por 512 (ou a frequência mestre do gerador de tom dividida por 16).

Modelo	**Freq. Z80**	**Freq. PSG**	**Freq. Gerador de Envoltória**
Gradient Expert	3575611 Hz	1787805.5 Hz	6983.6 Hz (≈ 6984 Hz)
Epcom HotBit	3579545 Hz	1789772.5 Hz	6991.3 Hz (≈ 6991 Hz)

PSG: Forma da Envoltória

	7	6	5	4	3	2	1	0
Registrador 13	×	×	×	×	Forma da Envoltória			

O registrador 13 controla a forma da envoltória conforme a tabela abaixo.

Valor	Forma da Envoltória
00xx	
01xx	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

PSG: Leitura da Porta A

Registrador 14

CAS In	Kana Mode	Joy Bot.B	Joy Bot.A	Joy Right	Joy Left	Joy Down	Joy Up
7	6	5	4	3	2	1	0

O registrador 14 lê a porta A do PSG. Os bits 0-5 lêem o estado do joystick selecionado pelo registrador 15 (os bits 0-5 são invertidos, 0=acionado, 1=solto, é recomendável utilizar as rotinas BIOS para ler o joystick). O bit 6 indica o modo kana em teclados de MSX japoneses. O bit 7 é usado para ler a entrada do gravador cassete.

PSG: Escrita na Porta B

Registrador 15

Kana LED	Joy Select	Pulso 2	Pulso 1	1	1	1	1
7	6	5	4	3	2	1	0

O registrador 15 é usado para escrever na porta B do PSG. Os bits 0-3 estão ligados aos pinos 6 e 7 dos conectores de joystick, podendo servir como saída de dados (não faz sentido com joysticks, mas pode fazer sentido com tablets ou para usar os conectores de joystick como interface de rede). Os bits de pulso (4-5) são usados para controlar paddles. O bit 6 seleciona a porta de joystick a ser lida pela porta A/registrador 14 (0=Joy A, 1=Joy B). O bit 7 é usado para controlar o led Kana em MSXs japoneses.

9.4 Escala Cromática

A tabela abaixo representa as 8 oitavas da escala cromática (assumindo 440 Hz para o Lá central) e os valores de período apropriados para os registradores 0-5 em um MSX usando 11860.8 Hz como frequência mestre do gerador de tom.[1]

[1]Para subir uma oitava multiplica-se a frequência por 2, para descer uma oitava divide-se a frequência por 2; Para subir um semitom multiplica-se a frequência por $\sqrt[12]{2} \cong 1.05946309436$; Para descer um semitom divide-se a frequência por $\sqrt[12]{2}$.

Nota/8^a	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C	32.70 Hz $0D5C	65.41 Hz $06AE	130.81 Hz $0357	261.63 Hz $01AC	523.25 Hz $00D6	1046.50 Hz $006B	2093.00 Hz $0035	4186.01 Hz $001B
C#	34.65 Hz $0C9D	69.30 Hz $064E	138.59 Hz $0327	277.18 Hz $0194	554.37 Hz $00CA	1108.73 Hz $0065	2217.46 Hz $0032	4434.92 Hz $0019
D	36.71 Hz $0BE7	73.42 Hz $05F4	146.83 Hz $02FA	293.66 Hz $017D	587.33 Hz $00BE	1174.66 Hz $005F	2349.32 Hz $0030	4698.64 Hz $0018
D#	38.89 Hz $0B3C	77.78 Hz $059E	155.56 Hz $02CF	311.13 Hz $0168	622.25 Hz $00B4	1244.51 Hz $005A	2489.02 Hz $002D	4978.03 Hz $0016
E	41.20 Hz $0A9B	82.41 Hz $054D	164.81 Hz $02A7	329.63 Hz $0153	659.26 Hz $00AA	1318.51 Hz $0055	2637.02 Hz $002A	5274.04 Hz $0015
F	43.65 Hz $0A02	87.31 Hz $0501	174.61 Hz $0281	349.23 Hz $0140	698.46 Hz $00A0	1396.91 Hz $0050	2793.83 Hz $0028	5587.65 Hz $0014
F#	46.25 Hz $0973	92.50 Hz $04B9	185.00 Hz $025D	369.99 Hz $012E	739.99 Hz $0097	1479.98 Hz $004C	2959.96 Hz $0026	5919.91 Hz $0013
G	49.00 Hz $08EB	98.00 Hz $0475	196.00 Hz $023B	392.00 Hz $011D	783.99 Hz $008F	1567.98 Hz $0047	3135.96 Hz $0024	6271.93 Hz $0012
G#	51.91 Hz $086B	103.83 Hz $0435	207.65 Hz $021B	415.30 Hz $010D	830.61 Hz $0087	1661.22 Hz $0043	3322.44 Hz $0022	6644.88 Hz $0011
A	55.00 Hz $07F2	110.00 Hz $03F9	220.00 Hz $01FC	440.00 Hz $00FE	880.00 Hz $007F	1760.00 Hz $0040	3520.00 Hz $0020	7040.00 Hz $0010
A#	58.27 Hz $0780	116.54 Hz $03C0	233.08 Hz $01E0	466.16 Hz $00F0	932.33 Hz $0078	1864.66 Hz $003C	3729.31 Hz $001E	7458.62 Hz $000F
B	61.74 Hz $0714	123.47 Hz $038A	246.94 Hz $01C5	493.88 Hz $00E2	987.77 Hz $0071	1975.53 Hz $0039	3951.07 Hz $001C	7902.13 Hz $000E

