

TÉCNICAS DE PROGRAMACIÓN

MODULAR

EN

ENSAMBLADOR DEL 80x86

Mariano Raboso Mateos

INTRODUCCIÓN.-

La programación modular es la técnica que consiste en hacer independientes una serie de módulos de programa, de tal forma que el programa resultante esté formado por un módulo principal y varios secundarios que se enlazan con el anterior.

Los módulos pueden ser codificados en el mismo fichero fuente que el programa principal, o pueden estar en ficheros fuente independientes. En este caso cada uno de los módulos se compila o ensambla (según el caso) de forma independiente, obteniéndose el módulo objeto correspondiente. Con el linkador se enlazan todos juntos para formar un sólo fichero ejecutable.

El contenido de los módulos secundarios suele ser estar organizado en procedimientos y funciones. De esta forma un mismo procedimiento puede ser utilizado en cualquier otro programa, formándose así una librería de funciones y procedimientos.

No sólo se pueden utilizar los procedimientos de un módulo, sino que también se puede hacer referencia a las variables declaradas en cada uno de ellos. No obstante esto no es recomendable, utilizándose como vía de comunicación con los procedimientos, el paso de parámetros por la pila.

Una de las mayores ventajas de la programación modular, en este caso en ensamblador, es la de poder hacer llamadas a rutinas escritas en ensamblador desde un lenguaje de alto nivel. De esta forma se consigue acceder a recursos del sistema que no eran accesibles desde Pascal o C. Aún en el caso de que sí lo fueran, seguramente se ejecutarán de una forma mucho más rápida y eficiente que la opción que nos proporciona el lenguaje de alto nivel.

En los siguientes apartados se hablará de cada una de las opciones disponibles a la hora de construir módulos de programa en ensamblador.

1.- LLAMADAS A PROCEDIMIENTOS

En Ensamblador, procedimientos y funciones no se distinguen en cuanto a la declaración, si en cuanto a que la forma de codificación será diferente.

Una función devuelve siempre un valor, que de una forma estándar irá almacenado en el registro acumulador.

Los procedimientos y funciones se construyen de la siguiente forma:

```
nombre      PROC      [NEAR] [FAR]
```

```
    cuerpo del procedimiento
```

```
nombre      ENDP
```

Los procedimientos siempre estarán en el segmento de código y la última instrucción del mismo será la de retorno *RET* para la familia del 8086.

Si el procedimiento va a ser llamado desde otro segmento de código, se deberá declarar de tipo *FAR*, pudiéndose especificar *NEAR* o no en caso contrario.

En la fase de ensamblado, automáticamente se sustituye la instrucción *RET* por *RETF* si el procedimiento es lejano.

La llamada a un procedimiento se hace de a través de la instrucción *CALL*:

```
call  nombre_proc
```

PASO DE PARÁMETROS POR LA PILA

Aunque la comunicación con un procedimiento se puede hacer a través de variables globales, la opción más indicada es la de paso de parámetros, bien a través de registros o por la pila. Sin duda es la única forma de trabajar cuando los procedimientos se encuentran en módulos independientes.

Respecto a la primera opción, el utilizar los registros suele quedar reducido a las llamadas a interrupciones BIOS o DOS, puesto que éstas suelen forzar a su uso.

Sí es mucho más eficaz el paso de parámetros a través de la pila. Este es el procedimiento estándar que utilizan los procedimientos y funciones en lenguajes de alto nivel.

Tiene las dos variantes ya conocidas: valor y referencia.

Paso de parámetros por valor.

En el caso de los parámetros por valor, lo que se introduce en la pila es simplemente el valor del parámetro. En el ejemplo siguiente se introduce el registro AX en la pila, cuyo contenido corresponde con el valor del parámetro. También se muestra la forma standard de recoger ése parámetro.

```
mov ax,valor ; valor del parámetro  
push ax     ; se introduce el parámetro en la pila
```


Paso de parámetros por referencia.

