giovannireisnunes

Escrevendo um jogo para MSX – parte 2

05/02/201604/02/2016 / GIOVANNI NUNES

Programar em *assembly* é interagir diretamente com o hardware, mesmo que seja um "simples" joguinho. Para amenizar um pouco isto optei por preferencialmente usar as rotinas da **BIOS** do **MSX**, elas podem não garantir melhor performance mas asseguram compatibilidade, portabilidade e também agiliza a codificação já que não preciso escrever (e depurar) minhas próprias rotinas para ler o estado dos *joystick*, enviar dados para para o processador de vídeo etc.

Uma observação, esta parte deveria ter ficado junto da primeira

mas para evitar que todo o conjunto fosse elevado à categoria de TL;DR (https://en.wikipedia.org/wiki/Wikipedia:Too_long;_didn't_read) preferi dividir em duas partes. A primeira com aspectos mais conceituais e visuais (https://giovannireisnunes.wordpress.com/2016/01/22/escrevendo-um-jogo-para-msx-parte-1/) e esta aqui um pouco mais técnica.

Ocupando a memória

Diferente do que acontece nas demais linguagens de programação, em *assembly* é necessário literalmente indicar onde é que o programa ficará — a diretiva **org** ("origin"). No caso de executáveis do **MSX-DOS** a "regra" é simples: começam no endereço de memória 256 (0x100) e não há discussão, escreva o programa começando lá e o programa funcionará. 🙂

Mas há "regra" para os cartuchos? Sim, a arquitetura do **MSX** oferece duas opções: (i) usar uma faixa de 16KiB entre os endereços 0x4000–0x7FFF ou (ii) entre 0x8000–0xBFFF. Isto é feito, respectivamente, através dos pinos /CS1 e /CS2 do conector de cartuchos de um **MSX** real. — Há uma "terceira" opção combinando ambas em uma espaço total de 32KiB entre 0x4000–0xBFFF.

Por que só há duas opções? Acontece que na faixa entre 0x0000–0x3FFF está a **BIOS** (ou *firmware* pra usar um termo mais moderno) enquanto que entre 0xC000–0xFFFF fica mapeada a RAM[1]. E esta é utilizada pela **BIOS** para armazenar informações importantes como configuração de *slots*, variáveis de sistema, *hooks* etc de forma decrescente a partir do endereço 0xFFFF.

[1] Nos modelos com 8KiB de RAM ela fica mapeada entre 0xE000–0xFFFF.

Decidi que o jogo ocupará a faixa de endereços entre 0x4000–0x7FFF e também que rodaria em modelos com apenas 8KiB, então a RAM começará a ser usada a partir do endereço 0xE000 — neste caso tenho uns 4KiB disponíveis antes de começar a sobrescrever informações da **BIOS**.

Mas por que não usar a faixa de endereços entre 0x8000–0xBFFF? Eu tenho três "nobres motivos" para não fazê-lo:

1. Não usarei o **MSX-BASIC**, logo não faz sentido deixá-lo ocupando 25% do meu espaço de endereçamento. Além do mais a sobreposição é feita de forma automática pela **BIOS**;

2. Se o jogo crescer e ultrapassar o limite de 16KiB terei de obrigatoriamente usar a faixa de 0x4000–0x7FFF e
3. Rodando direto em RAM (via ExecROM (http://sourceforge.net/projects/execrom/), ODO (https://www.msx.org/pt-br/downloads/utilities/rom/odo-03) etc) em um **MSX** real ao pressionar «Reset» o computador reiniciará normalmente ao invés de voltar (assombrado) com o jogo.

Aliás, a versão em cartucho do **Flappy Bird** usa justamente a faixa entre 0x8000–0xBFFF por pura inércia. Mas pretendo corrigir esta falha de caráter dele um dia. 🙂

Como funciona um cartucho

Durante a inicialização o **MSX** mapeia a **BIOS** na faixa de endereços entre 0x0000–0x3FFF, o **MSX-BASIC** em 0x4000–0x7FFF e toda a RAM que conseguir encontrar de forma decrescente a partir de 0xFFFF até totalizar 32KiB — ao todo são três casos para modelos com 8KiB (0xE000—0xFFFF), 16KiB (0xC000-0xFFFF) e ≥32KiB (0x8000-0xFFFF) de RAM. Terminada esta tarefa ele começa a procurar ROM nas expansões (ou cartuchos se preferir) conectadas aos *slots*.