Capítulo 10

MegaRAM e Memory Mapper

10.1 MegaRAM

A MegaRAM é uma família de cartuchos para MSX desenvolvida no Brasil por Ademir Carchano (da ACVS) e comercializada pela ACVS e pela DDX.
A MegaRAM é uma expansão de memória desenvolvida com o objetivo permitir a execução de jogos MegaROM (jogos de 128K, como Nemesis) a partir de disquetes.

10.1.1 Programação da MegaRAM

A memória da MegaRAM é dividida em blocos de 8 KB, que podem ser mapeados no slot em que se encontra a MegaRAM. Operar a MegaRAM consiste em realizar duas operações: selecionar blocos a serem mapeados e ler/escrever normalmente no slot da MegaRAM para modificar a área de memória mapeada.
Para mapear um bloco da MegaRAM, deve-se chavear a MegaRAM para o modo BLOCK-SELECT realizando uma instrução OUT para a porta $8E (o dado escrito é ignorado) e escrevendo o número do bloco desejado na área onde desejamos mapear o bloco. Embora seja possível escrever em qualquer endereço da área destino, é boa prática d programação escrever no primeiro endereço da área. Em uma MegaRAM de 256 KB há 32 blocos, numerados de 0 a 31. A maior MegaRAM comercial produzida foi de 768 KB, mas há registro de construções amadoras de 1 MB, e o máximo teórico é 2 MB (256 blocos de 8KB).
Os blocos são sempre mapeados em regiões múltiplas de $2000 (8KB) no slot da MegaRAM, mas apenas as 4 regiões $4000-$5FFF, $6000-$7FFF, $8000-$9FFF e $A000-$BFFF podem ser usadas confiavelmente em qualquer MegaRAM.
Nas MegaRAMs DDX as áreas $0000-$1FFF, $2000-$3FFF, $C000-$DFFF e $E000-$FFFF simplesmente não são utilizáveis (sempre contêm lixo), e nas MegaRAMs ACVS ocorre o mirror-effect , em que há espelhamento de regiões, ignorando o bit mais alto do endereço (a área $0000 é espelho da área $8000, a aŕea $2000 é espelho da área $A000, e assim sucessivamente).
Uma vez mapeado o bloco desejado, deve-se colocar a MegaRAM em modo WRITE-ENABLE para acessar a região mapeada, realizando uma instrução IN na porta $8E.
O trecho de código abaixo mapeia a segunda página da MegaRAM em $8000 e lê o terceiro byte da página, assumindo que a MegaRAM esteja no slot 1. Em situações reais, deve ser realizada uma busca pela MegaRAM antes de utilizá-la.

```
1   .define WRSLT $0014
    .define RDSLT $000C
```

