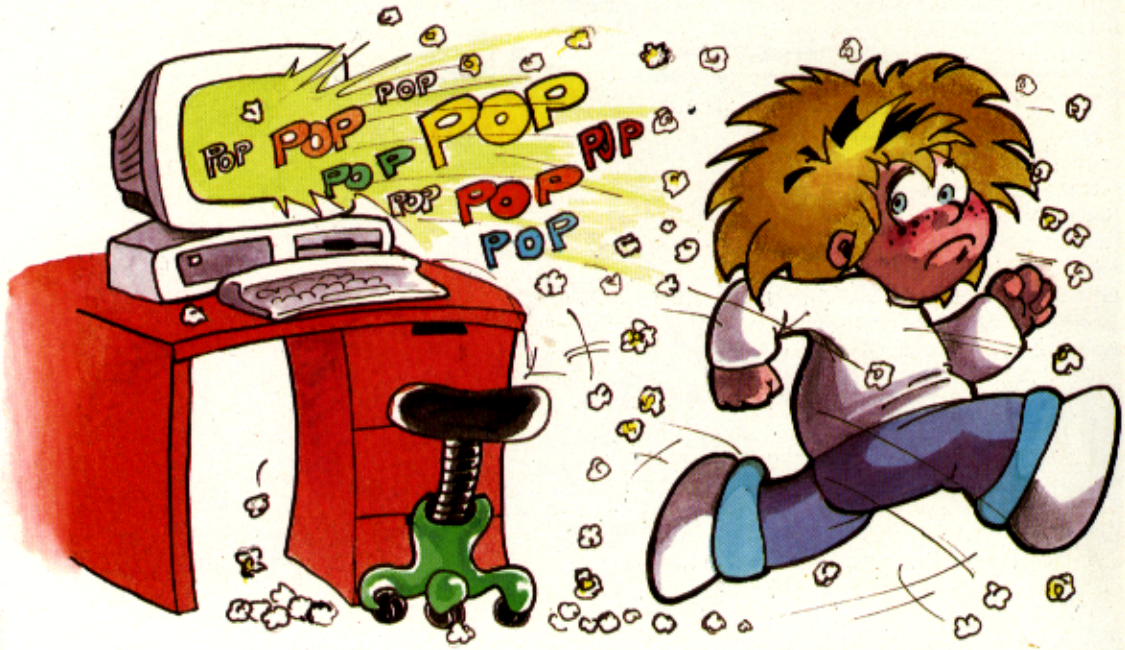
PROGRAMAR EM *ASSEMBLY* (PARTE V)



> **As dificuldades oriundas do *hardware* são explicadas pelo facto de o *Assembly* ser, acima de tudo, uma linguagem nativa da máquina em que se trabalha.**

Os leitores que nos têm acompanhado nesta pequena «viagem» pelo mundo dos PC e do *Assembly* já tiveram oportunidade de notar que manisfestámos por várias vezes a opinião de que a linguagem *Assembly* é, em si mesma, extremamente simples.

Contudo, a esta simplicidade da linguagem contrapõe-se uma certa complexidade do meio ou ambiente em que os programadores desta linguagem estão imersos, a qual dá origem a dois tipos de dificuldades que tendem a desanimar alguns dos candidatos menos determinados na sua aprendizagem.

Uma dessas dificuldades tem a ver com o próprio *hardware* dos PC e a outra com o *interface* dos nossos programas ou módulos *Assembly*, quer com o sistema operativo, quer com o *firmware* (programas residentes em ROM), quer ainda com outros módulos escritos em linguagens de alto nível.

As dificuldades oriundas do *hardware* são explicadas pelo facto de o *Assembly* ser, acima de tudo, uma linguagem nativa da máquina em que se trabalha, a qual no nosso caso é um computador ainda compatível com os «velhinhos» IBM Personal Computer (PC), cujo nascimento remonta ao início dos anos 80.

E como tomando por base o «avôzinho» IBM PC muitos descendentes foram aparecendo cada vez mais evoluídos, requintados e sofisticados, mas sempre mantendo a filosofia da compatibilidade retrógrada (*backward compatibility*) com toda a árvore genealógica, de tudo isso resultou aquilo que se pode bem considerar como um verdadeiro «molho de bróculos».