Como "não existe mágica", para ser reconhecida e inicializada pela **BIOS** elas precisam se identificar da forma correta e o padrão **MSX** especifica um cabeçalho de 16 bytes na ROM contendo todas as informações necessárias para que isto ocorra.

Os cartuchos contendo software (ou "de jogos" se preferir) são o tipo mais simples e "tudo" que precisam fazer é colocar nos 4 primeiros bytes do cabeçalho a identificação de cartucho, a *string* "AB" (igual para todos), seguido do endereço de execução do código. Daí o **MSX** desviará automaticamente a execução para aquele endereço e o software executado.

Esqueleto de um cartucho

Para exemplificar melhor um cartucho pode ser montado assim.

1	romSize: equ 8192 ; O tamanho da ROM (8192 ou
2	romArea: equ 0x4000 ; Endereço inicial da ROM (

```
3   ramArea:    equ 0xe000              ; Início da RAM (0xc000 ou 0
4
5           org romArea
6
7           db "AB"                 ; ID
8           dw startProgram         ; INIT
9           dw 0x0000               ; STATEMENT
10          dw 0x0000               ; DEVICE
11          dw 0x0000               ; TEXT
12          ds 6,0                  ; RESERVED
13
14  startProgram:
15          proc
16          ret
17          endp
18  romPad:
19          ds romSize-(romPad-romArea),0
20          end
```

view raw cartridge.asm hosted with ♥ by **GitHub**

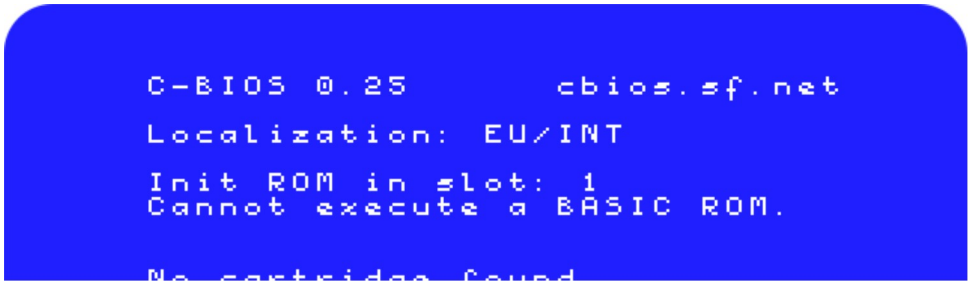
Explicações rápidas:

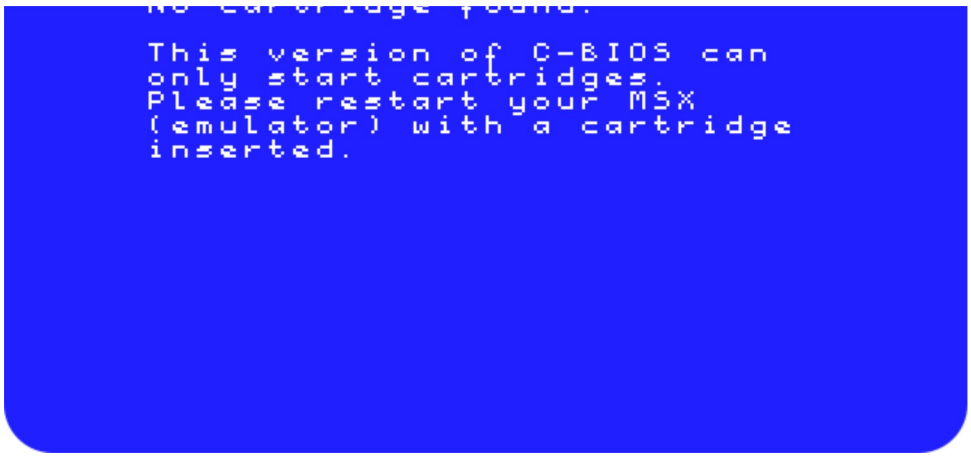
- Os rótulos **romSize**, **romArea** e **ramArea** correspondem respectivamente ao tamanho final do cartucho, o endereço inicial em ROM e o endereço inicial da RAM (que não é usado);
- Em seguida está o cabeçalho de 16 bytes contendo todos os campos descritos no padrão **MSX**;
- O "programa" está entre as diretivas **proc** e **endp** e
- Por último, a diretiva necessária para completar o arquivo com os bytes necessários para ter o tamanho correto.

