Taller de ASM con FASM
by E0N

INTRODUCCIÓN:

Bienvenidos al taller de ASM. Abril Negro 2008.

En este taller vamos a aprender a programar en ASM :P. Usaremos FASM como ensamblador ya que es gratuito y open source, lo podéis descargar de su página oficial: http://flatassembler.net/download.php . Así mismo nuestras aplicaciones serán de 32bits y correrán en procesadores x86.

¿QUÉ ES ASM?

Bueno, como no quiero agobiaros con mucha teoría que casi todo el mundo conocerá simplemente cito de la wikipedia, que esto es un taller ¡nos interesa la práctica!

*El **lenguaje ensamblador** es un tipo de lenguaje de bajo nivel utilizado para escribir programas informáticos, y constituye la representación más directa del código máquina específico para cada arquitectura de computadoras legible por un programador.*

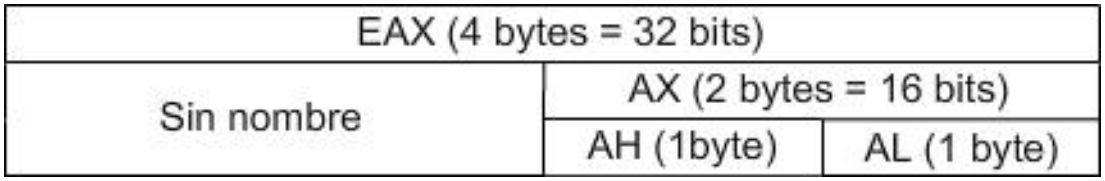
Fue usado ampliamente en el pasado para el desarrollo de software, pero actualmente sólo se utiliza en contadas ocasiones, especialmente cuando se requiere la manipulación directa del hardware o se pretenden rendimientos inusuales de los equipos.

Convendría que os leyerais lo que pone en la wikipedia, por enteraros un poco de lo que trata el taller.

LOS REGISTROS DEL MICROPROCESADOR:

ASM es totalmente diferente a cualquier otro lenguaje de alto nivel que podáis conocer. Al usar ASM nos valemos directamente de los registro de nuestro microprocesador. No voy a explicar para que valen todos y cada uno de los registros, simplemente os indico los más representativos, que nos sobrarán para diseñar cualquier programa:

EAX, EBX, ECX, EDX: Estos cuatro registros serán los que usemos principalmente para almacenar datos. Tienen el tamaño de un DWORD, es decir 4 bytes o lo que es lo mismo 32 bits. A su vez los podemos dividir en dos para referirnos a datos del tamaño de un WORD o 2 bytes y a su vez en dos para almacenar datos que ocupen un único byte. Para que quede más claro os dejo un dibujo de cómo queda dividido el registro y el nombre que toma:



En la imagen e puesto el registro EAX de ejemplo, pero esto mismo es aplicable a los otros tres cambiando la A por una B, una C o una D según corresponda.

EDI, ESI: Al igual que los cuatro anteriores los usaremos para almacenar datos, aunque tienen otras funciones. También podemos referirnos a ellos como SI o DI si queremos modificar sus 2 bytes más bajos.

EIP: Este registro contendrá la dirección de la siguiente instrucción que se ejecutará.

EBP, ESP: El registro EBP apunta a la **B**ase de la pila y el E**S**P apunta a la parte **S**uperior de la pila. La pila no es más que una estructura de tipo LIFO (http://es.wikipedia.org/wiki/LIFO) que nos servirá para pasar parámetros a las funciones o almacenar datos entre otras cosas. Más adelante todo esto se explicará con mayor claridad, pero para que os hagáis una idea, cuando vosotros en C o en otro lenguaje llamáis al api MessageBox por ejemplo en realidad pasa esto:

Llamada a MessageBoxA en C:
MessageBoxA (0, "Hola mundo", "xD", 0);

Pila:

Código equivalente en ASM:

```
push 0
push Puntero_xD
push Puntero_HolaMundo
push 0
call[MessageBoxA]
```

Lo último que se mete en la pila con el call es la dirección de retorno, es decir, el valor que tomará el registro EIP al salir de la función para que el programa siga su curso normal. Si no entendéis muy bien esto ahora mismo no os preocupéis, luego se irá explicando todo con mayor detalle.

Aparte de estos hay muchos más registros en el microprocesador, pero nosotros con estos pocos tendremos más que suficiente para hacer la gran mayoría de nuestras aplicaciones en ASM.