En este caso lo que se introduce en la pila no es un valor, sino la dirección de una variable. Hay que tener en cuenta que la dirección corresponde con un offset respecto al valor del segmento de datos. En el ejemplo se pasa como parámetro la dirección (offset) de la variable contador.

```
---  
lea ax,contador ; obtenemos dirección de contador  
push ax         ; y la introducimos en la pila como parámetro  
call nom_proc  ; llamada al procedimiento  
---
```

En este caso la disposición de la pila es la misma que antes. La diferencia es que el valor introducido en la pila es la dirección de una variable. La dirección puede incluir también el valor de un segmento:

```
---  
mov ax,seg_tabla ; obtener segmento  
push ax  
lea ax,tabla     ; obtener desplazamiento  
push ax         ; en la pila segmento:offset  
call nom_proc   ; llamada al procedimiento  
---
```

Para obtener el valor de la variable se suele recoger la dirección en un registro y utilizar un direccionamiento indirecto:

```
ejemplo      proc   far  
              push bp  
              mov  bp,sp  
              mov  di,6[bp] ; recoger el offset de la variable  
              mov  ax,8[bp] ; recoger el segmento  
              mov  es,ax    ; cargar registro segmento extra  
              mov  ax,es:[di] ; direccionar la variable  
              ----  
              ret 4  
ejemplo      endp
```

VALORES DEVUELTOS POR UNA FUNCIÓN

Aunque los parámetros por referencia nos devuelven los resultados del procedimiento, a veces es necesario recoger un valor que no tiene que ver con los parámetros introducidos. Es el caso de la llamada a una función, que aunque no se distingue en ensamblador de un procedimiento en cuanto a la declaración, sí en el uso.

En este caso, el valor de vuelta siempre se almacena en registros de la CPU. El registro por defecto es el acumulador. Si el tamaño del valor de vuelta sobrepasa el del acumulador, se suelen emplear otros registros adicionales.

Debido a que ésta es la solución empleada con las funciones de alto nivel, se detallarán más adelante en el resumen de tipos de datos de vuelta y ubicación correspondiente.

2.- LLAMADAS DESDE OTRO MODULO EN ENSAMBLADOR

Este es sin duda el caso más sencillo. Se limita a hacer llamadas a rutinas escritas en ensamblador, desde un módulo principal escrito en el mismo lenguaje. Hay dos características fundamentales a tener en cuenta:

Por un lado, cualquier procedimiento de otro módulo al que se haga referencia deberá ser declarado como externo. Para ello se utiliza la directiva del ensamblador *EXTRN*, seguida del símbolo y tipo. En el ejemplo se declara el procedimiento externo inicializa, de tipo far.

```
EXTRN nombre:tipo [, nombre:tipo] ...
```

```
Ej:   extrn inicializa:far
```

En el caso de las variables ocurre lo mismo. En el ejemplo se declara externa la variable modo de tipo byte:

```
extrn modo:byte
```

El cuadro completo de tipos es el siguiente:

Descripción	Tipos posibles
Especificador de distancia	NEAR, FAR
Especificador tamaño	BYTE, WORD, DWORD, FWORD, QWORD, TBYTE
Especificador de constantes	ABS

Por otra parte, en cada módulo hay que declarar como públicos los procedimientos y variables que se quiera puedan ser utilizados por otros módulos. Esto se hace a través de la directiva *PUBLIC*, seguida del nombre de la variable o procedimiento:

```
PUBLIC nombre [,nombre] ...
```

```
Ej:   public modo, inicializa
```

Un aspecto importante es la posibilidad de fusionar varios segmentos del mismo tipo. Sería el caso de formar un sólo segmento de datos a partir de cada uno de los segmentos correspondientes de los módulos.

