Traducción a FASM (Windows)

(Tutorial Iczelion Masm32)

Nombre: Jose Manuel Vargas Cruz **Carrera:** Ing. Informática

Tutorial 1: Lo básico

Teoría:

Los programas de Win32 corren en modo protegido, disponible desde el 80286. Pero ahora el 80286 es historia. Asi que ahora debemos interesarnos en el 80386 y sus descendientes. Windows corre cada programa de 32 bits en espacios virtuales de memoria separados. Eso significa que cada programa de Win32 tiene sus propios 4 GB de memoria en el espacio de direcciones. Como sea, esto no significa que cada programa de Win32 tenga 4GB de memora física, sino que el programa puede direccionar cualquier dirección en ese rango.

Windows hará cualquier cosa que sea necesaria para hacer que la memoria y las referencias del programas sean válidas. Por supuesto, el programa debe adherirse a las reglas impuestas por Windows, si no, causará un error de protección general. Cada programa está solo en su espacio de direcciones. Esto contrasta con la situación en Win16. Todos los programas de Win16 podían *verse* unos a otros. No es lo mismo en Win32. Esto reduce la posibilidad de que un programa escriba sobre el código/datos de otros programas.

El modelo de la memoria es también drásticamente diferente al de los antiguos días del mundo de 16-bits. Bajo Win32, ya no necesitamos meternos nunca más con el modelo o los segmentos de memoria! Hay un solo tipo de modelo de memoria: El modelo plano (**flat**).

Ahora, no hay solamente segmentos de 64k. La memoria es un largo y continuo espacio de 4GB. Eso también significa que no tendrás que jugar más con los registros de los segmentos. Ahora puedes usar cualquier registro de segmento para direccionar a cualquier punto en el espacio de la memoria. Eso es una GENIAL ayuda para los programadores. Esto hace la programación de ensamblador para Win32 tan fácil como C.

Cuando programas bajo Win32, debes tener en cuenta unas cuantas reglas importantes. Una es que **Windows usa esi ,edi ebp y ebx internamente y no espera que los valores en esos registros cambien**. Así que recuerda esta regla primero: **si usas cualquiera de estos cuatro registros en tu función callback, nunca olvides restaurarlos antes de regresar el control a Windows**. Una función callback es una función escrita por ti que Windows llama cuando algún evento específico se produce. El ejemplo más obvio es el procedimiento de

ventana. Esto no significa que no puedas usar estos cuatro registros; sí puedes. Solo asegúrate de restaurarlos antes de pasarle el control a Windows.

Contenido:

format PE GUI 4.0 //Win32 ejecutable portable formato GUI, puede ser console

entry start // start es el punto de entrada del programa

include <**win32a.inc**> //Puede ser win32ax, win32wx, win64ax y win64wx

<Tus constantes>

//Esta sección contiene declaraciones de constantes usadas por tu programa. Las constantes nunca pueden ser modificadas en tu programa. Sólo son *constantes*

section '.text' code readable executable

start:

<Código> //Sección de código

section '.data' data readable writeable

<Datos (información) inicializada o no inicializada>

section '.idata' import data readable writeable

//Sección de import data, se importan librerías, funciones, etc.

Tutorial 2: MessageBox

Teoría:

Windows tiene preparado una gran cantidad de recursos para sus programas. En el centro de esta concepción se ubica la API (Application Programming Interface = Interface de Programación de Aplicaciones) de Windows. La API de Windows es una enorme colección de funciones muy útiles que residen en el propio sistema Windows, listas para ser usadas por cualquier programa de Windows. Estas funciones están almacenadas en varias librerías de enlace dinámico [dynamic-linked libraries (DLL)] tales como kernel32.dll, user32.dll y gdi32.dll. Kernel32.dll contiene las funciones de la API que tienen que ver con el manejo de memoria y de los procesos. User32.dll controla los aspectos de la interface de usuario de tu programa. Gdi32.dll es la responsable de las operaciones gráficas. Además de estas "tres funciones principales", hay otras DLLs que nuestros programas pueden emplear, siempre y cuando tengas la suficiente información sobre las funciones de la API que te interesan.

Los programas de Windows se enlazan dinámicamente a estas DLLs, es decir, las rutinas de las funciones de la API no están incluidas en el archivo ejecutable del programa de Windows. Con el fin de que tu programa pueda encontrar en tiempo de ejecución las funciones de la API deseadas, tienes que meter esa información dentro del archivo ejecutable. Esta información se encuentra dentro de archivos .LIB. Debes enlazar tus programas con las librerías de importación correctas o no serán capaces de localizar las funciones de la API.

Cuando un programa de Windows es cargado en la memoria, Windows lee la información almacenada en el programa. Esa información incluye el nombre de las funciones que el programa usa y las DLLs donde residen esas funciones. Cuando Windows encuentra esa información en el programa, cargará las DLLs y ejecutará direcciones de funciones fijadas en el programa de manera que las llamadas transfieran el control a la función correcta.

Hay dos categorías de funciones de la API: Una para ANSI y la otra para Unicode. Los nombres de las funciones de la API para ANSI terminan con "A", por ejemplo, MessageBoxA. Los de Unicode terminan con "W" [para Wide Char (caracter ancho), pienso]. Windows 95 soporta ANSI y Windows NT Unicode.

Generalmente estamos familiarizados con las cadenas ANSI, que son arreglos de caracteres terminados en NULL. Un caracter ANSI tiene un tamaño de 1 byte. Si bien el código ANSI es suficiente para los lenguajes europeos, en cambio no puede manejar algunos lenguajes orientales que tienen millares de caracteres únicos. Esa es la razón por la cual apareció UNICODE. Un caracter UNICODE tiene un tamaño de 2 bytes, haciendo posible tener 65536 caracteres únicos en las cadenas.

Sin embargo, la mayoría de las veces, usarás un archivo include que puede determinar y seleccionar las funciones de la API apropiadas para tu plataforma. Sólo referencia los nombres de las funciones de la API sin su sufijo.

Ejemplo:

Vamos a ponerle ahora una caja de mensaje [Dialog Box]. Su prototipo de función es:

MessageBox PROTO hwnd:DWORD, lpText:DWORD, lpCaption:DWORD, uType:DWORD

hwnd es el manejador de la ventana padre. Puedes pensar en el manejador como un número que representa la ventana a la cual te refieres. Su valor no es tan importante para tí. Sólo recuerda que representa la ventana. Cuando quieres hacer algo con la ventana, debes referirte a ella por su manejador [handle].
lpText es el puntero al texto que quieres desplegar en el área cliente de la caja de mensaje. En realidad, un puntero es la dirección de lago: Puntero a cadena de texto==Dirección de esa cadena.
lpCaption es un puntero al encabezado de la caja de mensaje
uType especifica el icono, el número y el tipo de botones de la caja de mensajes

```
format PE GUI 4.0

entry start

include 'win32a.inc'


section '.text' code readable executable


   start:

      invoke  MessageBox,0,texto,titulo,MB_OK

      invoke  ExitProcess,0


section '.data' data readable writeable

      titulo     db "Iczelion Tutorial No.2",0

      texto      db "Win32 Assembly is Great!",0


section '.idata' import data readable writeable

       library kernel32,'KERNEL32.DLL',\    ; Importamos las bibliotecas

               user32,'USER32.DLL'
```

```
include 'api\kernel32.inc'          ; KERNEL32 API calls
include 'api\user32.inc'            ; USER32 API calls
```

Tutorial 3: Una Ventana Simple

Teoría:

Los programas de Windows realizan la parte pesada del trabajo de programación a través funciones API para sus GUI (Graphic User Interface = Interface de Usuario Gráfica). Esto beneficia a los usuarios y a los programadores. A los ususarios, porque no tienen que aprender cómo navegar por la GUI de cada nuevo programa, ya que las GUIs de los programas Windows son semejantes. A los programadores, porque tienen ya a su disposición las rutinas GUI, probadas y listas para ser usadas. El lado negativo para los programadores es la creciente complejidad involucrada. Con el fin de crear o de manipular cualquiera de los objetos GUI, tales como ventanas, menúes o iconos, los programadores deben seguir un "récipe" estricto. Pero esto puede ser superado a través de programación modular o siguiendo el paradigma de Progranación orientada a Objetos (OOP = Object Oriented Programming).

Esbozaré los pasos requeridos para crear una ventana sobre el escritorio:

1. Obtener el manejador [handle] de instancia del programa (requerido)
2. Obtener la línea de comando (no se requiere a menos que el programa vaya a procesar la línea de comando)
3. Registrar la clase de ventana (requerido, al menos que vayan a usarse tipos de ventana predefinidos, eg. MessageBox o una caja o de diálogo)
4. Crear la ventana (requerido)
5. Mostrar la ventana en el escritorio (requerido al menos que se quiera mostrar la ventana inmediatamente)
6. Refrescar el área cliente de la ventana
7. Introducir un bucle (loop) infinito, que chequée los mensajes de Windows
8. Si llega un mensaje, es procesado por una función especial, que es responsable por la ventana
9. Quitar el programa si el usuario cierra la ventana

Como puedes ver, la estructura de un programa de Windows es más compleja que la de un programa de DOS, ya que el mundo de Windows es totalmente diferente al mundo de DOS. Los programas de Windows deben ser capaces de coexistir pacíficamente uno junto a otro. Por eso deben seguir reglas estrictas. Tú (o Usted), como programador, debes ser más estricto con tus estilos y hábitos de programación.

```
format PE GUI 4.0

entry start


include 'win32a.inc'        ; Incluímos definiciones de estructuras y constantes

;WinMain proto dword,dword,dword,dword


cdXPos       EQU  128       ; Constante double X-Posición de la ventana(esq sup izqda)

cdYPos       EQU  128       ; Constante double Y-Posición de la ventana(esq sup izqda)

cdXSize      EQU  320       ; Constante double X-tamaño de la ventana

cdYSize      EQU  200       ; Constante double Y-tamaño de la ventana

cdColFondo    EQU  COLOR_BTNFACE + 1  ; Color de fondo de la ventana: gris de un botón de
comando

cdVIcono     EQU  IDI_APPLICATION ; Icono de la ventana, véase Resource.H

cdVCursor    EQU  IDC_ARROW   ; Cursor para la ventana

; Tipo de ventana (Barra de cabecera)

;cdVBarTipo  EQU  WS_EX_TOOLWINDOW                ; Tipo de barra de Cabecera: delgado, sin
icono, sin reflejo en barra de tareas

cdVBarTipo    EQU  NULL                          ; Normal, con icono

cdVBtnTipo    EQU  WS_VISIBLE+WS_DLGFRAME+WS_SYSMENU  ; Normal sólo con botón cerrar

; cdVBtnTipo     EQU  WS_OVERLAPPEDWINDOW          ; Normal sólo con los tres botones


section '.text' code readable executable
```

```
 start:
   invoke    GetModuleHandle,0             ; obtener el manejador de instancia del programa.
   mov       [wc.hInstance],eax
   invoke    GetCommandLine                ; Obtener la línea de comando. No hay que llamar esta función
                                           ; si el programa no procesa la línea de comando
   mov       [CommandLine], EAX
   stdcall   WinMain, [wc.hInstance], NULL, [CommandLine], SW_SHOWDEFAULT       ; llamar la función principal
   invoke    ExitProcess,[msg.wParam]      ; quitar nuestro programa. El código de salida es devuelto en eax desde WinMain.

proc WinMain uses ebx esi edi, hInst, hPrevInst, CmdLine, CmdShow
   ;  Propósito: Inicializamos la ventana principal de la aplicación y captura errores, si los hubiere
   ;  Entrada  : hInst, hPrevInst, CmdLine, CmdShow
   ;  Salida   : Ninguna
   ;  Destruye : Ninguna
   invoke    LoadIcon,0,cdVIcono
   mov       [wc.hIcon],eax
   invoke    LoadCursor,0,cdVCursor
   mov       [wc.hCursor],eax
   invoke    RegisterClass,wc       ; registrar nuestra clase de ventana
   test      eax,eax
   jz        error

   invoke    CreateWindowEx,cdVBarTipo,NombreClase,MsgCabecera,\
             cdVBtnTipo,cdXPos, cdYPos, cdXSize, cdYSize,\
             NULL,NULL,[wc.hInstance],NULL
   test      eax,eax
   jz        error
```

```
  msg_loop:
    invoke    GetMessage,msg,NULL,0,0     ; Introducir en bucle (loop) de mensajes
    cmp       eax,1
    jb        end_loop
    jne       msg_loop
    invoke    TranslateMessage,msg
    invoke    DispatchMessage,msg
  jmp       msg_loop

 error:
  invoke    MessageBox,NULL,MsgError,NULL,MB_ICONERROR+MB_OK

 end_loop:
  MOV       EAX, [msg.wParam]
  ret
endp

proc WndProc uses ebx esi edi, hwnd,wmsg,wparam,lparam
  ;  Propósito: Procesa los mensajes provenientes de las ventanas
  ;  Entrada  : hwnd,wmsg,wparam,lparam
  ;  Salida   : Ninguna
  ;  Destruye : Ninguna
  cmp       [wmsg],WM_DESTROY             ; si el usuario cierra nuestra ventana
  je        .wmdestroy
 .defwndproc:
  invoke    DefWindowProc,[hwnd],[wmsg],[wparam],[lparam]       ; Procesar el mensaje por
defecto
  jmp       .finish
```

```
 .wmdestroy:
   invoke    PostQuitMessage,0                   ; quitar nuestra aplicación
   xor       eax,eax
 .finish:
   ret
endp

section '.data' data readable writeable

 NombreClase TCHAR    'SimpleWinClass',0     ; el nombre de nuestra clase de ventana
 MsgCabecera TCHAR    'Our First Window',0       ; el nombre de nuestra ventana
 MsgError    TCHAR    'Carga inicial fallida.',0
 CommandLine DD       ?
 wc          WNDCLASS 0,WndProc,0,0,NULL,NULL,NULL,cdColFondo,NULL,NombreClase
 msg         MSG

section '.idata' import data readable writeable

 library kernel32,'KERNEL32.DLL',\    ; Importamos las bibliotecas para que el enlazador pueda
trabajar
        user32,'USER32.DLL'         ; Importamos las bibliotecas para que el enlazador pueda trabajar

 include 'api\kernel32.inc'          ; KERNEL32 API calls
 include 'api\user32.inc'            ; USER32 API calls
```