Pois que, se a história tivesse sido outra, talvez não nos tivessemos que preocupar hoje em aprender conceitos tão pouco intuitivos como **segmentação de memória, memória expandida, memória extendida, modo protegido, memória virtual** e muitos outros que tais.

E para complicar mais as coisas, à complexidade do *hardware* foi-se associando a complexidade dos sistemas operativos aos quais compete estabelecer a ponte entre os nossos programas de aplicações e os programas residentes de origem no *hardware* do computador (entre eles o célebre BIOS - *Basic Input Output System*).

O mais comum dos sistemas operativos utilizados nos PC, e o que nos está a servir de base para este estudo é, como se sabe, o MS-DOS.

E nunca é de mais insistir quanto útil é ter um conhecimento prático de muitas das inúmeras rotinas que o MS-DOS e o BIOS pôem à nossa disposição, as quais são acedidas, como vimos no último artigo, por meio de *interrupts*.

Muitas dessas rotinas estão ilustradas, com exemplos de aplicação e tudo, em muitos livros fácilmente encontrados nas secções de informática das livrarias, e a quase totalidade delas são descritas no célebre INTERRUPT.LST de Ralph Brown (ver Artigo Nº 4 desta série).

Para além das dificuldades acima citadas alguns programadores de linguagem *Assembly* são incumbidos de tarefas algumas vezes ingratas, como a de produzir módulos em *Assembly* para ligar com programas produzidos em linguagens de alto nível.

E aqui o «pobre coitado» do programador de *Assembly* necessita também de saber (no mínimo) como se processa aquilo que é designado por *subrotine interface* da linguagem de alto nível em questão. E, naturalmente, o dito «pobre coitado» não perderá nada em saber também algo dessa própria linguagem de alto nível.

Mas todas estas dificuldades que referimos são superáveis, óbviamente, com a prática, fazendo programas e mais programas, analizando as listagens de programas produzidos por outros programadores, lendo aqui e ali um ou outro livro, um ou outro artigo de revista.

Aliás esta regra geral aplicar-se-ía a quem desejasse dominar qualquer outra área do conhecimento; por conseguinte nada de sobrevalorizar o estudo da linguagem *Assembly*.

Mas o tema do nosso artigo de hoje é, tal como prometemos, **os ficheiros executáveis de estrutura EXE**. Para ilustrar este tipo de ficheiro «confeccionámos» um interessante e útil programa cujo código fonte tem o nome de ONDE.ASM.

Por limitações óbvias de espaço na revista, o ONDE.ASM apenas é incluído na disqueta **Spooler** (dentro do arquivo comprimido ASM5.LZH) quer sob a forma de código fonte, quer na sua forma de executável com o nome ONDE.EXE.

O ficheiro ONDE.EXE propõe-se pesquisar toda uma partição do disco rígido ou *drive* de disqueta à procura de qualquer nome de ficheiro, mesmo que o este contenha metacaracteres (*wildcards*).

Como muitos de nós têm uma habilidade natural para esquecer em que directório guardaram «o tal» ficheiro que faz muita falta em dada altura, daí o ONDE.EXE poder ser extremanente útil (na verdade eu próprio estou a usá-lo regularmente). Além do mais, como o ONDE.EXE foi programado totalmente em *Assembly*, ele é efectivamente tão rápido na pesquisa quanto é possível sê-lo. Para além do ONDE.ASM e ONDE.EXE fizemos incluir ainda no mesmo arquivo comprimido ASM5.LZH, dois ficheiros texto de suporte.

O primeiro deles, de seu «simpático» nome MNEMNCA.TXT, passa em revista todas as mnemónicas utilizadas dentro do programa ONDE.ASM, permitindo assim ao leitor que não disponha de outros meios de estudo a compreensão do que fazem as diversas instruções *Assembly* do nosso programa.

O segundo, baptizado de DOS_SERV.TXT dá, por seu turno, uma explicação do objectivo e sintaxe das chamadas efectuados pelo nosso programa a funções do MS-DOS via **interrupt 21H**. Recomendamos vivamente que o leitor interessado na aprendizagem da linguagem *Assembly* reserve, à partida, pelo menos algumas horas (eventualmente distribuídas por vários dias ou semanas) para o estudo detalhado do conteúdo do material contido no ASM5.LZH.

E agora é altura de darmos uma explicação sobre o que são concretamente os ficheiros de estrutura EXE e como lidar com eles na programação em *Assembly*.