OLLY DGB: EL DEBUGGER

Un debugger o depurador es una herramienta que nos permitirá ejecutar nuestro programa paso a paso para solucionar ciertos problemas de programación con los que nos podamos encontrar cuando estemos programando en ASM o en cualquier otro lenguaje.

Lo primero que debemos hacer es descargarnos este debugger de manera totalmente gratuita de su página oficial (http://www.ollydbg.de/odbg110.zip). Una vez lo tengamos todo preparado comenzaremos la primera practica de nuestro taller ;D

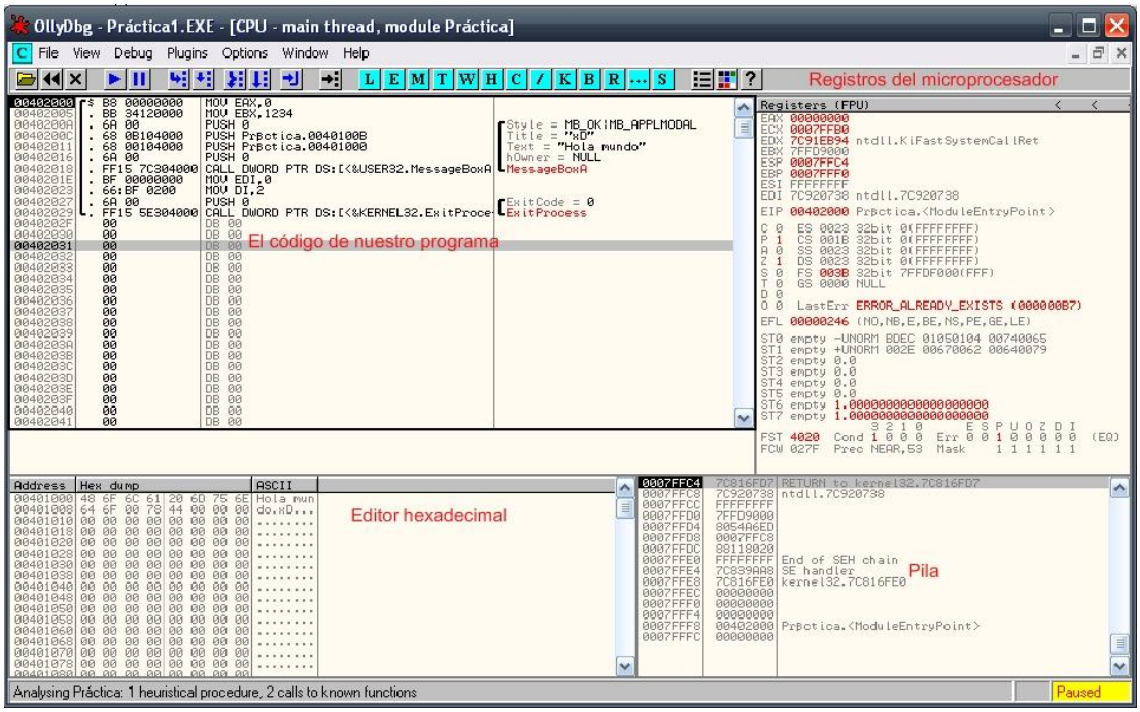
Empezamos. Copiamos este código en el FASM y ensamblarlo pulsando Control+F9. Por ahora no tenéis que entender nada del código, esta práctica es solamente para que veáis lo que pasa dentro del microprocesador al ejecutar un programa:

```
include 'H:\archivos de programa\fasm\include\win32ax.inc'

.data

  cuerpo db 'Hola mundo', 0

  titulo db 'xD', 0



.code

start:



  mov eax, NULL    ; Esto equivale a eax = 0

  mov ebx, 0x1234  ; Esto equivale a ebx = 1234 (el 0x de delante indica que es hexadecimal)



  push 0           ; Observamos la pila para ver que pasa

  push titulo

  push cuerpo
```

```
  push 0

  call [MessageBoxA]



  mov edi, NULL

  mov di, 2



  push 0

  call [ExitProcess]

.end start
```

Apretamos Control+F9, guardamos este código en cualquier ruta con el nombre de Practica1.asm y vereis como se genera un .exe.

