ASSEMBLY XIV

Aqui estou eu novamente!!! Nos textos de "SoundBlaster Programming" a gente vai precisar entender um pouquinho sobre o TURBO ASSEMBLER, então é disso que vou tratar aqui, ok?

Well... O TURBO ASSEMBLER 'compila' arquivos .ASM, transformando-os em .OBJ (sorry "C"zeiros, mas os "PASCAL"zeiros talvez não estejam familiarizados com isso!). Os arquivos .OBJ devem ser linkados com os demais módulos para formar o arquivo .EXE final. Precisamos então conhecer como criar um .OBJ que possa ser linkado com códigos em "C" e "PASCAL". Eis um exemplo de um módulo em ASSEMBLY compatível com as duas linguagens:

```
IDEAL                ; Poe TASM no modo IDEAL
MODEL LARGE,PASCAL   ; Modelo de memória...
LOCALS
JUMPS

GLOBAL  ZeraAX : PROC   ; ZeraAX é público aos outros módulos

CODESEG     ; Inicio do (segmento de) código

PROC    ZeraAX          ; Inicio de um PROCedimento.
        sub     ax,ax
        ret
ENDP                    ; Fim do PROCedimento.

END     ; Fim do módulo .ASM
```

As duas linhas iniciais informam ao TURBO ASSEMBLER o modo de operação (IDEAL), o modelamento de memória (LARGE - veja discussão abaixo!) e o método de passagem de parametros para uma função (PASCAL).

O modo IDEAL é um dos estilos de programação que o TURBO ASSEMBLER suporta (o outro é o modo MASM), e é o meu preferido por um certo número de razões. O modelo LARGE e a parametrização PASCAL também são minhas preferidas porque no modelo LARGE é possível termos mais de um segmento de dados e de código (podemos criar programas realmente GRANDES e com MUITA informação a ser manipulada!). PASCAL deixa o código mais limpo com relação ao conteúdo dos registradores após o retorno de uma função (alguns compiladores C, em algumas circunstancias, têm a mania de modificar o conteúdo de CX no retorno!). Fora isso PASCAL também limpa a pilha ANTES do retorno da procedure/função. Mas, isso tudo tem uma pequena desvantagem: Usando-se PASCAL, não podemos passar um número variável de parametros pela pilha (os três pontos da declaração de uma função C: void f(char *, ...);)!

Ahhh... Você deve estar se perguntando o que é o LOCALS e JUMPS. LOCALS diz ao compilador que qualquer label começado por @@ é local ao PROC atual (não é visivel em outros PROCs!)... Assim podemos usar labels com mesmo nome dentro de várias PROCs, sem causar nenhuma confusão:

```
; modelamento, modo, etc...
LOCALS
```

```
PROC    F1
        mov cx,1000
@@Loop1:
        dec cx
        jnz @@Loop1
        ret
ENDP

PROC    F2
        mov cx,3000
@@Loop1:
        dec cx
        jnz @@Loop1
        ret
ENDP
;... O resto...
```

Repare que F1 e F2 usam o mesmo label (@@Loop1), mas o fato da diretiva LOCALS estar presente informa ao assembler que elas säo diferentes!

Já JUMPS resolve alguns problemas para nós: Os saltos condicionais (JZ, JNZ, JC, JS, etc..) säo relativos a posição atual (tipo: salte para frente tantas posiçöes a partir de onde está!)... Em alguns casos isso pode causar alguns erros de compilação pelo fato do salto näo poder ser efetuado na faixa que queremos... ai entra o JUMPS... Ele resolve isso alterando o código para que um salto incondicional seja efetuado. Em exmplo: Suponha que o label @@Loop2 esteja muito longe do ponto atual e o salto abaixo näo possa ser efetuado:

```
    JNZ     @@Loop2
```

O assembler substitui, caso JUMPS esteja presente, por:

```
    JZ      @@P1
    JMP     @@Loop2     ; Salto absoluto se NZ!
@@P1:
```

A linha seguinte do exemplo inicial informa ao assembler que o PROCedimento ZeraAX é público, ou GLOBAL (visível por qualquer um dos módulos que o queira!). Logo após, a diretiva CODESEG informa o inicio de um segmento de código.

Entre as diretivas PROC e ENDP vem o corpo de uma rotina em assembly. PROC precisa apenas do nome da função (ou PROCedimento). Mais detalhes sobre PROC abaixo.

Finalizamos a listagem com END, marcando o fim do módulo em .ASM.

Simples, né?! Suponha agora que você queira passar um parametro para um PROC. Por exemplo:

```
; Equivalente a:
;   void pascal SetAX(unsigned v) { _AX = v; }
;   PROCEDURE SetAX(V:WORD) BEGIN regAX := V; END;
IDEAL
```

```
MODEL LARGE,PASCAL
LOCALS
JUMPS

GLOBAL SetAX : PROC

PROC    SetAX
ARG     V : WORD
        mov     ax,[V]
        ret
ENDP

END
```

Hummmm... Surgiu uma diretiva nova. ARG especifica a lista de parametros que deverá estar na pilha após a chamada de SetAX (ARGumentos de SetAX). Note que V está entre colchetes na instrução 'mov'... isso porque V é, na verdade, uma referência à memória (na pilha!) e toda referência à memória precisa ser cercada com colchetes (senäo dá um baita erro de sintaxe no modo IDEAL!). Depois da compilação o assembler substitui V pela referência certa.

Os tipos, básicos, válidos para o assembler säo: BYTE, WORD, DWORD... Näo existe INTEGER, CHAR como em PASCAL (INTEGER = WORD com sinal; assim como CHAR = BYTE com sinal!).

Para finalizar: Em um único módulo podem existir vários PROCs:

```
IDEAL               ; modo IDEAL do TASM
MODEL LARGE, PASCAL ; modelamento de memória...
LOCALS
JUMPS

; ... aqui entra os GLOBALS para os PROCs que vc queira que
;     sejam públicos!

CODESEG     ; Começo do segmento de código...

PROC    P1
    ; ... Corpo do PROC P1
ENDP

PROC    P2
    ; ... Corpo do PROC P2
ENDP

;... outros PROCs...

END     ; Fim da listagem
```

Existem MUITOS outros detalhes com relação do TASM... mas meu objetivo no curso de ASM é a mixagem de código... pls, alguma dúvida, mandem mensagem para cá ou via netmail p/ mim em 12:2270/1.