Para conseguirlo, se emplea la directiva *GROUP*. La sintaxis es la siguiente:

```
nombre_seg_grupo   GROUP           seg1 [,seg2] ...
```

Si en cada módulo tenemos un segmento llamado datos y se quieren fusionar en otro llamado DGROUP, se debería incluir en cada módulo la directiva:

```
dgroup GROUP datos
```

Una vez contruidos todos los módulos, se ensamblan por separado y se linkan o enlazan, especificando el nombre del fichero ejecutable final, que suele coincidir con el del módulo principal.

```
link   mod1.obj + mod2.obj + ...
```

3.- LLAMADAS DESDE UN PROGRAMA DE ALTO NIVEL

Otra de las ventajas de la programación modular consiste en poder utilizar desde un programa en alto nivel como Pascal o C, procedimientos y funciones escritos en lenguaje ensamblador.

Por una parte se podrá acceder a recursos del sistema a bajo nivel que antes no podían ser utilizados. Si ésto ya era posible, por ejemplo en el caso de C, las rutinas seguramente se ejecutarán de una forma más eficiente.

LLAMADAS A PROCEDIMIENTOS

En el caso de los procedimientos será necesario en el programa fuente declararlos como externos. Los parámetros se definirán normalmente.

Para ello se suele emplear la directiva *EXTERNAL* (Turbo Pascal) o en otros casos *EXTERN* (Turbo C, Pascal IBM).

En el siguiente ejemplo se declara en Turbo Pascal un procedimiento para borrar la pantalla:

```
procedure borra_pan( modo:word ): External;
```

Es recomendable incluir la directiva `{$L modulo.obj}` (en Turbo Pascal) para indicar al enlazador el módulo donde están escritos los procedimientos en Ensamblador. Puede variar para otros compiladores.

Otra característica a tener en cuenta es el tipo de llamada al procedimiento: lejana o cercana. Esto significa que en el salto variará el segmento de código (lejana o far) o no (cercana o near). Es imprescindible conocer esta característica para poder retornar de la llamada correctamente.

Suele indicarse con una opción del compilador, aunque se puede forzar con una directiva. En el caso de Turbo Pascal es:

```
{$F+}
```

Un ejemplo de un módulo en ensamblador sería:

```
;definición de procedimientos públicos
      public borra_pan
assume cs:code
      code segment
;procedimiento que borra la pantalla. Admite como parámetro el modo de inicialización de la pantalla
      borra_pan proc
          push ds          ; salvar los
          push ax         ; registros
          push bp         ; afectados

          mov bp,sp
          mov al,08[bp]   ; el parámetro en SS:[BP+8]
          mov ah,0        ; borrar pantalla. Modo en AL
          int 10h         ;

          pop bp          ; recuperar
          pop ax          ; los registros
          pop ds          ; afectados

          ret 2           ; retornar con 1 parámetro ( 2 bytes )
      borra_pan endp
code ends
```

end

Si la llamada hubiera sido lejana:

```
borra_pan proc far
    ---
    mov al,0ah[bp]    ; el parámetro en SS:[BP+10]
    ---
borra_pan endp
```

LLAMADAS A FUNCIONES

En el caso de las funciones la declaración es igual que para los procedimientos, sólo que habrá que añadir el tipo del valor devuelto.

El siguiente ejemplo muestra la declaración de dos funciones en Turbo Pascal y el código en Ensamblador de una de ellas:

```
program ejemplo (input,output);
{$F+}

function power_2(a:integer;b:integer):integer; External;
function fact(n:integer):word; External;

{$L modulo.obj}

BEGIN

writeln('5 * 2^3 = ',power_2(5,3));
writeln('el factorial de 8 es ',fact(8));

END.
```