Ahora abrimos el olly, File -> Open y seleccionamos nuestro ejecutable. Veremos algo como esto:



En rojo os he indicado las partes fundamentales de las que consta nuestra ventana. Las analizo a continuación una a una ;)

Código del programa: Aquí podemos ver varias cosas interesantes. En la primera columna empezando por la izquierda podemos ver la dirección de memoria en la que se encuentra cierta instrucción. Presupongo que todo el mundo sabe más o menos como se carga un archivo en memoria. En la segunda columna podemos ver nuestro código en forma de opcodes, o lo que es lo mismo, nuestro código expresado en unos y ceros, solo que en base 16 para una mayor claridad, es decir, en hexadecimal. En la tercera columna vemos nuestro código tal y como lo hemos escrito en el FASM y en la última columna vemos algunas aclaraciones que nos pone el olly que nos serán de gran ayuda para entender mejor lo que hace el programa.

Registros: En esa parte del debugger podemos ver los registro que hemos explicado antes y los datos que contienen.

Editor hexadecimal: Vemos nuestro programa tal y como está cargado en memoria. De nuevo todo eso son unos y ceros, pero se expresa en hexadecimal por que para un humano es más facil comprenderlo.

Pila: Aquí podemos ver nuestra famosa pila. Según vayamos ejecutando el programa veremos las cosas que entran y salen de ella y de que manera lo hacen.

Bien, ahora ya sabemos donde están los datos que nos interesan de nuestro programa, pero el programa no se está ejecutando ni haciendo nada, ¿por qué? Este es precisamente el objetivo de un debugger, ejecutar el programa paso a paso, así que vamos a aprender lo necesario para lograrlo.

F8: Si pulsamos esta tecla el programa se ejecutará instrucción a instrucción permitiéndonos ver en cada momento el valor que tienen los registros, las variables y de mas.

F7: Causa el mismo efecto que F8, con la diferencia de que si nos encontramos en un call y pulsamos F8 el programa no entrará dentro de la función, si pulsamos F7 si lo hará y podemos ver como funciona por dentro la misma.

F2: Sirve para poner un breakpoint. Es decir, una "marca" que hará que el programa se para al llegar a ella.

F9: Ejecuta con normalidad el programa a no ser que se encentre con un breakpoint.

Bien, ahora ya conocemos todo lo necesario para depurar nuestros programas, así que manos a la obra. Lo primero que vamos a hacer es poner un breakpoint justo después de la llamada a MessageBoxA pulsando F2. Veréis que el 40201E que indica la posición de memoria de esta instrucción (por lo menos esa es la dirección en mi caso) se pone en rojo.

Ahora vamos a ir pulsando F8 poco a poco para ir viendo lo que sucede. Ejecutamos el mov eax, NULL y si miramos a la derecha, en la zona de los registros, podemos ver como el registro eax se a puesto a cero. Si pulsamos F8 de nuevo veremos como el ebx toma valor 1234.

Ahora entramos en la zona de los push, así que miramos a la pila (abajo a la derecha). Ejecutamos con F8 el primer push y veremos como "empujamos" dentro de la pila un 0. Pulsamos F8 hasta que estemos justamente en el call. Ahora en vez de F8 pulsamos F7 para entrar dentro de la llamada al api MessageBoxA. Lo primero que notaremos es que en la pila entra un parámetro más, el 0x40102E en mi caso por lo menos, que es la dirección de retorno, esto le indica al programa que cuando termine la llamada al api tiene que regresar a ese punto la ejecución.

Si queremos podemos ir pulsando F8 dentro del api, pero como ese código no nos interesa pulsamos F9 para que el programa se ejecute con normalidad. Si lo hacemos veremos que se nos muestra un mensaje emergente y que el programa se detiene donde habíamos puesto nuestro breakpoint.

Finalmente podemos seguir pulsando F8 para ver como van cambiando los registros indicados (incluido el EIP, que apuntará constantemente a la siguiente instrucción) y finalmente salir del programa al llamar al api ExitProcess.

INSTRUCCIONES BÁSICAS DEL PROCESADOR:

Ha continuación explicaré algunas instrucciones básicas que nos proporcionan los micros de la familia x86. Hay más, podéis googlear un poco para verlas, pero con estas nos apañaremos:

MOV destino, origen

Sirve para mover información del origen al destino. Por ejemplo de un registro a otro, de un registro a una variable, etc. Aunque no nos sirve para mover datos entre dos variables.

Ejemplos:

```
mov eax, ebx    ; eax = ebx

mov eax, 1 ; Hace eax = 1

mov eax, miDWORD ; eax = Puntero a una variable

mov eax, [miDWORD]  ; eax = Contenido de la variable

mov [miDWORD], eax ; MiDWORD pasa a valer lo que valga eax

mov [miDWORD], [otroDWORD] ; No válido
```

Es muy importante distinguir que en FASM si ponemos una variable entre "[ ]" nos estamos refiriendo a su contenido y si no a su posición en memoria.