Nuestra primera instrucción llama a GetModuleHandle para recuperar el manejador de instancia de nuestro programa. Bajo Win32, el manejador de la instancia y el manejador del módulo son una y la misma cosa. Se puede pensar en el manejador de instancia como el ID de nuestro programa. Es usado como parámetro en algunas funciones API que nuestro programa debe llamar, así que generalmente es una buena idea obtenerlo al comienzo del programa.

Nota: Realmente bajo win32, el manejador de instancia es la dirección lineal de nuestro programa en la memoria.

Al regresar a una función de Win32, el valor regresado, si hay alguno, puede encontrarse en el registro eax. Todos los demás valores son regresados a través de variables pasadas en la lista de parámetros de la función que va a ser llamada.

Cuando se llama a una función Win32, casi siempre preservará los registros de segmento y los registros ebx, edi, esi y ebp. Al contrario, los registros eax, ecx y edx son considerados como registros dinámicos y siempre sus valores son indeterminados e impredecibles cuando retorna una función Win32.

Nota: No esperes que los valores de eax, ecx, edx sean preservados durante las llamadas a una función API.

La línea inferior establece que: cuando se llama a una fucnión API, se espera que regrese el valor en eax. Si cualquiera de las funciones que creamos es llamada por Windows, también debe seguir la siguiente regla: preservar y restablecer los valores de los registros de segmentos ebx, edi, esi y ebp cuando la función regrese, sino el programa se quebrará (***crash***) de inmediato, esto incluye el procedimiento de ventana y las funciones callback de ventanas.

La llamada a GetCommandLine es inecesaria si el programa no procesa la línea de comando. En este ejemplo muestro como llamarla en caso que sea necesario en un programa.

Lo siguiente es la llamada a WinMain. Aquí recibe cuatro parámetros: el manejador de instancia de nuestro programa, el manejador de instancia de la instancia previa del programa, la línea de comando y el estado de la ventana en su primera aparición. Bajo Win32, no hay instancia previa. Cada programa está aislado en su espacio de direcciones, así que el valor de hPrevInst siempre es 0. Esto es uno de los restos de los días de Win16 cuando todas las instancias de un programa corrían en el mismo espacio de direcciones y una instancia necesitaba saber si era la primera. En win16, si hPrevInst es NULL entonces es la primera instancia.

Nota: Esta función no tiene que ser declarada como WinMain. En realidad, hay completa libertad a este respecto. Ni siquiera hay que usar siempre una función equivalente a WinMain. Se puede pegar el código dentro de la función WinMain inmediatamente después de GetCommandLine y el programa funcionará perfectamente.

Al regresar de WinMain, eax tiene el código de salida. Pasamos el código de salida como parámetro de ExitProcess, que terminará nuestra aplicación.

WinMain proc Inst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD

La línea de arriba forma la declaración de la función WinMain. Nota que los pares parámetro:tipo que siguen a la directiva PROC. Son parámetros queWinMain recibe desde la instrucción que hace la llamada [caller]. Puedes referirte a estos parámetros por nombre en vez de a través de la manipulación de la pila. Además, MASM generará los códigos de prólogo y epílogo para la función. Así que no tenemos que preocuparnos del marco de la pila cuando la función entre (*enter*) y salga (*exit*).

```
mov   wc.cbSize,SIZEOF WNDCLASSEX
   mov   wc.style, CS_HREDRAW or CS_VREDRAW
   mov   wc.lpfnWndProc, OFFSET WndProc
   mov   wc.cbClsExtra,NULL
   mov   wc.cbWndExtra,NULL
   push  hInstance
   pop   wc.hInstance
   mov   wc.hbrBackground,COLOR_WINDOW+1
   mov   wc.lpszMenuName,NULL
   mov   wc.lpszClassName,OFFSET ClassName
   invoke LoadIcon,NULL,IDI_APPLICATION
   mov   wc.hIcon,eax
   mov   wc.hIconSm,eax
   invoke LoadCursor,NULL,IDC_ARROW
   mov   wc.hCursor,eax
   invoke RegisterClassEx, addr wc
```

Las líneas intimidantes de arriba son realmente comprensibles en cuanto concepto. Toma varias líneas de instrucciones realizar la operación ahí implicada. El concepto detrás de todas estas líneas es el de *clase de ventana* (***window class***). Una clase de ventana no es más que un anteproyecto o especificación de una ventana. Define algunas de las características importantes de una ventana tales como un icono, su cursor, la función que se resposabiliza de ella, su color etc. Una ventana se crea a partir de una clase de ventana. Este es una especie de concepto orientado a objeto. Si se quiere crear más de una ventana con las mismas características, lo razonable es almacenar todas estas características en un solo lugar y referirse a ellas cuando sea necesario. Este esquema salva gran cantidad de memoria evitando duplicación de código. Hay que recordar que Windows fue diseñado cuando los chips de memoria eran prohibitivos ya que una computadora tenía apenas 1 MB de memoria. Windows debía ser muy eficiente al usar recursos de memorias escasos. El punto es: si defines tu propia ventana, debes llenar las características de tu ventana en una estructura WNDCLASS o WNDCLASSEX y llamar a RegisterClass o RegisterClassEx antes de crear la ventana. Sólo hay que registrar la clase de ventana una vez para cada tipo de ventana que se quiera crear desde una clase de ventana.

Windows tiene varias clases de ventanas predefinidas, tales como botón (*button*) y caja de edición (*edit box*). Para estas ventanas (o controles), no tienes que registrar una clase de venana, sólo hay que llamara a CreateWindowEx con el nombre de la clase predefinido.

El miembro más importante en WNDCLASSEX es lpfnWndProc. lpfn se concibe como un puntero largo a una función. Bajo Win32, no hay puntero "cercano" o "lejano" pointer, sino sólo puntero, debido al nuevo modelo de memoria FLAT. Pero esto también es otro de los restos de los días de Win16. Cada clase de ventana debe estar asociada con la función llmada procedimiento de ventana. El procedimiento de ventana es la función responsable por el manejo de mensajes de todas las ventanas creadas a partir de la clase de ventana asociada. Windows enviará mensajes al procedimiento de ventana para notificarle sobre eventos importantes concernientes a la ventana de la cual el procedimiento es responsable, tal como el uso del teclado o la entrada del ratón. Le toca al procedimiento de ventana responder inteligentemante a cada evento que recibe la

ventana. Seguro que pasarás bastante tiempo escribiendo manejadores de evento en el procedimiento de ventana.

Describo abajo los miembros de WNDCLASSEX:

```
WNDCLASSEX STRUCT DWORD
 cbSize          DWORD     ?
 style           DWORD     ?
 lpfnWndProc     DWORD     ?
 cbClsExtra      DWORD     ?
 cbWndExtra      DWORD     ?
 hInstance       DWORD     ?
 hIcon           DWORD     ?
 hCursor         DWORD     ?
 hbrBackground   DWORD     ?
 lpszMenuName    DWORD     ?
 lpszClassName   DWORD     ?
 hIconSm         DWORD     ?
WNDCLASSEX ENDS
```

cbSize: Tamaño de la estructura WNDCLASSEX en bytes. Podemos usar el operador SIZEOF para obtener este valor.
style: El estilo para las ventanas creadas a partir de esta clase. Se pueden combinar varios tipos de estilo combinando el operador "or".
lpfnWndProc: La dirección del procedimiento de ventana responsable para las ventanas creadas a partir de esta clase.
cbClsExtra: Especifica el número de bytes extra para localizar la siguiente estructura de clase de ventana. El sistema operativo inicializa los bytes poniéndolos en cero. Puedes almacenar aquí datos específcos de la clase de ventana.
cbWndExtra: : Especifica el número de bytes extra para localizar the window instance. El sistema operativo inicializa los bytes poniéndolos en cero. Si una aplicación usa la estructura WNDCLASS para registrar un cuadro de diálogo creado al usar la directiva CLASS en el archivo de recurso, debe poner este miembro en DLGWINDOWEXTRA.
hInstance: Manejador de instancia del módulo.
hIcon: Manejador [handle] del icono. Se obtiene llamando a LoadIcon.
hCursor: Manejador del cursor. Se obtiene llamando a LoadCursor.
hbrBackground: Color de fondo de la ventana creada a partir de esta clase.
lpszMenuName: Manejador del menú por defecto para la ventana creada a partir de esta clase.
lpszClassName: Nombre para esta clase de ventana.
hIconSm: Manejador del icono pequeño asociado con la clse de ventana. Si este miembro es NULL, el sistema busca el recurso de icono especificado por el miembro hIcon para un icono de tamaño apropiado para ser usado como icono pequeño.

```
  invoke CreateWindowEx, NULL,\
                         ADDR ClassName,\
                         ADDR AppName,\
```

```
WS_OVERLAPPEDWINDOW,\
CW_USEDEFAULT,\
CW_USEDEFAULT,\
CW_USEDEFAULT,\
CW_USEDEFAULT,\
NULL,\
NULL,\
hInst,\
NULL
```

Después de registrar la clase de ventana, podemos llamar a CreateWindowEx para crear nuestra ventana basada en la clase de ventana propuesta. Nota que hay 12 parámetros para esta función.

```
CreateWindowExA proto dwExStyle:DWORD,\
  lpClassName:DWORD,\
  lpWindowName:DWORD,\
  dwStyle:DWORD,\
  X:DWORD,\
  Y:DWORD,\
  nWidth:DWORD,\
  nHeight:DWORD,\
  hWndParent:DWORD ,\
  hMenu:DWORD,\
  hInstance:DWORD,\
  lpParam:DWORD
```

Veamos la descripción detallada de cada parámetro:

dwExStyle: Estilos extra de ventana. Es el nuevo parámetro agregado a la antigua función CreateWindow. Aquí puedes poner estilos nuevos para Windows 95 y NT. Puedes especificar tu estilo de ventana ordinario en dwStyle pero si quieres algunos estilos especiales tales como "topmost window" (nventana en el tope), debes especificarlos aquí. Puedes usar NULL si no quieres usar estilos de ventana extra.

lpClassName: (Requerido). Dirección de la cadena ASCIIZ que contiene el nombre de la clase de ventana que quieres usar como plantilla para esta ventana. La Clase puede ser una clase registrada por tí mismo o una clase de ventana predefinida. Como se estableció arriba, todas las ventanas que creas deben estar basadas en una clase de ventana.

lpWindowName: Dirección de la cadena ASCIIZ que contiene el nombre de la ventana. Será mostrada en la barra de título de la ventana. Si este parámetro es NULL, la barra de título de la ventana aparecería en blanco.

dwStyle: Estilos de la ventana. Aquí puedes especificar la apariencia de la ventana. Pasar NULL pero la ventana no tendrá el menú de sistema, ni botones minimizar-maximizar, y tampoco el botón cerrar-ventana. La ventana no sería de mucha utilidad. Necesitarás presionarAlt+F4 para cerrarla. El estilo de ventana más común es WS_OVERLAPPEDWINDOW. UN estilo de ventana sólo es una bandera de un bit. Así que puedes combinar varios estilos usando el operador "or" para

alcanzar la apariencia deseada de la ventana. El estilo WS_OVERLAPPEDWINDOW es realmante la combinación de muchos estilos de ventana comunes empleando este método.