Este tipo de ficheiros é caracterizado por uma separação em memória, após o seu carregamento para execução, em áreas reservadas ao código executável, aos dados e à pilha (ou *stack*). Cada uma dessas áreas é designada por **segmento de programa** e pode ocupar um máximo de 64 KB.

Se, por exemplo, um programa tiver uma área de dados muito grande, será necessário distribuir esses dados por mais de um segmento de dados. Idênticamente, se o código executável for maior que 64 KB, é necessário distribuir esse código por dois ou mais segmentos de um programa. Também se pode dar o caso de o código ou os dados ocuparem menos de 64 KB e haver interesse em dividí-los, quer um quer outro, ou ambos, por vários segmentos de programa.

Isso acontece muito em situações em que um dado programa é produzido em módulos por equipas de programadores (algumas vezes localizados fisicamente em cidades ou países diferentes). No caso do código executável estar distribuído por vários segmentos de programa, deve-se ter em atenção que as chamadas a subrotinas dentro do mesmo segmento são do tipo NEAR, enquanto as chamadas a subrotinas fora desse segmento são do tipo FAR.

Vejamos o caso seguinte de um mini-programa em *Assembly* com dois segmentos de código de nomes SEGMENTO1 e SEGMENTO2, contendo o primeiro uma rotina de nome CHAMANTE e outra de nome CHAMADAPERTO e o segundo uma rotina de nome CHAMADALONGE.

O esquema para se chamar as rotinas do SEGMENTO1 e SEGMENTO2 de dentro do SEGMENTO1 é o sequinte:

```
CODIGO1 SEGMENT
ASSUME CS:CODIGO1

CHAMANTE PROC FAR

   CALL CHAMADAPERTO
   CALL FAR PTR CHAMADALONGE
   MOV AH, 4Ch
   INT 21h

CHAMANTE ENDP

 CHAMADAPERTO PROC NEAR

        RET

 CHAMADAPERTO ENDP

CODIGO1 ENDS

CODIGO2 SEGMENT
ASSUME CS:CODIGO2

CHAMADALONGE PROC FAR

     RET

CHAMADALONGE ENDP

CODIGO2 ENDS

PILHA SEGMENT PARA STACK

DW 128 DUP (?)

PILHA ENDS
END CHAMANTE
```

Experimente passar o código fonte acima para um ficheiro texto e seguidamente assemblá-lo e ligá-lo do modo habitual, com o auxilio do MASM e LINK ou do TASM e TLINK ou ainda do A86 e LINK. Por exemplo, se designar o ficheiro texto por EXEMPLO1.ASM e estiver a utilizar o MASM faça:

```
MASM EXEMPLO1;
LINK EXEMPLO1;
```

Seguidamente utilize o DEBUG que acompanha o MS-DOS para ver o código obtido, fazendo:

```
DEBUG EXEMPLO1.EXE
U
```

Veja por si próprio como o assemblador substituiu espontaneamente na rotina CHAMADALONGE a instrução RET pela instrução RETF.

Agora faça executar o programa instrução a instrução, pressionando sucessivamente na tecla **T** de dentro do DEBUG, excepto na instrução **INT 21h** em que deverá pressionar **P** para evitar ser levado no acompanhamento da chamada ao serviço do MS-DOS.

E uma nota final sob o código fonte acima:

O leitor reparou que na chamada intersegmento da instrução CALL FAR PTR CHAMADALONGE se utilizou explicitamente um especificador de distância, o FAR

PTR (que significa *far pointer*). Isto é necessário apenas com alguns assembladores que têm menos facilidade em resolver endereçamentos localizados mais à frente no programa (*forward references*), embora com outros (como por exemplo o MASM versão 6.0 ou superior) seja redundante neste caso o especificador de distância e ter-se-ía podido pôr muito simplesmente CALL CHAMADALONGE.

Vejamos agora um outro pequeno exemplo de mini-programa em que os dados se encontram distribuídos por três segmentos de programa de nomes DADOS1, DADOS2 e DADOS3. E, para «aumentar um pouco a confusão», decidimos arbitrariamente colocar o segmento DADOS1 antes do segmento de código de nome CODIGO, o segmento DADOS2 colocámo-lo após o segmento CODIGO, e o segmento DADOS3 foi posicionado a seguir ao segmento de pilha de nome PILHA.