Este exemplo pode ser montado com assim:

```
$ pasmo cartridge.asm cartridge.rom
```

O resultado será um arquivo ROM de exatos 8.192 bytes que faz…
…coisa alguma! 😀





A instrução “ret” (retorne) que consiste em “todo o programa” faz com que a execução volte à rotina de inicialização de expansões da **BIOS** que continuará procurando por novos cartuchos (e no caso da **C-BIOS** terminará dizendo que não achou interpretador **BASIC**).

Mas repare que o cartucho foi perfeitamente identificado (“Init ROM in slot: 1”).

Fazendo um “Hello World”

Aprimorando este esqueleto para fazer algo mais útil: um originalíssimo “Hello World!” usando rotinas própria da **BIOS**, mais precisamente **CHPUT** (que escreve um caractere na tela) e **INITXT** (que inicializa o modo texto de 40×24, a SCREEN 0) :

```
1  romSize:    equ 8192                ; o tamanho que a ROM deve t
2  romArea:    equ 0x4000               ; minha ROM começa aqui
3  ramArea:    equ 0xe000               ; inicio da área de variáveis
4
5  INITXT:     equ 0x006c
6  CHPUT:      equ 0x00a2
7
8          org romArea
9
10         db "AB"                 ; ID
11         dw startProgram         ; INIT
12         dw 0x0000               ; STATEMENT
13         dw 0x0000               ; DEVICE
14         dw 0x0000               ; TEXT
15         ds 6,0                  ; RESERVED
```

```
16
17  startProgram:
18          proc
19          call INITXT
20          ld b,71
21  writeMore:
22          ld hl,helloWorld
23  getChar:
24          ld a,(hl)
25          cp 0
26          jr z,writeLoop
27          call CHPUT
28          inc hl
29          jr getChar
30  writeLoop:
31          djnz writeMore
32  loop:
33          jr loop
34          endp
35
36  helloWorld:
37          db "Hello World!",0
38
39  romPad:
40          ds romSize-(romPad-romArea),0
41          end
```

view raw hello.asm hosted with ♥ by **GitHub**

A função do programa, além de escrever 72 vezes a mesma coisa, é demostrar que o **MSX-BASIC** pode estar "desligado" — ele foi sobreposto por mim, digo, pelo meu cartucho — mas que a **BIOS** continua acessível e que posso utilizá-la sem precisar de malabarismos.

Como ficar escrevendo "pasmo blá, blá, blá…" a toda hora é **MUITO** chato é bom criar um **Makefile** para deixar o GNU Make (https://www.gnu.org/software/make/) cuidar disto:

```
1  CAT=cat
2  ECHO=echo
3  EMULATOR=openmsx
4  INFILE=hello.asm
5  OUTFILE=hello
6  PASMO=pasmo
```

```
7  RM=rm
8
9  .PHONY: default clean superclean test
10
11 default:
12     ${PASMO} -d \
13         -v \
14         -1 \
15         --err \
16         ${INFILE} ${OUTFILE}.rom | \
17         tee ${OUTFILE}.log 2> ${OUTFILE}.err
18
19 test:
20     ${EMULATOR} ${OUTFILE}.rom &
21
22 clean:
23     ${RM} ${OUTFILE}.rom
24
25 superclean:
26     ${RM} -f ${OUTFILE}.rom ${OUTFILE}.log ${OUTFILE}.e
```

view raw Makefile hosted with ♥ by **GitHub**

Para montar o programa basta digitar **make** (e todas as mensagens produzidas durante a montagem vão para a a tela mas também ficam preservadas), para testar o código use **make test**, para apagar somente o cartucho use **make clean** e para remover todos os arquivos produzidos **make superclean**.

Fim da segunda parte

A partir deste ponto o desenvolvimento efetivamente pode começar pois o que precisava ser definido já foi, os gráficos já estão esboçados (como é pouca coisa acredito que até possam ser considerados como concluídos), as ferramentas definidas e até já se tem uma ideia da ordem a ser seguida.

Hora de começar a programar como "se não houvesse amanhã"? Ainda não, considerar um último detalhe: a organização e manutenção do código.

Mas isto fica para a próxima parte, até lá! 🙂

Programação

ASSEMBLY , **JOGOS** , **MSX** , **RETROCOMPUTARIA** , **SURVIVE** , **Z80**

Um comentário sobre “Escrevendo um jogo para MSX – parte 2”

1. Pingback: Survive — Organizando o código | giovannireisnunes

Os comentários estão desativados.