X,Y: La coordenada de la esquina izquierda superior de la vetana. Noramlamente este valor debería ser CW_USEDEFAULT, es decir, deseas que Windows decida por tí dónde poner la ventana en el escritorio.

nWidth, nHeight: El ancho y el alto de la ventana en pixeles. También puedes usar CW_USEDEFAULT para dejar que Windows elija por tí el ancho y la altura apropiada.

hWndParent: El manejador [handle] de la ventana padre de la ventana (si existe). Este parámetro dice a Windows si esta ventana es una hija (subordinada) de alguna otra ventana y, si lo es, cual ventana es el padre. Nota que no se trata del tipo de interrelación padre-hija de la MDI (Multiple Document Interface = Interface de Documento Múltiple). Las ventanas hijas no están restringidas a ocupar el área cliente de la ventana padre. Esta interrelación es únicamente para uso interno de Windows. Si la ventana padre es destruida, todas las ventanas hijas serán destruidas automáticamente. Es realmente simple. Como en nuestro ejemplo sólo hay una ventana, especificamos este parámetro como NULL.

hMenu: Manejador del menú de la ventana. NULL si la clase menú va a ser usada. Observa el miembro de la estructura WNDCLASSEX, lpszMenuName. lpszMenuName especifica el menú *por defecto* para la clase de ventana. Toda ventana creada a partir de esta clase de ventana tendrá por defecto el mismo menú, a menos que especifiques un nuevo menú *imponiéndolo* a una ventana específica a través de su parámetro hMenu. hMenu es realmente un parámetro de doble propósito. En caso de que la ventana que quieras crear sea de un tipo predefinido (como un control), tal control no puede ser propietario de un menú. hMenu es usado entonces más bien como un ID de control. Windows puede decidir si hMenu es realmente un manejador de menú o un ID de control revisando el parámetro lpClassName. Si es el nombre de una clase de ventana predefinida, hMenu es un ID de control, sino entonces es el manejador del menú de la ventana.

hInstance: El manejador de instancia para el módulo del programa que crea la ventana.

lpParam: Puntero opcional a la estructura pasada a la ventana. Es usada por la ventana MDI para pasar los datos CLIENTCREATESTRUCT. Normalmente, este valor es puesto en NULL, que significa que ningún dato es pasado vía CreateWindow(). La ventana puede recibir el valor de este parámetro llamando a la función GetWindowLong.

WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM

Este es nuestro procedimiento de ventana. No tienes que llamarlo necesariamente WndProc. El primer parámetro, hWnd, es el manejador de la ventana hacia el cual el mensaje está destinado. uMsg es el mensaje. Nota que uMsg no es una estructura MSG. Realmente sólo es un número. Windows define cientos de mensajes, muchos de los cuales carecen de interés para nuestro programa. Windows enviará un mensaje apropiado a la ventana en caso de que ocurra algo relevante a la ventana. El procedimiento de ventana recibe el mensage y recciona a él inteligentemente. wParam y lParam sólo son parámetros extra a ser utiizados por algunos mensajes. Algunos mensajes envían datos junto con ellos en adición al mensaje propiamente dicho. Estos datos son pasados al procedimiento de ventana por medio de lParam y wParam.

Aquí viene la parte crucial. Es donde reside gran parte de la inteligencia de los programas. El código que responde a cada mensaje de Windows está en el procedimiento de ventana. El código debe chequear el mensaje de Windows para ver si hay un mensaje que sea de interés. Si lo es, se hace algo que se desee en respuesta a ese mensaje y luego se regresa cero en eax. Si no es así, debe llamarse a DefWindowProc, pasando todos los parámetros recibidos para su procesamiento por defecto. DefWindowProc es una función de la API que procesa los mensajes en los que tu programa no está interesado.

El único mensaje a que DEBES responder es WM_DESTROY. Este mensaje es enviado a tu procedimiento de ventana cada vez que la ventana se va a cerrar. En el momento que tu procedimiento de diálogo reciba este mensaje, tu ventana estará ya removida del escritorio. Este mensaje sólo es una notificación de que tu ventana ha sido destruida y de que debes prepararte para regresar a Windows. En respuesta a esto, puedes ejecutar alguna tarea doméstica antes de regresar a Windows. No queda más opción que quitar cuando la ventana llega a este estado. Si quieres tener la oportunidad de detener el usuario cuando éste intente cerrar la ventana, debes procesar el mensaje WM_CLOSE.

Ahora, regresando a WM_DESTROY, después de ejecutar la tareas domésticas, debes llamar al PostQuitMessage que enviará el mensaje WM_QUIT de vuelta a tu módulo. WM_QUIT hará que GetMessage regrese el valor NULL en eax, el cual terminará el bucle de mensajes y devolverá el control a Windows. Puedes enviar el mensaje WM_DESTROY a tu propio procedimiento de ventana llamando a la función DestroyWindow.

Tutorial 4: Pintando Textos

Teoría:

El texto en Windows es un tipo de objeto GUI. Cada carácter está compuesto por numerosos pixeles o puntos (dots) que están amontonados dentro de un patrones distintos. Por eso hablamos de "pintar" en vez de "escribir". Normalmente, pintas texto en tu propia area cliente (relamente, puedes también pintar fuera del area cliente, pero eso es otra historia). En Windows, poner texto en la pantalla es algo radicalmente distinto a DOS. En DOS, puedes pensar en la pantalla como una dimensión de 80x25. Pero en Windows, la pantalla es compartida por varios programas. Algunas reglas deben ser reforzadas para evitar que los programas escriban sobre la pantalla de otros. Windows asegura esto limitando el área para pintar de cada ventana a su área cliente solamente. El tamaño del area cliente de una ventana tampoco es constante. El usuario puede cambiarla en cualquier momento. Así que hay que determinar dinámicamente las dimensiones del área cliente de las ventanas.

Antes de que puedas pintar algo sobre el área cliente, debes pedir permiso a Windows. Eso es correcto, ya no tienes el control total sobre el monitor como lo tenías con DOS. Debes pedir permiso a Windows para pintar tu propia area cliente. Windows determinará el tamaño de tu área cliente, de la fuente, los colores y otros atributos GDI y regresará un manejador del contexto de dispositivo a tu programa.

Luego puedes emplear tu contexto de dispositivo como un pasaporte para pintar tu área cliente.

¿Qué es un contexto de dispositivo? Es sólo una estructura de datos que Windows mantiene en su interior. Un contexto de dispositivo está asociado con un dispositivo en particular, tal como una impresora o un monitor de video. Para un monitor de video, un contexto de dispositivo está normalmente asociado con una ventana particular en el monitor.

Algunos valores en el contexto de dispositivo son atributos gráficos como colores, fuentes etc. Estos son valores por defecto que se pueden cambiar a voluntad. Existen para ayudar a reducir la carga de tener que especificar estos atributos en todas las llamadas a funciones GDI.

Puedes pensar en un contexto de dispositivo como un ambiente por defecto preparado para tí por Windows. Luego puedes anular algunos de los elementos establecidos por defecto si quieres.

Cuando un programa necesita pintar, debe obtener un manejador al contexto de dispositivo. Normalmente, hay varias maneras de realizar esto.

llamar a BeginPaint en respuesta al mensaje WM_PAINT.
llamar a GetDC en respuesta a otros mensajes.
llamar a CreateDC para crear tu propio contexto de dispositivo

Hay algo que debes recordar para después de que tengas el manejador [handle] del contexto de dispositivo, y que debes realizar para el procesamiento de cualquier mensaje: no obtener el manejador en respuesta a un mensaje y emplearlo como respuesta a otro.

Windows envía mensajes WM_PAINT a la ventana para notificar que es ahora el momento de volver a pintar su área cliente. Windows no salva el contenido del área cliente de una ventana. En vez de eso, cuando ocurre una situación que garantiza que se va a volver a pintar el área cliente (tal como cuando una ventana ha sido cubierta por otra y luego descubierta), Windows pone el mensaje WM_PAINT en la cola de mensajes de ese programa. Es responsabilidad de Windows volver a pintar su propia área cliente. Debes reúnir toda la información sobre cómo volver a pintar el área cliente en la sección WM_PAINT de tu procedimiento de ventana, así que tu procedimiento de ventana puede volver a pintar tu area cliente cuando llega el mensaje WM_PAINT.

Otro concepto que debes tener en consideración es el de rectángulo inválido. Windows define un rectángulo inválido como el área rectangular más pequeña que el área cliente necesita para volver a ser pintada. Cuando Windows detecta un rectángulo inválido en el área cliente de una ventana, envía un mensaje WM_PAINT a esa ventana. En respuesta al mensaje WM_PAINT, la ventana puede obtener una estructura paintstruct que contiene, entre otras cosas, la coordenada del rectángulo inválido. Puedes llamar a BeginPaint en respuesta al mensaje WM_PAINT para validar el rectángulo inválido. Si no procesas el mensaje WM_PAINT, al menos debes llamar a DefWindowProc o a ValidateRect para validar el rectángulo inválido, sino Windows te enviará repetidamente el mensaje WM_PAINT.

Estos son los pasos que deberías realizar en respuesta a un mensaje WM_PAINT:

Obtener un manejador al contexto de dispositivo con BeginPaint.
Pintar el área cliente.
Liberar el manejador del contexto de dispositivo con EndPaint

Nota que no tienes que validar explícitamente el rectángulo inválido. Esto es realizado automáticamente por la llamada a BeginPaint. Entre las llamadas a BeginPaint y EndPaint, puedes llamar cualquiera de las funciones GDI para pintar tu área. Casi todas ellas requieren el manejador del contexto de dispositivo como parámetro.

Contenido:

Escribiremos un programa que despliega una cadena con el texto "Win32 assembly is great and easy!" en el centro del área cliente.

```
format  PE GUI 4.0
entry   start

include 'win32a.inc'        ; Incluímos definiciones de estructuras y constantes

cdXPos      EQU  128        ; Constante double X-Posición de la ventana(esq sup izqda)
cdYPos      EQU  128        ; Constante double Y-Posición de la ventana(esq sup izqda)
cdXSize     EQU  320        ; Constante double X-tamaño de la ventana
cdYSize     EQU  200        ; Constante double Y-tamaño de la ventana
cdColFondo  EQU  COLOR_WINDOW+1
cdVIcono    EQU  IDI_APPLICATION ; Icono de la ventana, véase Resource.H
cdVCursor   EQU  IDC_ARROW   ; Cursor para la ventana
; Tipo de ventana (Barra de cabecera)
; cdVBarTipo  EQU  WS_EX_TOOLWINDOW                 ; Tipo de barra de Cabecera: delgado, sin icono, sin reflejo en la barra de tareas
cdVBarTipo  EQU  NULL            ; Normal, con icono
cdVBtnTipo  EQU  WS_VISIBLE+WS_DLGFRAME+WS_SYSMENU    ; Normal sólo con botón cerrar
; cdVBtnTipo     EQU  WS_OVERLAPPEDWINDOW             ; Normal sólo con los tres botones
; Constantes para la subventana del texto
cdVCarText  EQU  WS_CHILD + WS_VISIBLE + SS_CENTER
cdTXPos     EQU  15     ; Constante double X-Posición subventana para el texto(esq sup izqda)
cdTYPos     EQU  30     ; Constante double Y-Posición subventana para el texto(esq sup izqda)
```

```
cdTXSize    EQU  cdXSize-3*cdTXPos;[rct + RECT.right]        ; Constante double X-tamaño de la subventana para el texto
cdTYSize    EQU  40;[rct + RECT.bottom]       ; Constante double Y-tamaño de la subventana para el texto
cdTipoSubV  EQU  NULL          ; Tipo de subventana (flat-NULL, 3D-1, etc.)



section '.text' code readable executable
 start:
   invoke    GetModuleHandle, NULL
   mov       [wc.hInstance], eax
   mov       [wc.lpfnWndProc], WndProc
   mov       [wc.lpszClassName], NombreClase
   mov       [wc.hbrBackground], COLOR_WINDOW+1
   stdcall   WinMain, [wc.hInstance], NULL, NULL, SW_SHOWNORMAL
   invoke    ExitProcess, [wMsg.wParam]

proc WinMain uses ebx esi edi, hInst, hPrevInst, CmdLine, CmdShow
   ;  Propósito: Inicializamos la ventana principal de la aplicación y captura errores, si los hubiere
   ;  Entrada  : hInst, hPrevInst, CmdLine, CmdShow
   ;  Salida   : Ninguna
   ;  Destruye : Ninguna
   invoke    LoadIcon,0,cdVIcono
   mov       edx, eax
   mov       eax, [hInst]
   mov       ebx, NombreClase
   mov       ecx, WndProc
   mov       [wc.hInstance], eax
   mov       [wc.lpszClassName], ebx
   mov       [wc.lpfnWndProc], ecx
```

```
 mov      [wc.hIcon], edx

 invoke   LoadCursor,0,cdVCursor
 mov      [wc.hCursor],eax

 invoke   RegisterClass,wc
 test     eax,eax
 jz       error

 invoke   CreateWindowEx,cdVBarTipo,NombreClase,MsgCabecera,\
          cdVBtnTipo,cdXPos, cdYPos, cdXSize, cdYSize,\
          NULL,NULL,[wc.hInstance],NULL
 mov      [wHMain],eax
 test     eax,eax
 jz       error

 mov      [hWnd], eax

 invoke   ShowWindow, dword hWnd, dword SW_SHOWNORMAL
 invoke   UpdateWindow, dword hWnd

msg_loop:
 invoke   GetMessage,wMsg,NULL,0,0
 cmp      eax,1
 jb       end_loop
 jne      msg_loop
 invoke   TranslateMessage,wMsg
 invoke   DispatchMessage,wMsg
jmp      msg_loop
```

```
 error:
   invoke    MessageBox,NULL,MsgError,NULL,MB_ICONERROR+MB_OK

 end_loop:
   mov       eax, [wMsg.wParam]
   ret
endp

proc WndProc,hWnd,uMsg,wParam,lParam
   push      ebx esi edi
   cmp       [uMsg],WM_DESTROY
   je        wmDESTROY
   cmp       [uMsg],WM_PAINT
   je        wmPAINT

   wmDEFAULT:
     invoke    DefWindowProc,[hWnd],[uMsg],[wParam],[lParam]
     jmp       wmBYE
   wmPAINT:
     invoke    BeginPaint,[hWnd],ps
     mov       [expDc],eax
     invoke    GetClientRect,[hWnd],rct
     invoke    DrawText,[expDc], OurText, -1, rct, DT_SINGLELINE or DT_CENTER or DT_VCENTER
     invoke    EndPaint,[hWnd],ps
     jmp       wmBYE
   wmDESTROY:
     invoke    PostQuitMessage,0
   wmBYE:
```

```
    pop       edi esi ebx
    ret
endp

section '.data' data readable writeable
  wHMain       rd    1
  hWnd         rd    1

  MsgCabecera  db    'Escribimos en nuestra ventana (FASM)',0
  NombreClase  db    'SimpleWinClass',0

  MsgError     db    'Carga inicial fallida.',0
  expDc        rd    1


  wMsg         MSG
  wc           WNDCLASS

  ps           PAINTSTRUCT
  rct          RECT
  OurText  db "Win32 assembly is great and easy!",0


section '.idata' import data readable writable

  library kernel32,'KERNEL32.DLL',\    ; Importamos las bibliotecas para que el enlazador pueda
trabajar
          user32,'USER32.DLL'          ; Importamos las bibliotecas para que el enlazador pueda trabajar
```

```
include 'api\kernel32.inc'          ; KERNEL32 API calls

include 'api\user32.inc'            ; USER32 API calls
```