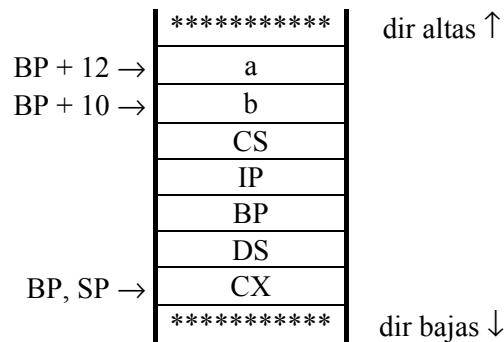
; función que calcula $a \cdot 2^b$. El resultado se devuelve en AX
power_2 proc far

```
    push bp
    push ds
    push cx
    mov bp,sp

    mov ax,[bp+0ch] ; se recoge el parámetro a
    mov cx,[bp+0ah] ; se recoge el parámetro b
    shl ax,cl

    pop cx
    pop ds
    pop bp
    ret 4
power_2 endp
```

La llamada, como se puede observar será de tipo far, luego la pila en la llamada queda como sigue:



LLAMADAS DESDE UN PROGRAMA ESCRITO EN C

Las llamadas desde C varían sensiblemente de las de Pascal. Lo primero que hay que tener en cuenta es que el orden de los parámetros es el inverso. Se leen de derecha a izquierda en la declaración de la función o procedimiento.

Otra característica es que el compilador de C añade al principio del nombre de la función o procedimiento el carácter de subrayado, por lo que es necesario hacerlo también en el módulo ensamblador.

La última es que el retorno de la función se debe hacer simplemente con *RET*, sin parámetros. El compilador de C se encarga de añadir el código necesario para eliminar los parámetros de la pila:

Test(i , j , 1); se traduciría en:

```

mov ax,1
push ax
push WORD PTR DGROUP:_j
push WORD PTR DGROUP:_i
call NEAR PTR_Test
add sp,6

```

El nº de bytes ocupado en la pila y la ubicación de los valores devueltos por cada tipo (en C) es el siguiente:

Tipo	Número de bytes
char , signed char , unsigned char	2
short , signed short , unsigned short	2
int , signed int , unsigned int	2
long , unsigned long	4
float	4
double	8
long double	10
puntero cercano (desplazamiento)	2
puntero lejano (segmento y desplazamiento)	4

Tipo	Registro
char , unsigned char	AX
short , unsigned short	AX
int , unsigned int	AX
long , unsigned long	DX:AX
estructura o unión	DX:AX (dirección)
puntero cercano	AX
puntero lejano	DX:AX

Podemos utilizar la siguiente variación que se encarga de adaptar el código a las exigencias del C:

```

dosseg
.MODEL small
.CODE

        _borra_pan proc

                push bp
                push ds
                push ax

                mov bp,sp
                mov al,08h[bp]
                mov ah,0
                int 10h

                pop ax
                pop ds
                pop bp

                ret
        _borra_pan endp
PUBLIC _borra_pan
end

```

Como se puede observar, la instrucción de retorno no va acompañada de ningún parámetro. La directiva `dosseg` nos permite utilizar una serie de segmentos por defecto como es `CODE`, `DATA`, `STACK`. La directiva `MODEL` especifica el modelo de memoria a usar, en este caso `small`.

A continuación aparece un listado de un módulo completo para ser utilizado desde C. Para obtener el ejecutable ejecutar (Turbo C):

```
tcc prin.c modulo.asm
```

donde `prin.c` es el fichero fuente en C y `modulo.asm` en ensamblador. Hay que tener en cuenta que `tcc` llamará al fichero `tasm.exe` para ensamblar el fichero `modulo.asm`.