Este programa executa procurando um primeiro dado de nome PRIMEIRODADO no segmento DADOS1 e transfere-o para o registo **BX**. De seguida procura um segundo dado de nome SEGUNDODADO no segmento DADOS2 e passa-o para o registo **CX**.

Finalmente, **CX** é adicionado ao conteúdo do registo **BX** e o respectivo resultado é guardado na variável de nome RESULTADO que existe no segmento DADOS3.

Mas vejamos o código fonte do programa:

```
DADOS1 SEGMENT

PRIMEIRODADO DW 200h

DADOS1 ENDS

CODIGO SEGMENT

ASSUME CS:CODIGO, DS:DADOS1

MAIN PROC FAR

    MOV AX, DADOS1
    MOV DS, AX
    MOV BX, [PRIMEIRODADO]

ASSUME DS:DADOS2

    MOV AX, DADOS2
    MOV DS, AX
    MOV CX, [SEGUNDODADO]
    ADD BX,CX

ASSUME DS:DADOS3

    MOV AX, DADOS3
    MOV DS, AX
    MOV [RESULTADO], BX

    MOV AH,4Ch
    INT 21h

MAIN ENDP

CODIGO ENDS


DADOS2 SEGMENT

SEGUNDODADO DW 300h

DADOS2 ENDS

PILHA SEGMENT PARA STACK

DW 128 DUP (?)

PILHA ENDS

DADOS3 SEGMENT

RESULTADO DW (?)

DADOS3 ENDS
END MAIN
```

O leitor apercebeu-se que, sempre que nos preparávamos para «atacar» um novo segmento de dados, incluímos a directiva ASSUME para informar o assemblador que a partir daquele ponto os deslocamentos (ou *offsets*) a considerar para referenciar as variáveis seriam a partir do início do segmento de programa que «assumimos», e não a partir do anterior segmento de programa.

Mas, para além disso, o leitor verificou também que carregámos no registo de segmento **DS** o valor do novo segmento de programa de modo a podermos endereçar directamente os dados existentes no segmento de dados que passou a ser «assumido» como *default*.

Leia novamente a explicação dada nos dois últimos parágrafos sobre o fundamento das directivas ASSUME e sobre a necessidade de se carregar o registo de segmento **DS** com o valor do segmento de dados do programa que passou a ser «assumido» como *default*.

Lembre-se que muito bom programador de *Assembly* ainda «esbarra» em verdades simples como esta, mesmo ao fim de muito tempo de prática.

Tente assemblar o programa acima tal como fez com o exemplo anterior. Tudo bem? Talvez sim, talvez não.

Efectivamente, se utilizou o A86 ou versões antigas do MASM ou TASM, é possível que tenha recebido mensagens de erro e a assemblagem não se tenha realizado.

A explicação é a mesma que demos acima e tem a ver com a pouca capacidade de certos assembladores em resolver as *forward references*. O modo mais simples de resolver o problema é alterar o nosso mini-programa passando os segmentos de dados DADOS2 e DADOS3 para antes do segmento de código CODIGO. E agora a assemblagem dever-se-á processar sem dificuldade nenhuma.

O leitor que estudou o nosso exemplo do artigo anterior sobre os programas de estrutura COM – o programa RELOGIO.COM – ter-se-á apercebido da quantidade enorme de trabalho realizada pelo dito cujo, tendo em conta um tamanho de código de apenas 137 bytes.

Por seu turno, qualquer dos dois exemplos cujo código fonte está reproduzido neste artigo assemblou para um programa de estrutura EXE cujo tamanho ultrapassou sempre os 512 bytes, e somos forçados a reconhecer que em termos de trabalho útil nada foi a bem dizer realizado (especialmente se comparado com o programa RELOGIO.COM). Será então que os ficheiros de estrutura EXE são ineficientes e contém código impuro ou redundante?

Nada disso. O que se passa é que todos os ficheiros de estrutura EXE vêm acompanhados por um **EXE FILE HEADER** (ou cabeçalho dos ficheiros de estrutura EXE) cujo comprimento é (pelo menos na sua forma canónica habitual) de 512 bytes.

Esse HEADER está localizado logo no início de todos os ficheiros EXE e contém importante informação que ajuda o MS-DOS no carregamento em memória do ficheiro.