RECT Struct
  left LONG ?
  top LONG ?
  right LONG ?
  bottom LONG ?
RECT ends

left y top son las coordenadas de la esquina izquierda superior de un rectángulo. right y bottom son las coordenadas de la esquina derecha inferior. Debe recordarse que: El origen de los ejes x-y está en la esquina superior izquierda. Entonces el punto y=10 está DEBAJO del punto y=0.

```
invoke   BeginPaint,[hWnd],ps

   mov      [expDc],eax

   invoke   GetClientRect,[hWnd],rct

   invoke   DrawText,[expDc], OurText, -1, rct, DT_SINGLELINE or DT_CENTER or DT_VCENTER

   invoke   EndPaint,[hWnd],ps
```

En respuesta al mensaje WM_PAINT, llamas a BeginPaint pasando como parámetros al manejador de la ventana que quieres pintar y una estructura PAINTSTRUCT no inicializada. Después de una llamada exitosa, eax contiene el manejador al contexto de dispositivo. Luego llamas a GetClientRect para recobrar la dimensión del área cliente. La dimensión es regresada en la variable rect variable que tú pasas a DrawText como uno de sus parámetros. La sintaxis de DrawText es:

DrawText proto hdc:HDC, lpString:DWORD, nCount:DWORD, lpRect:DWORD, uFormat:DWORD

DrawText es una función de la API de alto-nivel para salida de texto. Maneja algunos detalles tales como ajuste de línea, centramiento, etc. así que puedes concentrarte sólo en la cadena que quieres pintar. Su hermana de bajo nivel, TextOut, será examinada en el próximo tutorial. DrawText formatea una cadena de texto para fijar dentro de los límites de un rectángulo. Emplea la fuente seleccionada en el momento, color y fondo (en el contexto de dispositivo) para dibujar texto. Las líneas son ajustadas para fijarla dentro de los límites del rectángulo. Regresa la altura del texto de salida en unidades de dispositivo, en nuestro caso, pixeles. Veamos sus parámetros:

hdc manejador al contexto de dispositivo
lpString El puntero a la cadena que quieres dibujar en el rectángulo. La cadena debe estar terminada en NULL o sino tendrás que especificar su largo en el parámetro de texto, nCount.
nCount El número de caracteres para salida. Si la cadena es terminada en cero, nCount debe ser -1. De otra manera nCount debe contener el número de caracteres en la cadena que quieres dibujar.
lpRect El puntero al rectángulo (una estructura de tipo RECT) donde quieres dibujar la cadena. Nota que este rectángulo también es un rectángulo recortante [clipping rectangule], es decir, no podrás dibujar la cadena fuera del rectángulo.

uFormat El valor que especifica como la cadena es desplegada en el rectángulo. Usamos tres valores combinados por el operador "or":

- **DT_SINGLELINE** especifica una línea de texto
- **DT_CENTER** centra el texto horizontalmante.
- **DT_VCENTER** centra el texto verticalmente. Debe ser usado con DT_SINGLELINE.

Después de terminar de pintar el área cliente, debes llamar a la función EndPaint para liberar el manejador del contexto de dispositivo.

Eso es todo. Podemos hacer un sumario de los puntos relevantes:

- Llamas a BeginPaint-EndPaint en respuesta la mensaje WM_PAINT. Haces lo que gustes con el área cliente de la ventana entre las llamadas a las funciones BeginPaint y EndPaint.
- Si quieres volver a pintar tu área cliente en respuesta a otros mensajes, tienes dos posibilidades:
  - Usar el par GetDC-ReleaseDC y pintar entre estas dos llamadas
  - Llamar a InvalidateRect o a UpdateWindow para invalidar toda el área cliente, forzando a Windows a que ponga un mensaje WM_PAINT en la cola de mensajes de tu ventana y pinte durante la sección WM_PAINT

Tutorial 5: Más sobre Textos

Teoría:

El sistema de colores de Windows está basado en valores RGB, R=red (rojo), G=Green (verde), B=Blue (azul). Si quieres especificar un color en Windows, debes establecer el color que desees en términos de estos tres colores mayores. Cada valor de color tiene un rango desde 0 a 255 (un valor de un byte). Por ejemplo, si quieres un color rojo puro, deberías usar 255,0,0. O si quieres un color blanco puro, debes usar 255,255,255. Puedes ver en los ejemplos que obtener el color que necesitas es muy difícil con este sistema ya que tienes que tener una buena comprensión de como mezclar y hacer corresponder los colores.

Para el color del texto y del fondo, usas SetTextColor y SetBkColor.

Puedes "crear" una fuente llamando a CreateFont o a CreateFontIndirect. La diferencia entre las dos funciones es que CreateFontIndirect recibe sólo un parámetro: un puntero a la estructura lógica de la fuente, LOGFONT. CreateFontIndirect es la más flexible de las dos, especialmente si tus programas necesitan cambiar de fuentes con frecuencia. Sin embargo, como en nuestro ejemplo "crearemos" sólo una fuente para demostración, podemos hacerlos con CreateFont. Después de llamada a CreateFont, regresará un manejador a la fuente que debes seleccionar dentro del contexto de dispositivo. Después de eso, toda función de texto de la API usará la fuente que hemos seleccionado dentro del contexto de dispositivo.

```
format  PE GUI 4.0
entry   start

include 'win32a.inc'
                ; Incluímos definiciones de estructuras y constantes

cdXPos      EQU  128        ; Constante double X-Posición de la ventana(esq sup izqda)
cdYPos      EQU  128        ; Constante double Y-Posición de la ventana(esq sup izqda)
cdXSize     EQU  720        ; Constante double X-tamaño de la ventana
cdYSize     EQU  300        ; Constante double Y-tamaño de la ventana
cdColFondo  EQU  COLOR_WINDOW+1
cdVIcono    EQU  IDI_APPLICATION ; Icono de la ventana, véase Resource.H
cdVCursor   EQU  IDC_ARROW   ; Cursor para la ventana
; Tipo de ventana (Barra de cabecera)
; cdVBarTipo  EQU  WS_EX_TOOLWINDOW                 ; Tipo de barra de Cabecera: delgado, sin icono, sin reflejo en la barra de tareas
cdVBarTipo  EQU  NULL           ; Normal, con icono
cdVBtnTipo  EQU  WS_VISIBLE+WS_DLGFRAME+WS_SYSMENU    ; Normal sólo con botón cerrar
; cdVBtnTipo     EQU  WS_OVERLAPPEDWINDOW             ; Normal sólo con los tres botones
; Constantes para la subventana del texto
cdVCarText  EQU  WS_CHILD + WS_VISIBLE + SS_CENTER
cdTXPos     EQU  15     ; Constante double X-Posición subventana para el texto(esq sup izqda)
cdTYPos     EQU  30     ; Constante double Y-Posición subventana para el texto(esq sup izqda)
cdTXSize    EQU  cdXSize-3*cdTXPos;[rct + RECT.right]       ; Constante double X-tamaño de la subventana para el texto
cdTYSize    EQU  40;[rct + RECT.bottom]       ; Constante double Y-tamaño de la subventana para el texto
cdTipoSubV  EQU  NULL           ; Tipo de subventana (flat-NULL, 3D-1, etc.)
```

```
section '.text' code readable executable
  start:
    invoke    GetModuleHandle, NULL
    mov       [wc.hInstance], eax
    mov       [wc.lpfnWndProc], WndProc
    mov       [wc.lpszClassName], NombreClase
    mov       [wc.hbrBackground], COLOR_WINDOW+1
    stdcall   WinMain, [wc.hInstance], NULL, NULL, SW_SHOWNORMAL
    invoke    ExitProcess, [wMsg.wParam]

proc WinMain uses ebx esi edi, hInst, hPrevInst, CmdLine, CmdShow
    ;  Propósito: Inicializamos la ventana principal de la aplicación y captura errores, si los hubiere
    ;  Entrada  : hInst, hPrevInst, CmdLine, CmdShow
    ;  Salida   : Ninguna
    ;  Destruye : Ninguna
    invoke    LoadIcon,0,cdVIcono
    mov       edx, eax
    mov       eax, [hInst]
    mov       ebx, NombreClase
    mov       ecx, WndProc
    mov       [wc.hInstance], eax
    mov       [wc.lpszClassName], ebx
    mov       [wc.lpfnWndProc], ecx
    mov       [wc.hIcon], edx

    invoke    LoadCursor,0,cdVCursor
    mov       [wc.hCursor],eax
```

```
  invoke    RegisterClass,wc
  test      eax,eax
  jz        error

  invoke    CreateWindowEx,cdVBarTipo,NombreClase,MsgCabecera,\
            cdVBtnTipo,cdXPos, cdYPos, cdXSize, cdYSize,\
            NULL,NULL,[wc.hInstance],NULL
  mov       [wHMain],eax
  test      eax,eax
  jz        error

  mov       [hWnd], eax

  invoke    ShowWindow, dword hWnd, dword SW_SHOWNORMAL
  invoke    UpdateWindow, dword hWnd

msg_loop:
  invoke    GetMessage,wMsg,NULL,0,0
  cmp       eax,1
  jb        end_loop
  jne       msg_loop
  invoke    TranslateMessage,wMsg
  invoke    DispatchMessage,wMsg
jmp       msg_loop

error:
  invoke    MessageBox,NULL,MsgError,NULL,MB_ICONERROR+MB_OK
```

```
 end_loop:
   mov      eax, [wMsg.wParam]
   ret
endp

proc WndProc,hWnd,uMsg,wParam,lParam
   push     ebx esi edi
   cmp      [uMsg],WM_DESTROY
   je       wmDESTROY
   cmp      [uMsg],WM_PAINT
   je       wmPAINT

   wmDEFAULT:
    invoke    DefWindowProc,[hWnd],[uMsg],[wParam],[lParam]
    jmp       wmBYE
   wmPAINT:
    invoke    BeginPaint,[hWnd],ps
    mov       [expDc],eax
     invoke CreateFont,24,16,0,0,400,0,0,0,OEM_CHARSET,\
                           OUT_DEFAULT_PRECIS,CLIP_DEFAULT_PRECIS,\
                           DEFAULT_QUALITY,DEFAULT_PITCH or FF_SCRIPT,\
                            FontName
      invoke SelectObject, [expDc], eax
      invoke SetTextColor, [expDc],0x32C8C8;RGB=50,200,200
      invoke SetBkColor, [expDc],0xFF0000;RGB=0,0,255
      invoke TextOut,[expDc],0,0,OurText,len
      invoke SelectObject, [expDc], ps
    invoke    EndPaint,[hWnd],ps
    jmp       wmBYE
```

```
   wmDESTROY:
     invoke    PostQuitMessage,0
   wmBYE:
   pop       edi esi ebx
   ret
endp

section '.data' data readable writeable
  wHMain       rd    1
  hWnd         rd    1

  MsgCabecera  db    'Escribimos en nuestra ventana (FASM)',0
  NombreClase  db    'SimpleWinClass',0

  MsgError     db    'Carga inicial fallida.',0
  expDc        rd    1


  wMsg         MSG
  wc           WNDCLASS

  ps           PAINTSTRUCT
  rct          RECT
 ; hfont       HFONT
  OurText  db "Win32 assembly is great and easy!",0
  len =$-OurText
  FontName db "script",0

section '.idata' import data readable writable
```

```
 library kernel32,'KERNEL32.DLL',\    ; Importamos las bibliotecas para que el enlazador pueda trabajar
        user32,'USER32.DLL',\
        gdi32,'GDI32.DLL'         ; Importamos las bibliotecas para que el enlazador pueda trabajar


 include 'api\kernel32.inc'          ; KERNEL32 API calls
 include 'api\user32.inc'            ; USER32 API calls
 include 'api\gdi32.inc'
```