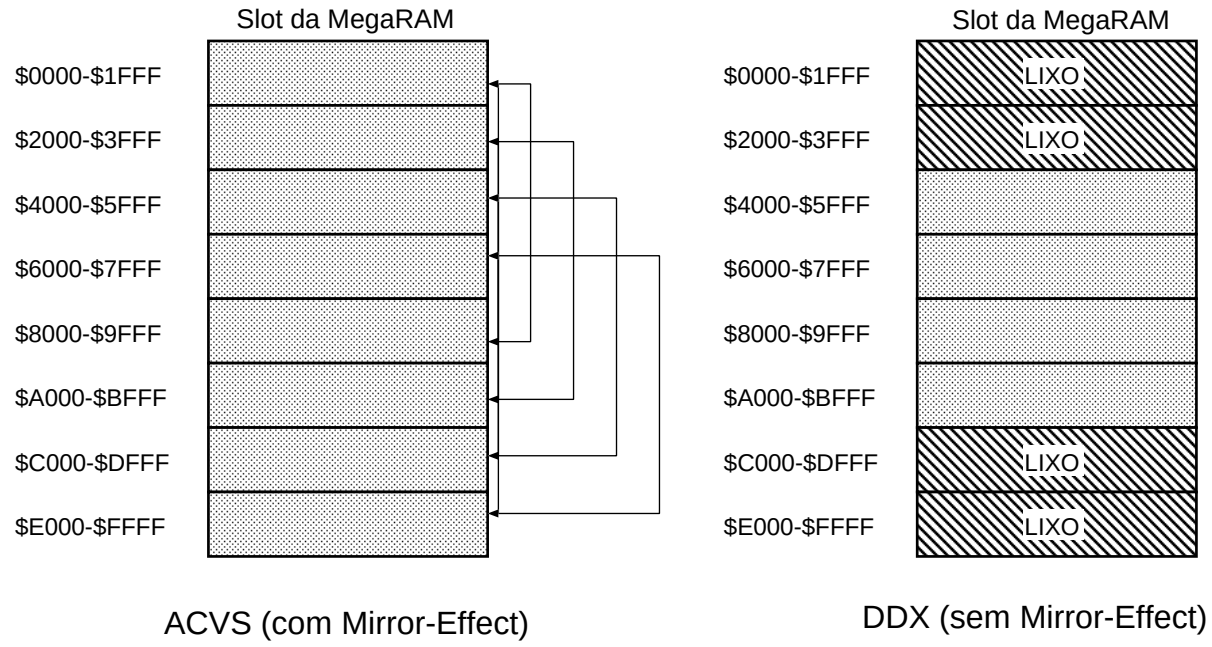


Figura 10.1: Mirror-Effect na MegaRAM

```
    ld      %b, $01     ; slot id da MegaRAM
 5  push    %bc         ; salva slot id
    out     $8e, %a     ; modo block-select
    ld      %e, 2       ; pagina 2
    ld      %a, %b      ; slot id
    ld      %hl, $8000  ; endereco de mapeamento
10  call    WRSLT       ; escreve %e no endereco %hl do slot %a
    in      %a, $8e     ; modo write-enable
    ld      %hl, $8002  ; terceiro byte da pagina
    pop     %af         ; %a <- slot id
    call    RDSLT       ; leˆ endereco %hl do slot %a e retorna em %a
```

Se for selecionado um número de página inválido, a MegaRAM faz *wrap* módulo-número de páginas, ou seja, selecionar a página 35 em uma MegaRam de 256 KB é o mesmo que selecionar a página 3 (35 mod 32).

10.1.2 MegaRAM Disk

A DDX comercializou a MegaRAM Disk, que pode atuar como uma MegaRAM normal ou como um RAM disk.

O modo MegaRAM é selecionado com uma instrução IN na porta $8F. O modo Disk é selecionado com uma instrução OUT na porta $8F. No modo disk a MegaRAM aparece como uma nova letra de drive e pode ser acessada como uma unidade de discos com as rotinas do BDOS.

10.2 Memory Mapper

A memory mapper, presente nos MSX 2, é organizada em blocos de 16 KB que podem ser mapeados em uma das 4 páginas de 16 KB do slot onde está instalada.
O número da página é escrito com uma instrução OUT nas portas $FC, $FD, $FE e $FF. (a porta $FC controla o bloco mapeado em $0000-$3FFF, a porta $FD controla o bloco mapeado em $4000-$7FFF, a porta $FE controla o bloco mapeado em $8000-$BFFF e a porta $FF controla o bloco mapeado em $C000-$FFFF)
Não é recomendável ler (instrução IN) as portas de controle da mapper. Nem todas mappers têm o circuito de tratamento do IN (as mappers externas da ACVS, por exemplo, não têm), e se houver diversas mappers de tamanhos diferentes no mesmo micro, a operação pode danificar o micro ao gerar conflitos no barramento.

Apêndice A

GNU Free Documentation License

Este livro está sob a licença GNU Free Documentation License, incluída neste apêndice. Não existe tradução oficial da licença para o português, e por isso preferi colocar o texto original em inglês. De forma bem resumida: você pode fazer quantas cópias quiser deste documento, seja em forma digital ou "árvores mortas", e você pode obter o código-fonte LaTeX deste livro em `http://foca.sf.net`.

GNU Free Documentation License

Version 1.1, March 2000

Preamble. The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and Definitions. This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".
A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could

be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.
The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.
The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.
A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".
Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.
The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

Verbatim Copying. You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.
You may also lend copies, under the same conditions stated above, and you may publicly display copies.

Copying in Quantity. If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.
If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.
If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.
It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications. You may copy and distribute a Modified Version of the Document under the conditions of sections

2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining Documents. You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Collections of Documents. You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation With Independent Works. A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

Translation. Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Termination. You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This License. The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version"applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents. To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Referências Bibliográficas

[Ava88] Avalon. *O Livro Vermelho do MSX*. Ed. McGraw-Hill, São Paulo, SP, Brasil, 1988.

[Bar90] Eduardo A. Barbosa. *Guia do Programador MSX*. Editora Ciência Moderna, 1990.

[Lev79] Lance A. Leventhal. *Z80 Assembly Language Programming*. Osborne/McGraw-Hill, Berkeley, CA, EUA, 1979.

[NW91] Peter Norton e Richard Wilton. *Novo Guia Peter Norton para Programadores do PC e PS/2*. Editora Campus, 1991.

[P+87] Pierluigi Piazzi et al. *Aprofundando-se no MSX*. Ed. Aleph, São Paulo, SP, Brasil, 5ª edição, 1987.

[RL87] Flávio Rossini e Henrique Figueredo Luz. *Linguagem de Máquina: MSX*. Ed. Aleph, São Paulo, SP, Brasil, 1987.

[You01] Sean Young. The Undocumented Z80 Documented. http://www.msxnet.org/tech/, Setembro 2001. v0.4, GNU FDL.

[Zil01] Zilog, San Jose, CA, EUA. *Z80 Family CPU User's Manual*, 2001. UM008002-0202, http://www.zilog.com.

Índice Remissivo