Es posible primero ensamblar `modulo.asm` y linkar con `prin`:

```
tasm /ml modulo.asm      ( /ml para activar case-sensitive )
tcc prin.c modulo.obj
```

```

_TEXT segment byte public 'CODE'
assume cs:_TEXT

    _power_2 proc
        push bp
        push ds
        push cx
        mov bp,sp

        mov cx,[bp+0ah]
        mov ax,[bp+8]
        shl ax,cl

        pop cx
        pop ds
        pop bp
        ret
    _power_2 endp

    _factorial proc
        push bp
        push ds
        push bx
        mov bp,sp

        mov ax,08h[bp]
        cmp ax,1
        jz fin
        mov bx,ax
        dec ax
        push ax
        call _factorial
        add sp,2
        mul bx

    fin:    pop bx
           pop ds
           pop bp

           ret
    _factorial endp

    _borra_pan proc
        push bp
        push ds
        push ax

        mov bp,sp
        mov al,08h[bp]
        mov ah,0
        int 10h

        pop ax
        pop ds
        pop bp
        ret
    _borra_pan endp

PUBLIC _borra_pan
PUBLIC _power_2
PUBLIC _factorial
_TEXT ends
end

```

4.- CREACIÓN DE LIBRERÍAS

Una de las mayores ventajas de utilizar diferentes módulos, es la creación de librerías. Una librería está formada por un conjunto de módulos, que contienen una serie de referencias públicas de (procedimientos y variables).

Cuando se tiene una colección de módulos objeto relacionados, se puede crear una librería para guardar todos ellos juntos. Al crear la librería también se creará un diccionario donde se guardan las referencias públicas y los módulos que las contienen. El diccionario forma parte del fichero de librería. Se puede añadir o quitar todos los módulos que se quieran.

A continuación se describe el procedimiento para crear una librería a partir del módulo en ensamblador anteriormente listado:

- Ensamblar el módulo fuente en ensamblador:

```
tasm /ml modulo.asm
```

La opción /ml hace que se distingan mayúsculas de minúsculas. Es imprescindible en C.

- Añadir el módulo a la librería:

```
tlib /E /C mylib.lib +modulo.obj ,mylib.lst
```

mylib.lst es un fichero que muestra el contenido de la librería.

mylib.lst:

```
Publics by module
```

```
MODULO      size = 62
  _borra_pan      _factorial
  _power_2
```

/E para crear un diccionario extendido, para trabajar con mayor rapidez.

/C para hacer que la librería sea case-sensitive.

- Compilar el fichero, sin necesidad de declarar externos los procedimientos de la librería.

```
tcc ppal.c mylib.lib
```

ppal.c:

```
#include <stdio.h>
```

```
void borra_pan(unsigned int modo);
int factorial(unsigned int i);
unsigned int power_2(unsigned int i, unsigned int j);
```

```
main()
{
  borra_pan(3);
  printf("el factorial de 4 es %d\n", factorial(4));
  printf("3 * 2^5 , es %d", power_2(3,5));
}
```

sólo se enlazarán los módulos que se necesiten. En este caso sólo modulo

5.- LLAMADAS EN ENSAMBLADOR A REFERENCIAS EN ALTO NIVEL

Respecto al módulo en ensamblador, únicamente hará falta definir externos los procedimientos, funciones o variables a utilizar. Esto se hace a través de la directiva `extrn`, como ya se ha visto.

En el módulo escrito en alto nivel, el procedimiento se ha de declarar de tipo público.

En el siguiente ejemplo se construye una función en C (borrar), que borra la pantalla a través de `clrscr()`. También se ha definido un procedimiento externo (`borra_pan`), que está escrito en un módulo en ensamblador. Este procedimiento, lo único que hace es llamar a la función en C `borra`.

También se ilustra el acceso a la variable `modo`.

Es un caso algo enrevesado y meramente ilustrativo.

prin.c:

```
#include <stdio.h>

int modo;
void borra_pan();
void borrar();

main()
{
    borra_pan();
    printf("%d",modo);
}

void borrar()
{
    clrscr();
}
```

mod.asm:

```
; modulo para usar en 'C'
; ensamblar tasm /ml mod.asm para sensible a mayúsculas-minúsculas

extrn _modo:word

_TEXT segment byte public 'CODE'
assume cs:_TEXT
    _borra_pan proc
        mov modo,77
        call _borrar
        ret
    _borra_pan endp

extrn _borrar:near
PUBLIC _borra_pan
_TEXT ends
end
```