Análisis:

```
invoke CreateFont,24,16,0,0,400,0,0,0,OEM_CHARSET,\
                        OUT_DEFAULT_PRECIS,CLIP_DEFAULT_PRECIS,\
                        DEFAULT_QUALITY,DEFAULT_PITCH or FF_SCRIPT,\
                         FontName
```

CreateFont creará la fuente lógica que más coincida con los parámetros dados y con los datos de fuentes disponibles. Esta función tiene más parámetros que cualquier otra función en Windows. Regresa un manejador a la fuente lógica a ser usada por la función SelectObject. Examinaremos sus parámetros en detalle.

```
CreateFont proto nHeight:DWORD,\
                 nWidth:DWORD,\
                 nEscapement:DWORD,\
                 nOrientation:DWORD,\
                 nWeight:DWORD,\
                 cItalic:DWORD,\
                 cUnderline:DWORD,\
                 cStrikeOut:DWORD,\
                 cCharSet:DWORD,\
                 cOutputPrecision:DWORD,\
                 cClipPrecision:DWORD,\
                 cQuality:DWORD,\
                 cPitchAndFamily:DWORD,\
                 lpFacename:DWORD
```

nHeight La altura deseada para los caracteres. 0 significa el tamaño por defecto.
nWidth La ancho deseado para los caracteres. Normalmente este valor debería ser 0 que permite a Windows coordinar el ancho y el alto. Sin embargo, en nuestro ejemplo, el ancho por defecto hace difícil la lectura del texto, así que usaremos mejor un ancho de 16.

nEscapement Especifica la orientación del próximo caracter de salida relativa al previo en décimas de grados. Normalmente, se pone en 0. Si se pone en 900 tendremos que todos los caracteres irán por encima del primer caracter, 1800 los escribirá hacia atrás, o 2700 para escribir cada caracter desde abajo.
nOrientation Especifica cuánto debería ser rotado el caracter cuando tiene una salida en décimas de grados. Si se pone en 900 todos los caracteres reposaran sobre sus respaldos, 1800 se emplea para escribirlos upside-down, etc.
nWeight Establece el grosor de las líneas de cada caracter. Windows define los siguientes tamaños:

FW_DONTCARE equ 0
FW_THIN equ 100
FW_EXTRALIGHT equ 200
FW_ULTRALIGHT equ 200
FW_LIGHT equ 300
FW_NORMAL equ 400
FW_REGULAR equ 400
FW_MEDIUM equ 500
FW_SEMIBOLD equ 600
FW_DEMIBOLD equ 600
FW_BOLD equ 700
FW_EXTRABOLD equ 800
FW_ULTRABOLD equ 800
FW_HEAVY equ 900
FW_BLACK equ 900

cItalic 0 para normal, cualquier otro valor para caracteres en itálicas.
cUnderline 0 para normal, cualquier otro valor para caracteres subrayados.
cStrikeOut 0 para normal, cualquier otro valor para caracteres con una línea a través del centro.
cCharSet Especifica la configuración de la fuente [character set]. Normalmente debería ser OEM_CHARSET lo cual permite a Windows seleccionar la fuente dependiente del sistema operativo.
cOutputPrecision Especifica cuánto la fuente seleccionada debe coincidir con las características que queremos. Normalmente debería ser OUT_DEFAULT_PRECIS que define la conducta de proyección por defecto de la fuente.
cClipPrecision Especifica la presisión del corte. La precisión del corte define cómo recortar los caracteres que son parcialmente fuera de la región de recorte. Deberías poder obtenerlo con CLIP_DEFAULT_PRECIS que define la conducta por defecto del recorte.
cQuality Especifiica la cualidad de la salida. La cualidad de salida define cuán cuidadosamente la GDI debe intentar hacer coincidir los atributos de la fuente lógica con los de la fuente física actual. Hay tres posibilidades: DEFAULT_QUALITY, PROOF_QUALITY and DRAFT_QUALITY.
cPitchAndFamily Especifiica la pitch y familia de la fuente. Debes combinar el valor pitch value y el valor de familia con el operador "or".
lpFacename Un puntero a una cadena terminada en cero que especifica la tipografía de la fuente.

La descripción anterior no tiene nada de comprensible. Deberías revisar la referencia de la APi de Win32 API para más detalles.

invoke TextOut,[expDc],0,0,OurText,len

Llama a la función TextOut para dibujar el texto sobre el área cliente. El texto estará en la fuente y el color que especificamos previamente.

invoke SelectObject,[expDc], ps

Cuando estemos trabajando con fuentes, deberíamos almacenar la fuente original dentro del contexto de dispositivo. Deberías almacenar siempre el objeto que reemplazaste en el contexto de dispositivo.

Tutorial 6: Entrada del teclado

Teoría:

Como normalmente sólo hay un teclado para cada PC, todos los programas de Windows deben compartirlo entre sí. Windows es responsable de enviar los golpes de tecla a la ventana que tiene el foco de entrada.

Aunque pueden haber varias ventanas en el monitor, sólo una de ellas tiene el foco de entrada. La ventana que tiene el foco de entrada es la única que puede recibir los golpes de tecla. Puedes diferenciar la ventana que tiene el foco de entrada de las otras ventanas observando la barra de título. La barra de título del programa que tiene el foco está iluminada.

Realmente, hay dos tipos principales de mensajes de teclado, dependiendo de tu punto de vista sobre el teclado. Puedes ver el teclado como una colección de teclas. En este caso, si presionas una tecla, Windows envía un mensaje WM_KEYDOWN a la ventana que tiene el foco de entrada, que notifica que una tecla ha sido presionada. Cuando sueltas la tecla, Windows envía un mensaje WM_KEYUP. Tú tratas a las teclas como si fueran botones.

Otra manera de ver el teclado es como un dispositivo de entrada de caracteres. Cuando presionas "una" tecla, Windows envía un mensaje WM_CHAR a la ventana que tiene el foco de entrada, diciéndole que el usuario envía "un" carácter a ella. En realidad, Windows envía mensajes WM_KEYDOWN y WM_KEYUP a la ventana que tiene el foco de entrada y esos mensajes serán traducidos a mensajes WM_CHAR por una llamada a TranslateMessage. El procedimiento de ventana puede decidir si procesa los tres mensajes o sólo los mensajes que interesan. Muchas veces, podrás ignorar WM_KEYDOWN y WM_KEYUP ya que la función TranslateMessage en el bucle de mensajes traduce los mensajes WM_KEYDOWN y WM_KEYUP a mensajes WM_CHAR. En este tutorial nos concentraremos en WM_CHAR.

format PE GUI 4.0

```
entry   start

include 'win32a.inc'
                ; Incluímos definiciones de estructuras y constantes

cdXPos      EQU  128        ; Constante double X-Posición de la ventana(esq sup izqda)
cdYPos      EQU  128        ; Constante double Y-Posición de la ventana(esq sup izqda)
cdXSize     EQU  720        ; Constante double X-tamaño de la ventana
cdYSize     EQU  300        ; Constante double Y-tamaño de la ventana
cdColFondo  EQU  COLOR_WINDOW+1
cdVIcono    EQU  IDI_APPLICATION ; Icono de la ventana, véase Resource.H
cdVCursor   EQU  IDC_ARROW   ; Cursor para la ventana
; Tipo de ventana (Barra de cabecera)
; cdVBarTipo  EQU  WS_EX_TOOLWINDOW                 ; Tipo de barra de Cabecera: delgado, sin icono, sin reflejo en la barra de tareas
cdVBarTipo  EQU  NULL           ; Normal, con icono
cdVBtnTipo  EQU  WS_VISIBLE+WS_DLGFRAME+WS_SYSMENU    ; Normal sólo con botón cerrar
; cdVBtnTipo     EQU  WS_OVERLAPPEDWINDOW             ; Normal sólo con los tres botones
; Constantes para la subventana del texto
cdVCarText  EQU  WS_CHILD + WS_VISIBLE + SS_CENTER
cdTXPos     EQU  15     ; Constante double X-Posición subventana para el texto(esq sup izqda)
cdTYPos     EQU  30     ; Constante double Y-Posición subventana para el texto(esq sup izqda)
cdTXSize    EQU  cdXSize-3*cdTXPos;[rct + RECT.right]       ; Constante double X-tamaño de la subventana para el texto
cdTYSize    EQU  40;[rct + RECT.bottom]       ; Constante double Y-tamaño de la subventana para el texto
cdTipoSubV  EQU  NULL         ; Tipo de subventana (flat-NULL, 3D-1, etc.)


section '.text' code readable executable
```

```
start:
  invoke   GetModuleHandle, NULL
  mov      [wc.hInstance], eax
  mov      [wc.lpfnWndProc], WndProc
  mov      [wc.lpszClassName], NombreClase
  mov      [wc.hbrBackground], COLOR_WINDOW+1
  stdcall  WinMain, [wc.hInstance], NULL, NULL, SW_SHOWNORMAL
  invoke   ExitProcess, [wMsg.wParam]

proc WinMain uses ebx esi edi, hInst, hPrevInst, CmdLine, CmdShow
  ;  Propósito: Inicializamos la ventana principal de la aplicación y captura errores, si los hubiere
  ;  Entrada  : hInst, hPrevInst, CmdLine, CmdShow
  ;  Salida   : Ninguna
  ;  Destruye : Ninguna
  invoke   LoadIcon,0,cdVIcono
  mov      edx, eax
  mov      eax, [hInst]
  mov      ebx, NombreClase
  mov      ecx, WndProc
  mov      [wc.hInstance], eax
  mov      [wc.lpszClassName], ebx
  mov      [wc.lpfnWndProc], ecx
  mov      [wc.hIcon], edx

  invoke   LoadCursor,0,cdVCursor
  mov      [wc.hCursor],eax

  invoke   RegisterClass,wc
  test     eax,eax
```

```
  jz       error

  invoke    CreateWindowEx,cdVBarTipo,NombreClase,MsgCabecera,\
            cdVBtnTipo,cdXPos, cdYPos, cdXSize, cdYSize,\
            NULL,NULL,[wc.hInstance],NULL
  mov       [wHMain],eax
  test      eax,eax
  jz       error

  mov       [hWnd], eax

  invoke    ShowWindow, dword hWnd, dword SW_SHOWNORMAL
  invoke    UpdateWindow, dword hWnd

msg_loop:
  invoke    GetMessage,wMsg,NULL,0,0
  cmp       eax,1
  jb       end_loop
  jne       msg_loop
  invoke    TranslateMessage,wMsg
  invoke    DispatchMessage,wMsg
jmp       msg_loop

error:
  invoke    MessageBox,NULL,MsgError,NULL,MB_ICONERROR+MB_OK

end_loop:
  mov       eax, [wMsg.wParam]
  ret
```

```
endp

proc WndProc,hWnd,uMsg,wParam,lParam
   push     ebx esi edi
   cmp      [uMsg],WM_DESTROY
   je       wmDESTROY
   cmp      [uMsg],WM_PAINT
   je       wmPAINT
   cmp      [uMsg],WM_CHAR
   je       wmCHAR
   wmDEFAULT:
    invoke    DefWindowProc,[hWnd],[uMsg],[wParam],[lParam]
    jmp       wmBYE
  wmCHAR:
      push dword [wParam]
      pop dword [char]
      invoke InvalidateRect,[hWnd],NULL,TRUE
      jmp wmBYE
   wmPAINT:
      invoke    BeginPaint,[hWnd],ps
      mov       [expDc],eax
      invoke TextOut,[expDc],0,0,char,1
      invoke SelectObject, [expDc], ps
    invoke    EndPaint,[hWnd],ps
    jmp       wmBYE
   wmDESTROY:
    invoke    PostQuitMessage,0
   wmBYE:
   pop       edi esi ebx
```

```
   ret
endp

section '.data' data readable writeable
  wHMain       rd    1
  hWnd         rd    1

  MsgCabecera  db    'Escribimos en nuestra ventana (FASM)',0
  NombreClase  db    'SimpleWinClass',0

  MsgError     db    'Carga inicial fallida.',0
  expDc        rd    1


  wMsg         MSG
  wc           WNDCLASS

  ps           PAINTSTRUCT
  rct          RECT
  char dd ?                    ; el caracter que el programa recibe del teclado

  FontName db "script",0

section '.idata' import data readable writable

  library kernel32,'KERNEL32.DLL',\    ; Importamos las bibliotecas para que el enlazador pueda
trabajar
        user32,'USER32.DLL',\
        gdi32,'GDI32.DLL'          ; Importamos las bibliotecas para que el enlazador pueda trabajar
```

```
include 'api\kernel32.inc'        ; KERNEL32 API calls
include 'api\user32.inc'          ; USER32 API calls
include 'api\gdi32.inc'
```

wmCHAR:

```
    push dword [wParam]
    pop dword [char]
    invoke InvalidateRect,[hWnd],NULL,TRUE
```