INC/DEC Variable o registro

INC suma uno a una variable o registro y DEC le resta uno.

Ejemplos:

inc eax

dec ecx

inc [miDWORD]

dec [miDWORD]

ADD/SUB Destino, Cantidad

ADD suma un determinado número a un registro o variable. SUB lo resta. Como siempre no está permitido sumar o restar datos directamente entre variables.

Ejemplos:

add eax, 5

add edi, [miDWORD]

add [miDWORD], eax

sub [miDWORD], 2

MUL Multiplicando

MUL varía dependiendo de si el Multiplicando tiene 8 bits, 16 bits o 32 bits:

- Para 8 bits se hace AL * Multiplicando. Se guarda el resultado en AX.

- Para 16 bits se hace AX * Multiplicando. Se guarda el resultado en DX..AX (DX..AX significa que se concatenan los valores, es decir si DX = 0001 y AX = 2001 DX..AX = 00012001)

- Para 32 bits se hace EAX * Multiplicando. Se guarda el resultado en EDX:EAX

Ejemplo:

```
mov [miDWORD], 0x12932

mov eax, 0x54562

mul [miDWORD]

; EDX..EAX = 0x000000061E863F24
```

DIV Divisor

DIV de nuevo varía un poco dependiendo de si el divisor tiene 8 bits, 16 bits o 32 bits:

- Para 8 bits se hace AL / Divisor. Se guarda el resultado AL y el resto en AH.
- Para 16 bits se hace DX..AX / Divisor. Se guarda el resultado en AX y el resto en DX.
- Para 32 bits se hace EDX..EAX / Divisor. Se guarda el resultado en EAX y el resto en EDX.

Ejemplo:

```
mov [miByte], 4

mov al, 9

div [miByte]

; al = Resultado entero = 9/4 = 2

; ah = Resto = 1
```

PUSH DWORD

PUSH mete un DWORD en la pila. La pila se utiliza por ejemplo para pasar parámetros a las funciones o para utilizar variables locales. Es importante que cada PUSH tenga un POP (ahora lo explico).

Ejemplos:

```
push [miDWORD]

push eax

push edi
```

POP DWORD

POP al contrario que PUSH saca un DWORD de la pila.

Ejemplos:

```
push [miDWORD]

pop ebx

push eax

pop [miDWORD]
```

Como ya sabéis podéis ver como cambia la pila con el olly por ejemplo ;)

RET

RET se usa generalmente para regresar de una función. Lo que hace realmente es "popear" un DWORD de la pila y saltar a él. De ahí que en las funciones se meta la dirección de retorno en la pila y con el RET continúe la ejecución del programa donde corresponde.

A los interesados en el Stack Overflow esto les puede interesar bastante ;)

JMP Etiqueta/Dirección

JMP viene de jump y como el nombre indica vale para saltar a una determinada posición de memoria. FASM nos permite utilizar etiquetas que no son más que una palabra seguida de ":" para facilitar el salto. Luego el linker ya se encarga de ajustarlo todo. Con un ejemplo se entiende mejor.

Ejemplo:

```
jmp continuar

mov eax, eax ; Esto no se ejecutaría

continuar:
```

CMP Valor1, Valor2

CMP compara dos valores. En realidad lo que hace esta instrucción es restar el Valor2 del Valor1 y activar el Flag correspondiente a el resultado. Por ejemplo si son iguales, si uno es mayor que el otro… (un Flag es como un registro pero que solo almacena 1 bit).

Usaremos esta instrucción para los saltos condicionales que se explicarán a continuación.

SALTOS CONDICIONALES:

Estos saltos se diferencian del JMP en que mientras que el primero salta donde le indiquemos siempre estos pueden hacerlo o no. Esto nos puede resultar muy útil para simular lo que sería la estructura IF de un lenguaje de alto nivel por poner un ejemplo.

Hay muchos tipos, así que no pondré todos, pero por citar algunos:

JE – Salta si los números comparados son iguales.

JNE – Salta si no son iguales.

JG – Salta si Valor1 es mayor que Valor2.

JGE – Salta si Valor1 es mayor o igual que Valor2.

JB – Salta si Valor1 es menor que Valor2.

JBE – Salta si Valor1 es menor o igual que Valor2.