Para se visualizar com o auxílio do DEBUG o conteúdo

do HEADER é necessário primeiro fazer um RENAME (ou melhor ainda um COPY) de um dado ficheiro, mudando-lhe a extensão para uma outra que não seja EXE ou COM. Experimente com o programa ONDE.EXE, fazendo:

```
COPY ONDE.EXE ONDE.XYZ
DEBUG ONDE.XYZ
D
D
```

E o que está a ver são os primeiros 256 bytes do HEADER do ficheiro ONDE.EXE.

Para saber o seu significado refira-se à tabela abaixo:

```
OFFSET
COMPR.
DESCRIÇÃO
00h
word
Assinatura dos ficheiros EXE (4D5Ah)
02h
word
Comprimento módulo 512
04h
word
Comprimento do programa que será carregado
inicialmente em memória, arredondado por
excesso, em páginas de 512 bytes. Inclui o
HEADER.
06h
word
Número de items na tabela de relocação.
08h
word
Comprimento do HEADER em parágrafos.
0Ah
word
Memória mínima requerida acima do programa
para que este possa carregar (em parágrafos).
0Ch
word
Memória máxima desejada acima do programa
(em parágrafos)
0Eh
word
Deslocamento do segmento STACK no módulo
relativo ao início do programa (em parágrafos).
10h
word
Conteúdo de SP no início.
12h
word
CheckSum
14h
word
Conteúdo de IP no início.
16h
word
Deslocamento do segmento inicial de código
relativamente ao início do programa.
18h
word
Deslocamento do primeiro item da tabela de
relocação relativamente ao início do HEADER.
1Ah
word
Número de overlay ou 0 se programa principal.
1Ch
depende
Reservado
depende
depende
Tabela de Relocação.
```

Da nossa análise ao HEADER do programa ONDE.EXE ficamos, por exemplo, com a informação de que:

a) Se trata efectivamente de um ficheiro EXE pois a assinatura **4D5Ah** está presente no *offset* 0. Fique o leitor sabendo (se é que não sabia já) que é esta assinatura e não a extensão EXE que informa o MS-DOS que ele está perante um ficheiro EXE.

Experimente fazer o RENAME de ONDE.EXE para ONDE.COM e executá-lo com essa extensão. Verificará que a execução se processa normalmente pois o MS-DOS não se deixou enganar.

b) O comprimento do programa em páginas de 512 bytes é 4. Como o arredondamento é por excesso e no *offset* **02h** do HEADER temos indicação que existe um *overflow* de 180h bytes (384 em decimal), então o comprimento que é carregado inicialmente em memória é de 3 x 512 + 384 = 1920 bytes. Dado que este comprimento é exactamente igual ao comprimento do ficheiro ONDE.EXE, podemos tirar a conclusão brilhante de que este ficheiro não contém *overlays* internos.

Muitas outras conclusões se poderiam tirar da análise do HEADER do ficheiro ONDE.EXE mas como já se vai fazendo tarde resolvemos deixar para o leitor o seu estudo com a recomendação de que, efectivamente, vale a pena tentar fazê-lo.

Já estudámos os ficheiros de estrutura COM e EXE. O leitor que nos tem acompanhado no decorrer de todo este pequeno curso e que trabalhou todos os nossos exemplos e programas de demonstração e teve ainda a felicidade de os ter compreendido, pode-se considerar neste momento como apto a tentar fazer um pequeno programa por si próprio.

Encha-se de coragem e arrisque fazer o seu primeiro progama em linguagem *Assembly*.

Ou em alternativa experimente proceder a algumas modificações ou melhoramentos no programa ONDE.EXE. Que tal a introdução de uma subrotina que nos informe o número de ficheiros pesquisados e encontrados? Difícil? Claro que não. Tente e verá.

E para a próxima e última lição vamos abordar um tipo de ficheiros «ainda mais misteriosos» que os de estrutura EXE: os famosos *Device Drivers* que se carregam no CONFIG.SYS.

E por ser também a última lição, decidimos revelar um segredo muito especial: o método de fabrico daqueles *Device Drivers* que são simultâneamente ficheiros de estrutura EXE, isto é, que tanto podem ser carregados pelo CONFIG.SYS como pelo AUTOEXEC.BAT ou pelo *PROMPT* do MS-DOS.

Pois então fique atento ao próximo número.

José Páscoa