Esto es agregado al manejador del mensaje WM_CHAR en el procedimiento de ventana. Pone el caracter dentro de la variable llamada "char" y luego llama a InvalidateRect. InvalidateRect hace que le rectángulo específico en el área cliente quede invalidado para forzar a Windows para que envíe el mensaje WM_PAINT al procedimiento de ventana. Su sintaxis es como sigue:

```
InvalidateRect proto hWnd:HWND,\
                     lpRect:DWORD,\
                     bErase:DWORD
```

lpRect es un puntero al rectángulo en el área cliente que queremos declarar inválida. Si este parámetro es nulo, toda el área cliente será marcada como inválida.
bErase es una bandera que dice a Windows si necesita borrar el fondo. Su ventana es TRUE, luego Windows borrará el fondo del rectángulo invalidado cuando se llama a BeginPaint.

Así que la estrategia que usamos aquí es: almacenamos toda la información necesaria involucrada en la acción de pintar el área cliente y generar el mensaje WM_PAINT para pintar el área cliente. Por supuesto, el código en la sección WM_PAINT debe saber de antemano qué se espera de ella. Esto parece una manera indirecta de hacer las cosas, pero así es como lo hace Windows.

Realmente podemos pintar el área cliente durante el proceso del mensaje WM_CHAR llamando el par de funciones GetDC y ReleaseDC. No hay problema. Pero lo gracioso comienza cuando nuestra ventana necesita volver a pintar su área cliente. Como el código que pinta el carácter está en la sección WM_CHAR, el procedimiento de ventana no será capaz de pintar nuestro carácter en el área cliente. Así que la línea de abajo es: poner todos el código y los datos necesarios para que realicen la acción de pintar en WM_PAINT. Puedes enviar el mensaje WM_PAINT desde cualquier lugar de tu código cada vez que quieras volver a pintar el área cliente.

```
    invoke TextOut,[expDc],0,0,char,1
```

Cuando se llama a InvalidateRect, envía un mensaje WM_PAINT de regreso al procedimiento de ventana. De esta manera es llamado el código en la sección WM_PAINT. Llama a BeginPaint como es usual para obtener el manejador al contexto del dispositivo y luego llama a TextOut que dibuja

nuestro carácter en el área cliente en x=0, y=0. Cuando corres el programa y presionas cualquier tecla, verás un "eco" [echo] del carácter en la esquina izquierda superior del área cliente de la ventana. Y cuando la ventana sea minimizada, al ser maximizada de nuevo tendrá todavía el carácter ahí ya que todo el código y los datos esenciales para volver a pintar son todos activados en la sección WM_PAINT.

Tutorial 7: Entrada del Ratón

Teoría:

Como con la entrada del teclado, Windows detecta y envía notificaciones sobre las actividades del ratón que son relevantes para las ventanas. Esas actividades incluyen los clicks de los botones izquierdo y derecho del ratón, el movimiento del cursor del ratón sobre la ventana, doble clicks. A diferencia de la entrada del teclado, que es dirigida a la ventana que tiene el foco de entrada, los mensajes del ratón son enviados a cualquier ventana sobre la cual esté el cursor del ratón, activo o no. Además, también hay mensajes del ratón sobre el área no cliente. Pero la mayoría de las veces, afortunadamente podemos ignorarlas. Podemos concentrarnos en los mensajes relacionados con el área cliente.

Hay dos mensajes para cada botón el ratón: los mensajes WM_LBUTTONDOWN, WM_RBUTTONDOWN y WM_LBUTTONUP, WM_RBUTTONUP. Para un ratón con tres botones, están también WM_MBUTTONDOWN and WM_MBUTTONUP. Cuando el cursor del ratón se mueve sobre el área cliente, Windows envía mensajes WM_MOUSEMOVE a la ventana debajo del cursor. Una ventana puede recibir mensajes de doble clicks, WM_LBUTTONDBCLK o WM_RBUTTONDBCLK, ***si y sólo si*** *la clase de su ventana tiene activada la bandera correspondiente al estilo* CS_DBLCLKS, sino la ventana recibirá sólo una serie de mensajes del topo botón del ratón arriba o abajo.

Para todos estos mensajes, el valor de lParam contiene la posición del ratón. La palabra [word] baja es la coordenada 'x', y la palabra alta es la coordenada 'y' relativa a la esquina izquierda superior del área cliente de la ventana. wParam indica el estado de los botones del ratón y de las teclas Shift y Ctrl.

Ejemplo:

```
format PE GUI 4.0

entry start


include 'win32a.inc'        ; Incluímos definiciones de estructuras y constantes

;WinMain proto dword,dword,dword,dword
```

```
cdXPos       EQU  128        ; Constante double X-Posición de la ventana(esq sup izqda)
cdYPos       EQU  128        ; Constante double Y-Posición de la ventana(esq sup izqda)
cdXSize      EQU  320        ; Constante double X-tamaño de la ventana
cdYSize      EQU  200        ; Constante double Y-tamaño de la ventana
cdColFondo   EQU  COLOR_BTNFACE + 1  ; Color de fondo de la ventana: gris de un botón de comando
cdVIcono     EQU  IDI_APPLICATION ; Icono de la ventana, véase Resource.H
cdVCursor    EQU  IDC_ARROW   ; Cursor para la ventana
; Tipo de ventana (Barra de cabecera)
;cdVBarTipo  EQU  WS_EX_TOOLWINDOW                ; Tipo de barra de Cabecera: delgado, sin icono, sin reflejo en barra de tareas
cdVBarTipo   EQU  NULL                           ; Normal, con icono
cdVBtnTipo   EQU  WS_VISIBLE+WS_DLGFRAME+WS_SYSMENU  ; Normal sólo con botón cerrar
; cdVBtnTipo    EQU  WS_OVERLAPPEDWINDOW          ; Normal sólo con los tres botones

section '.text' code readable executable
 start:
   invoke   GetModuleHandle,0
   mov      [wc.hInstance],eax
   invoke   GetCommandLine
   mov      [CommandLine], EAX
   stdcall  WinMain, [wc.hInstance], NULL, [CommandLine], SW_SHOWDEFAULT
   invoke   ExitProcess,[msg.wParam]

proc WinMain uses ebx esi edi, hInst, hPrevInst, CmdLine, CmdShow
   ;  Propósito: Inicializamos la ventana principal de la aplicación y captura errores, si los hubiere
   ;  Entrada  : hInst, hPrevInst, CmdLine, CmdShow
   ;  Salida   : Ninguna
   ;  Destruye : Ninguna
```

```
 invoke    LoadIcon,0,cdVIcono
 mov       [wc.hIcon],eax
 invoke    LoadCursor,0,cdVCursor
 mov       [wc.hCursor],eax
 invoke    RegisterClass,wc
 test      eax,eax
 jz        error

 invoke    CreateWindowEx,cdVBarTipo,NombreClase,MsgCabecera,\
           cdVBtnTipo,cdXPos, cdYPos, cdXSize, cdYSize,\
           NULL,NULL,[wc.hInstance],NULL
 test      eax,eax
 jz        error

 msg_loop:
   invoke    GetMessage,msg,NULL,0,0
   cmp       eax,1
   jb        end_loop
   jne       msg_loop
   invoke    TranslateMessage,msg
   invoke    DispatchMessage,msg
 jmp       msg_loop

error:
 invoke    MessageBox,NULL,MsgError,NULL,MB_ICONERROR+MB_OK

end_loop:
 MOV       EAX, [msg.wParam]
 ret
```

```
endp

proc WndProc uses ebx esi edi, hwnd,wmsg,wparam,lparam
   ;  Propósito: Procesa los mensajes provenientes de las ventanas
   ;  Entrada  : hwnd,wmsg,wparam,lparam
   ;  Salida   : Ninguna
   ;  Destruye : Ninguna
   cmp      [wmsg],WM_DESTROY
   je       .wmdestroy
  ; cmp      [wmsg],WM_MOUSEMOVE
  ; je       .wmmousemove
   cmp      [wmsg],WM_PAINT
   je       .wmpaint
   cmp      [wmsg],WM_LBUTTONDOWN
   je       .wmLBUTTONDOWN

 .defwndproc:
   invoke    DefWindowProc,[hwnd],[wmsg],[wparam],[lparam]
   jmp      .finish
 .wmLBUTTONDOWN:
   mov      eax, [lparam]
   mov      edx, eax
   and      eax, 0ffffh
   inc      eax
   mov      [xPos], eax
   shr      edx, 16
   inc      edx
   mov      [yPos], edx
    mov [mouseClick],TRUE
```

```
    invoke    InvalidateRect,[hwnd],NULL,TRUE
    xor       eax,eax
    jmp       .finish
  .wmpaint:
    invoke    BeginPaint,[hwnd], Ps
    mov       [hDC],eax
    invoke    lstrlen,MsgRaton
    cmp [mouseClick],0
     je .seguir
        invoke    TextOut,[hDC],[xPos],[yPos],MsgRaton,eax
    .seguir:
    invoke    EndPaint, [hwnd], Ps
    xor       eax,eax
    jmp       .finish
  .wmdestroy:
    invoke    PostQuitMessage,0
    xor       eax,eax
  .finish:
    ret
endp

section '.data' data readable writeable

  NombreClase TCHAR    'SimpleWinClass',0
  MsgCabecera TCHAR    'Raton', 0
  MsgError    TCHAR    'Carga inicial fallida.',0
  MsgRaton    TCHAR    'Hiciste click aqui',0
  wc          WNDCLASS 0,WndProc,0,0,NULL,NULL,NULL,cdColFondo,NULL,NombreClase
  msg         MSG
```

```
 rect       RECT

 Ps         PAINTSTRUCT

 CommandLine RD      1

 xPos       RD      1

 yPos       RD      1

 hDC        RD      1

 mouseClick db 0


section '.idata' import data readable writeable


  library kernel32,'KERNEL32.DLL',\    ; Importamos las bibliotecas para que el enlazador pueda trabajar

        user32,'USER32.DLL',\       ; Importamos las bibliotecas para que el enlazador pueda trabajar

        gdi32,'GDI32.DLL'           ; Importamos las bibliotecas para que el enlazador pueda trabajar


  include 'api\kernel32.inc'          ; KERNEL32 API calls

  include 'api\user32.inc'            ; USER32 API calls

  include 'api\gdi32.inc'             ; GDI32 API calls
```

El código de pintura en la sección WM_PAINT debe chequear si MouseClick es uno (TRUE), ya que cuando la ventana fue creada, recibió un mensaje WM_PAINT en ese momento, ningún click del ratón había ocurrido aún, así que no dibujará la cadena en el área cliente. Inicializamos MouseClick a FALSE y cambiamos su valor a TRUE cuando ocurre un click del ratón.

Si ha ocurrido al menos un click de ratón, se dibuja la cadena en el área cliente en la posición del ratón. Nota que se llama a lstrlen para obtener el tamaño de la cadena a desplegar y envía este tamaño como último parámetro de la función TextOut.

Tutorial 8: Menú

Teoría:

El menú es uno de los componentes más importantes en nuestra ventana. El menú presenta una lista de los servicios que un programa ofrece a un usuario. El usuario ya no tiene que leer el manual incluido con el programa para utilizarlo, ya que puede leerse cuidadosamente el menú para obtener una visión general de las capacidades de un programa particular y comenzar a trabajar con él inmediatamante. Como el menú es una herramienta para obtener el acercamiento del usuario y correr el programa rápidamente, se debería seguir siempre el estandard. Puesto

sucintamente, los primeros dos elementos [items] del menú deberían ser Archivo [File] y Editar [Edit] y el último debería ser Ayuda [Help]. Puedes insertar tus propios elementos de menú entre Editar y Ayuda. Si un elemento de menú invoca una caja de diálogo, deberías anexar una ellipsis (...) a la cadena del menú.