Como ya he dicho hay muchos más, así que buscad un poco por google, que son unos cuantos para ponerlos todos.

Ejemplo:

```
mov eax, 2

cmp eax, 2

je salir


 invoke MessageBox, 0, 'No salta', '', 0 ; Esto no se ejecuta


salir:

      invoke ExitProcess, 0
```

INSTRUCCIONES LÓGICAS:

Aparte de las instrucciones que he puesto (y las que no he puesto :P) hay una serie de operaciones lógicas que nos sirven para tratar con bits. Aquí dejo las tablas de verdad de algunas de ellas. No os preocupéis si no sabéis aun que significa cada cosa, más abajo las explico todas, solo echadle un ojo a la tabla:

X	Y	AND	OR	XOR		X	NOT
0	0	0	0	0	|	0	1
0	1	0	1	1	|	1	0
1	0	0	1	1	|		
1	1	1	1	0	|		

Aparte de las que hay en la tabla también vamos a ver otras ;) No voy a entrar en mucho detalle, se usan como todas las instrucciones que ya he explicado, solo hay que seguir la tabla.

NOT Destino

Lo que hace esta operación lógica es cambiar 1's por 0's y viceversa. Imaginaos que tenemos en eax el valor 001101 (en binario), tras hacer un "not ax" nos quedaría almacenado en ese mismo registro 110010.

AND Destino, Dato

Imaginemos que tenemos guardado en eax el valor 101011 y en ebx el valor 011101,. Tras hacer "and eax, ebx" nos quedaría guardado en eax el valor 001001.

OR Destino, Dato

Imaginemos que ahora tenemos guardado en eax 1010 y en ebx 1100. Tras hacer "or eax, ebx" nos quedaría guardado en eax 1110.

XOR Destino, Dato

Ahora tenemos en eax 10110 y en ebx 11011. Tras hacer "xor eax, ebx" nos quedaría en eax 01101. Esta operación es especialmente útil (y muy utilizada) en métodos de encriptación, ya que si os fijáis en su tabla de verdad al hacer un xor a un bit "destino" con otro bit "dato" que valga 5 por poner un ejeplo, el bit "destino" quedará totalmente irreconocible. Lo gracioso del asunto es que si volvemos a hacer al bit "destino" un xor de nuevo con un 5 volverá a valer lo que valía en un inicio.

ROL/ROR Destino, Rotación

Estas operaciones a nivel bit lo que hacen es mover hacia la izquierda (ROL) o hacia la derecha (ROR) un grupo de bits tantas posiciones como indique "Rotación". Imaginemos que hacemos "ror eax, 2" pues moveríamos todos los bits de eax dos posiciones hacia la derecha. ¿Qué pasa con los bits que estan muy a la derecha y se

salen? Pues simplemente “entran” por la izquierda. Podemos imaginar que los bits están formando un círculo y los vamos desplazando por el mismo con ROR y ROL. Por poner un ejemplo si tenemos estos bits “101011” y les hacemos un ROL con “rotación” 1 nos quedaría 010111.

SHL/SHR Destino, Rotación

SHL es como ROL y SHR como ROR con la diferencia de que los bits que se salen al ser desplazados no entran por el otro lado, si no que los bits que se pierden son reemplazados por ceros. Por poner un ejemplo si hacemos un SHL con “rotación” 2 a 000101 nos quedaría 010100. Si os fijáis un poco veréis que 000101 en decimal es 5 y 010100 es 20. Es decir que SHL con rotación 1 es como multiplicar por 2, con rotación 2 es como multiplicar por 4….

Hasta aquí el apartado de instrucciones lógicas. Hay muchas más, al igual que hay muchas más instrucciones propias de ensamblador, pero por falta de tiempo no puedo explicarlas todas, así que recomiendo mirar las que faltan. Por recomendar algunas webs interesantes:

http://www.jegerlehner.com/intel/IntelCodeTable_es.pdf
http://www.wikilearning.com/curso_gratis/curso_de_programacion_de_virus-introduccion/4312-1

ESTRUCTURA DEL CÓDIGO EN FASM:

Una vez ya conocemos todas las instrucciones básicas con las que cuenta nuestro microprocesador ya podemos empezar a programar cosas más serias y útiles :P.
Para ello tenemos que saber como escribir nuestro código en FASM para que lo entienda y genere un ejecutable.
FASM tiene diversas formas de estructurar el código, entre ellas destacan estas tres. La primera utiliza el include win32ax.inc y las otras dos el include win32a.inc:

Con Win32ax:

Lo primero de debemos hacer es incluir el win32ax: *include ‘C:\archivos de programa\fasm\include\win32ax.in’* claro, que variará dependiendo de donde tengáis instalado el FASM ;)

Con este método nos ahorramos construir la ImportTable a mano, es decir ir poniendo el nombre de cada api que usamos indicando la dll a la que pertenece.
La estructura básica es esta:

```
Include ‘win32ax.inc’

.data
        ; Aquí van las variables
```

```
.code
start: ; Aquí empezaría el programa
        ; Aquí va el código

        ; Aquí abajo también podemos poner variables :D
.end start ; Aquí terminaría el programa
```

Si os fijais en el ejecutable generado mediante este método y sabéis un poco del formato PE veréis que aparte de las secciones .data y .code que le hemos indicado nosotros también a creado el solo una sección .idata con los nombres de todas las apis que hemos utilizado.

La ventaja de esta forma de estructurar el código es clara: nos ahorramos un montón de trabajo generando la ImporTable a mano.
Las desventajas son que si por ejemplo queremos usar un api que no esté declarada en la win32ax.inc tenemos que cargarla con LoadLibrary y GetProcAddress, mientras que si hubiesemos creado la Importable a mano esto no pasaría.

Con Win32a:

Usando este include existen dos formas de estructurar el código. La primera es indicando el nombre de cada sección y sus atributos, por ejemplo:

```
format PE GUI ; Tipo de ejecutable
entry Start ; El “main” de nuestro código

section '.code' code readable executable

  start:

        ; Aquí el código

section '.data' data readable writeable

        ; Aquí las variables

section '.idata' import data readable writeable

        ; Aquí el ImportData
```

Mirad el ejemplo que viene con FASM en la ruta FASM\EXAMPLES\PEDEMO para ver como funciona ;) Este formato es de lo más útil para manejar cuantas secciones queremos que tenga el archivo, sus características y donde queremos que vaya cada cosa dentro de las mismas.

Otra opción con este include es no señalar ninguna sección, con lo que FASM creará una única sección llamada ".flat" en la que pondrá todo, las estructura es esta:

```
include 'H:\Archivos de programa\FASM\INCLUDE\WIN32A.inc'
        ; Aquí el código
        ; Aquí las variables
data import
        ; Aquí el ImportData
end data
```

Mirad el ejemplo FASM\EXAMPLES\BEER para más información :P

VARIABLES EN FASM:

A diferencia de los lenguajes de programación de alto nivel en ensamblador no tenemos variables del tipo char, integer, long o cosas por el estilo, con ensamblador trataremos directamente con bytes y conjuntos de bytes. Los tipos principales de datos y con los que nos apañaremos son tres. El byte, el word (16 bits, es decir, 2 bytes) y el dword (32bits, es decir, 4 bytes).

La declaración de variables es muy simple, solamente hay que poner (en el sitio donde corresponda, que eso ya lo hemos visto en el punto anterior) el nombre de la variable, seguido de su tipo (db si es un byte, dw si es un word o dd si es un dword) seguido de un ? si queremos que no se inicialice con ningún valor o también podemos indicar el valor con el que queremos que se inicialice la variable. Por ejemplo:

```
cad db 'Hola', 0  ; Variable de tipo byte. Como podeis apreciar
                  ; la inicialozamos con un valor que es muy superior
                  ; a un unico byte, ya que FASM nos permite construir
                  ; cadenas de texto de esta manera. Fiajaos en el 0
```

```
                ; final, es es caracter de fin de cadena, que es un
                ; caracter que debemos poner para saber donde termina
                ; la cadena.
    palabra dw ?      ; Word no incializado
    doble   dd 5      ; Dword inicializado a 5, es decir, esos cuatro bytes
                ; contienen el valor 5
```

Pues ya está con esto ya sabemos declarar variables. Para utilizarlas solo tenéis que referiros a ellas como hemos hecho en los puntos anteriores con mov's, and's y de más ;)

DESPEDIDA:

Con mucha pena (y dolor en los dedos de tanto escribir) despido este taller creado originalmente para elhacker.net con motivo del evento "Abril negro 2008" y ahora reeditado y en formato pdf para mi blog:

http://e0n-productions.blogspot.com/

Dejo a cada uno la tarea de aprender a manejar las apis de su sistema operativo y empezar a crear sus propias aplicaciones en ASM con FASM.

1S4ludo, E0N