El menú es un tipo de recurso. Hay varios tipos de recursos, tales como dialog box, string table, icon, bitmap, menú etc. Los recursos son descritos en un archivo separado llamado archivo de recursos, el cual generalmente tiene extensión .rc. Luego combinas los recursos cion el archivo fuente durante el estadío de enlace. El resultado final es un archivo ejecutable que contiene tanto instrucciones como recursos.

Puedes escribir guiones [scripts] de recursos usando un editor de texto. Estos guiones están compuestos por frases que describen la apariencia y otros atributos de los recursos usados en un programa particular. Aunque puedes escribir guiones de recursos con un editor de texto, esto resulta más bien embarazoso. Una mejor altrenativa es usar un editor de recursos que te permita visualizar con facilidad el diseño de los recursos. Usualmente los paquetes de compiladores como Visual C++, Borland C++, etc, incluyen editores de recursos

Describes los recursos más o menos así:

```
Mymenu  MENU
{
  [menu list here]
}
```

Los programadores en lenguaje C pueden reconocer que es similar a la declaración de una estructura. **MyMenu** sería un nombre de menú seguido por la palabra clave **MENU** y una lista de menú entre llaves. Alternativamente, si quieres puedes usar BEGIN y END en vez de las llaves. Esta sintaxis es mas portable para los programadores en Pascal.

La lista de menú puede ser un enunciado **MENUITEM** o **POPUP**.

El enunciado **MENUITEM** define una barra de menú que no invoca un menú emetgente [popup] cuando es seleccionado. La sintaxis es como sigue:

MENUITEM "&text", ID [,options]

Se comienza por la palabra clave MENUITEM seguida por el texto que quieres usar como cadena de texto de la barra de menú. Nota el ampersand (&). Hace que el carácter que le sigue sea subrayado.

Después de la cadena de texto está el ID del elemento de menú. El ID es un número que será usado para identificar el elemento de menú respectivo en el mensaje enviado al procedimiento de ventana cuando el elemento de menú es selccionado. Como tal, cada ID de menú debe ser único entre ellos.

Las opciones son 'opcionales'. Las disponibles son:

- **GRAYED** El elemento del menú está inactivo, y no genera un mensaje WM_COMMAND. El texto está difuminado [grayed].
- **INACTIVE** El elemento del menú está inactivo, y no genera un mensaje WM_COMMAND. El texto es desplegado normalmente.
- **MENUBREAK** Este elemento y los siguientes aparecen en una línea nueva del menú.
- **HELP** Este elemento y los siguientes están justificados a la derecha.
-

Puedes usar una de las opciones de arriba o combinarlas con el operador "or". Sólo recuerda que **INACTIVE** y **GRAYED** no pueden ser combinados simultáneamente.

El enunciado **POPUP** tiene la siguiente sintaxis:

POPUP "&text" [,options]
{
[menu list]
}

El enunciado **POPUP** define una barra de menú que, cuando es seleccionada, despliega una lista de elementos de menú en una ventana emergente. La lista de menú puede ser una enunciado **MENUTIEM** o **POPUP**. Hay un tipo especial de enunciado **MENUITEM**, **MENUITEM SEPARATOR**, que dibuja una linea horizontal en la ventana emergente.

Ahora tienes suficiente información para usar el menú. Vamos a hacerlo.

Ejemplo:

---

menu.asm

```
format PE GUI 4.0

entry inicio


include 'Win32a.inc'


ventana_1       equ             1

salir_btn       equ             1000
```

```
section '.data' data readable writeable

Ist  dd 0
pWnd dd 0
hola db "Hola :D",0
adios db "Adios",0
testC db "Esto es un test",0

section '.code' code readable executable

inicio:
        invoke GetModuleHandle,0
        mov [Ist],eax
        invoke DialogBoxParam,eax,ventana_1,0,DlgProc,0
fin:
        invoke ExitProcess,0

proc DlgProc,hWnd,uMsg,wParam,lParam
        push edi esi ebx

        mov eax,[uMsg]
        cmp     eax,WM_COMMAND
        je      jCOMMAND
        cmp     eax,WM_INITDIALOG
        je      jINITDIALOG
        cmp     eax,WM_CLOSE
        je      jCLOSE
        xor eax,eax
        jmp finish
```

```
jINITDIALOG:
        mov eax,[hWnd]
        mov [pWnd],eax

        mov eax,1
        jmp finish

jCOMMAND:
        mov eax,[wParam]
        cmp     eax,salir_btn
        je      salir
        cmp eax,10002
        je mostrarHello
        cmp eax,10003
        je mostrarBye
        cmp eax,10004
        je salir
        cmp eax,10005
        je mostrarTest
        xor eax,eax
        jmp finish
        mostrarHello:
                invoke MessageBox,[hWnd],hola,0,MB_OK
                mov eax,1
                jmp finish
        mostrarBye:
                invoke MessageBox,[hWnd],adios,0,MB_OK
                mov eax,1
```

```
                jmp finish
        mostrarTest:
                invoke MessageBox,[hWnd],testC,0,MB_OK
                mov eax,1
                jmp finish
salir:
        invoke SendMessage,[hWnd],WM_CLOSE,0,0
        mov eax,1
        jmp finish

jCLOSE:
        invoke EndDialog,[hWnd],0
        mov eax,1

finish:
        pop ebx esi edi
        ret
endp

section '.idata' import data readable writeable
 ;importamos solo los procedimientos basicos para abrir una ventana

 ;otra forma para importar los procedimientos importando archivos INC
 library kernel32,'KERNEL32.DLL',user32,'USER32.DLL',gdi32,'GDI32.DLL'
 include 'api\kernel32.inc'
 include 'api\user32.inc'
 include 'api\gdi32.inc'

section '.rsrc' resource from 't8.res' data readable
```

---

menu.rc

```
#define IDM_popUp 10001
#define IDM_sayHello 10002
#define IDM_SayGoodBye 10003
#define IDM_Exit 10004
#define IDM_test 10005
10000 MENUEX
BEGIN
 POPUP "PopUP",IDM_popUp
 BEGIN
  MENUITEM "SayHello",IDM_sayHello
  MENUITEM "SayGoodBye",IDM_SayGoodBye
  MENUITEM "Exit",IDM_Exit
 END
 MENUITEM "Test",IDM_test
END
```

Tutorial 9: Controles de Ventanas Hijas

Teoría:

Windows provee algunas clases de ventana predefinidas que podemos usar satisfactoriamente dentro de nuestros programas. Muchas veces las usamos como componentes de una caja de diálogo por lo que ellas usualmente son llamadas controles de ventanas hijas. Los controles de ventanas hijas procesan sus propios mensajes de teclado y de ratón y notifican a las ventanas padres cuando sus estados han cambiado. Ellos liberan al programador de enormes cargas, así que deberías usarlas cada vez que sea posible. En este tutorial las pongo sobre una ventana normal para demostrar cómo puedes crearlas y usarlas, pero en realidad deberías ponerlas en una caja de diálogo.

Ejemplos de clases de ventanas predefinidas son el botón, la caja de lista [listbox], la caja de chequeo [checkbox], botón de radio [radio button], edición [edit] etc.

Con el fin de usar el control de vemtana hija, debes crearla con CreateWindow o CreateWindowEx. Nota que no tienes que registrar la clase de ventana puesto que Windows lo hace por tí. El parámetro nombre de la clase DEBE ser el nombre de la clase predefinida. Es decir, si quieres crear un botón, debes especificar "button" como nombre de la clase en CreateWindowEx. Los otros parámetros que debes llenar son la agarradera o manejador [handle] de la ventana padre y el ID del control. El ID del control debe ser único entre los controles. El ID del control es el ID de ese control. Lo usas para diferenciar entre controles.

Después de que el control fue creado, enviará mensajes de notificación a la ventana padre cuando su estado cambie. Normalmente, creas las ventanas hijas durante el mensaje WM_CREATE de la ventana padre. La ventana hija envía mensajes WM_COMMAND a la ventana padre con su ID de control en la palabra baja de wParam, el código de notificación en la palabra alta de wParam, y su manejador de ventana en lParam. Cada conrtol de ventana hija tiene su propio código de notificación, así que debes revisar la referencia de la API de Win32 para más información.

También la ventana padre puede enviar órdenes o comandos a la ventana hija, llamando a la función SendMessage. Esta función envía el mensaje especificado acompañado de otrops valores en wParam y lParam a la ventana especificada por el manejador de ventana. Es una función extremadamente útil, ya que puede enviar mensajes a cualquier ventana que conozcas su manejador.

Así que después de crear ventanas hijas, la ventana padre debe procesar los mensajes WM_COMMAND para poder recibir códigos de notificación desde las ventanas hijas.

Ejemplo:

Crearemos una ventana que contenga un control de edición y un "pushbutton". Cuando pulses el botón, un cuadro de mensaje aparecerá mostrando un texto que hayas escrito en el cuadro de diálogo.

```
format PE GUI 4.0

entry start


include 'win32a.inc'    ; Incluímos definiciones de estructuras y constantes

;WinMain proto dword,dword,dword,dword


cdXPos      EQU  128        ; Constante double X-Posición de la ventana(esq sup izqda)

cdYPos      EQU  128        ; Constante double Y-Posición de la ventana(esq sup izqda)

cdXSize     EQU  320        ; Constante double X-tamaño de la ventana
```

```
cdYSize      EQU  200        ; Constante double Y-tamaño de la ventana
cdColFondo   EQU  COLOR_BTNFACE + 1  ; Color de fondo de la ventana: gris de un botón de comando
cdVIcono     EQU  IDI_APPLICATION ; Icono de la ventana, véase Resource.H
cdVCursor    EQU  IDC_ARROW   ; Cursor para la ventana
; Tipo de ventana (Barra de cabecera)
;cdVBarTipo  EQU  WS_EX_TOOLWINDOW                ; Tipo de barra de Cabecera: delgado, sin icono, sin reflejo en barra de tareas
cdVBarTipo   EQU  NULL                           ; Normal, con icono
cdVBtnTipo   EQU  WS_VISIBLE+WS_DLGFRAME+WS_SYSMENU  ; Normal sólo con botón cerrar
cdID_MSG     EQU  500
cdID_SALIR   EQU  501

section '.text' code readable executable
 start:
   invoke   GetModuleHandle,0
   mov      [wc.hInstance],eax
   invoke   GetCommandLine
   mov      [CommandLine], EAX
   stdcall  WinMain, [wc.hInstance], NULL, [CommandLine], SW_SHOWDEFAULT
   invoke   ExitProcess,[msg.wParam]

proc WinMain uses ebx esi edi, hInst, hPrevInst, CmdLine, CmdShow
   ;  Propósito: Inicializamos la ventana principal de la aplicación y captura errores, si los hubiere
   ;  Entrada  : hInst, hPrevInst, CmdLine, CmdShow
   ;  Salida   : Ninguna
   ;  Destruye : Ninguna
   invoke   LoadIcon,0,cdVIcono
   mov      [wc.hIcon],eax
   invoke   LoadCursor,0,cdVCursor
```

```
  mov       [wc.hCursor],eax
  invoke    RegisterClass,wc
  test      eax,eax
  jz        error

  invoke    CreateWindowEx,cdVBarTipo,NombreClase,MsgCabecera,\
            cdVBtnTipo,cdXPos, cdYPos, cdXSize, cdYSize,\
            NULL,NULL,[wc.hInstance],NULL
  test      eax,eax
  jz        error

  msg_loop:
    invoke    GetMessage,msg,NULL,0,0
    cmp       eax,1
    jb        end_loop
    jne       msg_loop
    invoke    TranslateMessage,msg
    invoke    DispatchMessage,msg
  jmp       msg_loop

 error:
  invoke    MessageBox,NULL,MsgError,NULL,MB_ICONERROR+MB_OK

 end_loop:
  MOV       EAX, [msg.wParam]
  ret
endp

proc WndProc uses ebx esi edi, hwnd,wmsg,wparam,lparam
```

```
  ;  Propósito: Procesa los mensajes provenientes de las ventanas
  ;  Entrada  : hwnd,wmsg,wparam,lparam
  ;  Salida   : Ninguna
  ;  Destruye : Ninguna
  ;  Si usáramos WM_PAINT, al menos aquí habría que redireccionar a wmDefault para que funcione

  mov      eax, [wmsg]
  cmp      eax, WM_DESTROY
  jz       wmDestroy
  cmp      eax, WM_COMMAND
  jz       wmCommand
  cmp      eax, WM_CREATE
  jz       wmCreate

  wmDefault:
     invoke   DefWindowProcA, [hwnd], [wmsg], [wparam], [lparam]
     jmp      wmFin
  wmCreate:
     invoke   CreateWindowEx, NULL, ClaseBoton, MsgbtnTexto1, WS_CHILD + WS_VISIBLE + BS_DEFPUSHBUTTON,\
              90, 130,125, 30, [hwnd], 500, [wc.hInstance], NULL
      invoke   CreateWindowEx, NULL, claseEdit, 0, WS_CHILD + WS_VISIBLE+WS_BORDER,\
              80, 70, 175, 30, [hwnd], 501, [wc.hInstance], NULL
      mov [hedit1],eax

     jmp      wmFin
  wmCommand:
     cmp      [wparam], 500
     je       btn_Mensaje
```

```
      jmp     wmDefault
      btn_Mensaje:
        invoke SendMessage,[hedit1],WM_GETTEXT,50,texto
        invoke   MessageBox, NULL, texto, MsgCabBoton, MB_OK
        jmp     wmFin
      btn_Salir:
        jmp     wmDestroy
      jmp     wmFin
   wmDestroy:
      invoke   PostQuitMessage,0
   wmFin:
   ret
endp

section '.data' data readable writeable

 MsgCabecera     db    "Control", 0
 NombreClase     db    "SimpleWinClass", 0
 MsgError        db    "Carga inicial fallida.",0
 MsgCabBoton     db    "El texto",0
 ClaseBoton      db    "BUTTON", 0
 claseEdit       db    "EDIT",0
 MsgbtnTexto1    db    "My First Button", 0

 texto          db  50 dup(?)
 wc             WNDCLASS 0,WndProc,0,0,NULL,NULL,NULL,cdColFondo,NULL,NombreClase
 msg            MSG
 rct            RECT     NULL, NULL, NULL, NULL
 CommandLine     rd    1
```

```
  wHandle       rd   1
  hedit1        dd ?


section '.idata' import data readable writeable


  library kernel32,'KERNEL32.DLL',\    ; Importamos las bibliotecas para que el enlazador pueda
trabajar
         user32,'USER32.DLL'          ; Importamos las bibliotecas para que el enlazador pueda trabajar


  include 'api\kernel32.inc'          ; KERNEL32 API calls
  include 'api\user32.inc'            ; USER32 API calls
```

Tutorial 10: Caja de Diálogo [Dialog Box] como Ventana Principal

Teoría:

Si juegas bastante con los ejemplos del tutorial anterior, encontrarás que no puedes cambiar el foco de entrada de un control de ventana hija a otra con la tecla Tab. La única manera de realizar eso es haciendo click sobre el control que deseas que gane el foco de entrada. Esta situación es más bien incómoda. Otra cosa que deberías notar es que cambié el color del fondo de la ventana padre a gris en vez de a blanco, como lo había hecho en los ejemplos previos. Esto se hace así para que el color de la ventana hija pueda armonizar con el color del área cliente del ventana padre. Hay otra manera de salvar este problema pero no es fácil. Tienes que subclasificar todos los controles de ventana hija en tu ventana padre.

La razón de la existencia de tal inconveniente es que los controles de ventana hija están originalmente diseñados para trabajar dentro de cajas de diálogo, no en una ventana normal. Los colores por defecto de los controles de ventanas hijas, como los botones, es gris porque el área cliente de la caja de diálogo normalmente es gris para que armonicen entre sí sin ninguna intervensión por parte del programador.

Antes de entrar en detalles, deberíamos saber qué es una caja de diálogo. Una caja de diálogo no es más que una ventana normal diseñada para trabajar con controles de ventanas hijas. Windows también proporciona un administrador interno de cajas de diálogo ["dialog box manager"] responsable por gran parte de la lógica del teclado tal como desplazamiento del foco de entrada cuando el ususario presiona Tab, presionar el botón por defecto si la tecla Enter es presionada, etc; así los programadores pueden ocuparse de tareas de más alto nivel. Las cajas de diálogo son usadas primero como dispositivos de entrada/salida. Como tal, una caja de diálogo puede ser considerada como una "caja negra" de entrada/salida lo que siginifica que no tienes que saber cómo funciona internamente una caja de diálogo para usarla, sólo tienes que saber cómo interactuar con ella. Es un principio de la programación orientada a objetos [object oriented

programming (OOP)] llamado encapsulación u ocultamiento de la información. Si la caja negra es *perfectamente* diseñada , el usuario puede emplarla sin tener conocimiento de cómo funciona. Lo único es que la caja negra debe ser perfecta, algo difícil de alcanzar en el mundo real. La API de Win32 API también ha sido diseñada como una caja negra.

Bien, parece que nos hemos alejado de nuestro camino. Regresemos a nuestro tema. Las cajas de diálogo han sido diseñadas para reducir la carga de trabajo del programador. Normalmente si tienes que poner controles de ventanas hijas sobre una ventana normal, tienes que subclasificarlas y escribir tú mismo la lógica del teclado. Pero si quieres ponerlas en una caja de diálogo, Windows manejará la lógica por tí. Sólo tienes que saber cómo obtener la entrada del usuario de la caja de diálogo o como enviar órdenes a ella.

Como el menú, una caja de diálogo se define como un recurso. Escribes un plantilla describiendo las características de la caja de diálogo y sus controles y luego compilas el guión de recursos con un compilador de recursos.

Nota que todos los recursos se encuentran en el mismo archivo de guión de recursos. Puedes emplear cualquier editor de texto para escribir un guión de recursos, pero no lo recomiendo. Deberías usar un editor de recursos para hacer la tarea visualmente ya que arreglar la disposición de los controles en la caja de diálgo es una tarea dura de hacer manualmente. Hay disponibles algunos excelentes editores de recursos. Muchos de las grandes suites de compiladores incluyen sus propios editores de recursos. Puedes usar cualquiera para crear un guión de recursos para tu programa y luego cortar las líneas irrelevantes tales como las relacionadas con MFC.

Hay dos tipos principales de cajas de diálogo: modal y no-modal. Una caja de diálogo no-modal te deja cambiar de foco hacia otra ventana. Un ejempo es el diálogo Find de MS Word. Hay dos subtipos de caja de diálogo modal: modal de aplicación y modal de sistema. Una caja de diálogo modal de aplicación no permite cambiar el foco a otra ventana en la misma aplicación sino cambiar el foco de entrada a la ventana de **OTRA** aplicación. Una caja de diálogo modal de sistema no te permite cambiar de foco hacia otra ventana hasta que respondas a la primera.

Una caja de diálogo no-modal se crea llamando a la función de la API CreateDialogParam. Una caja de diálogo modal se crea llamando a DialogBoxParam. La única diferencia entre una caja de diálogo de no-modal y una modal de sistema es el estilo DS_SYSMODAL. Si quieres incluir el estilo DS_SYSMODAL en una plantilla de caja de diálogo, esa caja de diálogo será modal de sistema.

Puedes comunicarte con cualquier control de ventana hija sobre una caja de diálogo usando la función SendDlgItemMessage. Su sintaxis es:

```
SendDlgItemMessage proto hwndDlg:DWORD,\
                         idControl:DWORD,\
                         uMsg:DWORD,\
                         wParam:DWORD,\
                         lParam:DWORD
```

Esta llamada a la API es inmensamene útil para interactuar con un control de ventana hija. Por ejemplo, si quieres obtener el texto de un control de edición, puedes hacer esto:

call SendDlgItemMessage, hDlg, ID_EDITBOX, WM_GETTEXT, 256, ADDR text_buffer

Con el fin de saber qué mensaje enviar, deberías consultar la referencia de la API de Win32. Windows también provee algunas funciones específicas de la API para controles que permiten obtener y poner datos en los controles rápidamente, por ejemplo, GetDlgItemText, CheckDlgButton etc. Estas funciones específicas para controles son suministradas para conveniencia de los programadores de manera que él no tenga que revisar el significado de wParam y lParam para cada mensaje. Normalmente, deberías usar llamadas a las funciones específicas de la API para controles cada vez que sean disponibles ya que ellas facilitan el mantenimiento del código fuente. Recurre a SendDlgItemMessage sólo si no hay disponible llamadas a funciones específicas de la API.

El manejador de Windows de cajas de diálogos envía varios mensajes a una función " callback" particular llamada procedimiento de caja de diálogo que tiene el siguiente formato:

```
DlgProc  proto hDlg:DWORD ,\
               iMsg:DWORD ,\
               wParam:DWORD ,\
               lParam:DWORD
```

El procedimiento de diálogo es similar al procedimiento de ventana excepto por el tipo de valor de retorno, que es TRUE/FALSE en vez de LRESULT. El administrador interno de la caja de diálogo dentro de Windows **ES** el verdadero procedimiento de ventana para la caja de diálogo. Llama a nuestra caja de diálogo con algunos mensajes que recibió. Así que la regla general es que: si nuestro procedimiento de diálogo procesa un mensaje, **DEBE** regresar TRUE (1) en eax y si no procesa el mensaje, debe regresar FALSE (0) en eax. Nota que un procedimiento de caja de diálogo no pasa los mensajes no procesados a la llamada DefWindowProc ya que no es realmente un procedimiento de ventana.

Hay dos usos distintos de una caja de diálogo. Puedes usarlas como ventanas principal de tu aplicación o usarla como un dispositivo de entrada. Examinaremos el primer acercamiento en este tutorial. "Usar una caja de diálogo como una ventana principal" puede ser interpretado en dos sentidos.

1. Puedes usar una plantilla de caja de diálogo como la plantilla de clase que registras al llamar a RegisterClassEx. En este caso, la caja de diálogo se comporta como una ventana "normal": recibe mensajes del procedimiento de ventana referido por el miembro lpfnWndProc de la clase de ventana class, no a través de un procedimiento de caja de diálogo. El beneficio de esto es que no tienes que crear por ti mismo controles de ventana hija, Windows los crea por ti cuando se crea la caja de diálogo. También Windows maneja la lógica del teclado para ti, por ejemplo se encarga de la orden Tab, etc. Además puedes especificar el cursor y el icono de tu ventana en la estructura de la clase de ventana.

Tu programa crea la caja de diálogo sin ninguna ventana padre. Esta aproximación al problema hace innecesario el uso de un bucle de mensajes ya que los mensajes son enviados directamente al procedimiento de ventana de la caja de diálogo. ¡Ya no tienes que registrar la clase de ventana!

Ejemplos:

```
format PE GUI 4.0
entry inicio

include 'Win32a.inc'

ventana_1       equ             1
salir_btn       equ             1000

section '.data' data readable writeable

Ist  dd 0
pWnd dd 0
Msg db "Hola",0
hello db "Wow! I'm in an edit box now",0
buffer db 512 dup(?)
section '.code' code readable executable

inicio:
        invoke GetModuleHandle,0
        mov [Ist],eax
        invoke DialogBoxParam,eax,ventana_1,0,DlgProc,0
fin:
        invoke ExitProcess,0

proc DlgProc,hWnd,uMsg,wParam,lParam
        push edi esi ebx

        mov eax,[uMsg]
```

```
        cmp     eax,WM_COMMAND
        je      jCOMMAND
        cmp     eax,WM_INITDIALOG
        je      jINITDIALOG
        cmp     eax,WM_CLOSE
        je      jCLOSE
        xor eax,eax
        jmp finish

jINITDIALOG:
        mov eax,[hWnd]
        mov [pWnd],eax

        mov eax,1
        jmp finish

jCOMMAND:
        mov eax,[wParam]
        cmp     eax,salir_btn
        je      salir
        cmp eax,10002
        je obtenerTexto
        cmp eax,10003
        je limpiar
        cmp eax,10004
        je salir
        cmp eax,1002
        je sayH
        xor eax,eax
```

```
        jmp finish
obtenerTexto:
        invoke GetDlgItemText,[hWnd],1001,buffer,512
    invoke   MessageBox, [hWnd], buffer, 0, MB_OK
        mov eax,1
        jmp finish
limpiar:
        invoke SetDlgItemText,[hWnd],1001,NULL
        mov eax,1
        jmp finish
sayH:

        invoke SetDlgItemText,[hWnd],1001,hello
        mov eax,1
        jmp finish

salir:
        invoke SendMessage,[hWnd],WM_CLOSE,0,0
        mov eax,1
        jmp finish

jCLOSE:
        invoke EndDialog,[hWnd],0
        mov eax,1

finish:
        pop ebx esi edi
        ret
endp
```

```
section '.idata' import data readable writeable

  ;otra forma para importar los procedimientos importando archivos INC
  library kernel32,'KERNEL32.DLL',user32,'USER32.DLL',gdi32,'GDI32.DLL'
  include 'api\kernel32.inc'
  include 'api\user32.inc'
  include 'api\gdi32.inc'

section '.rsrc' resource from 't10F.res' data readable
```